TÜBİTAK

Turk J Elec Eng & Comp Sci, Vol.20, No.4, 2012, © TÜBİTAK
doi:10.3906/elk-1009-756

Fast software multiplication in $\mathbb{F}_2[x]$ for embedded processors

Serdar Süer ERDEM
Gebze Institute of Technology, 41400 Çayırova, Gebze, Kocaeli-TURKEY
e-mail: serdem@gyte.edu.tr

Received: 14.09.2010

Abstract

We present a novel method for fast multiplication of polynomials over $\mathbb{F}_2$ which can be implemented efficiently in embedded software. Fast polynomial multiplication methods are needed for the efficient implementation of some cryptographic and coding applications. The proposed method follows a strategy to reduce the memory accesses for input data and intermediate values during computation. This strategy speeds up the binary polynomial multiplication significantly on typical embedded processors with limited memory bandwidth. These multiplications are usually performed by the comb method or the Karatsuba-based methods in embedded software. The proposed method has speed and memory advantages over these methods on embedded platforms for the polynomial degrees usually encountered in practical cryptosystems. We perform a detailed complexity analysis of the proposed method and complexity comparisons with the other methods. Finally, we present the running times of the proposed method and its alternatives on ARM7TDMI processor.

Key Words: *Finite fields, computer arithmetic, cryptography, algorithms*

1. Introduction

The multiplication of polynomials over $\mathbb{F}_2$ has important applications in number theory and cryptography. For example, it is extensively used in factorization of polynomials over $\mathbb{F}_2$ [1, 2] and elliptic curve cryptosystems [3, 4, 5].

The comb method [4, 6], the Karatsuba algorithm [5, 7, 8], and the Karatsuba-like formulae [9] are very suitable for multiplication of polynomials over $\mathbb{F}_2$ in software, if polynomial degrees do not exceed one thousand. Actually, this is the case for most practical applications. For multiplication of larger polynomials, fast subquadratic algorithms are given in [10]. However, these algorithms are inefficient when applied to the multiplication of small polynomials. Therefore, the practical implementations of these algorithms also include some implementation of the comb method or some Karatsuba-based multiplication method.

This work proposes an efficient alternative to the comb method and the Karatsuba-based multiplication methods to multiply the polynomials over $\mathbb{F}_2$ of several hundred degrees in software. Practical cryptosystems generally deal with the polynomials in this range. The proposed method has some similarities to the comb method. However, the comb method aims at decreasing the number of shifts performed on computer words

while the proposed method aims at decreasing the number of memory load and store operations. Since the load and store operations are more costly than shift operations in a typical embedded processor, the proposed method is very suitable for embedded applications. According to simulation results on ARM7TDMI processors, the proposed method is faster than the comb method and the Karatsuba algorithm, while it shows similar performance to the Karatsuba-like formulae. However, the Karatsuba-like formulae consist of several specialized multiplication methods splitting their inputs into 3, 4, 5, 6, and 7 parts. Thus, its code size is several times more than the proposed method.

Remainder of this paper is organized as follows. Section 2 introduces the polynomials over $\mathbb{F}_2$. Section 3 describes the comb method. Section 4 presents the proposed method. Section 5 compares the complexities of the proposed method and its alternatives. Section 6 provides timing data and performance comparisons.

2. Polynomials over $\mathbb{F}_2$

The polynomials over $\mathbb{F}_2$ are the polynomials whose coefficients are the elements of $\mathbb{F}_2$. The field $\mathbb{F}_2$ consists of only the integers $\{0, 1\}$ under modulo 2 addition and multiplication.

The addition and subtraction of polynomials over $\mathbb{F}_2$ can be performed by simply XORing their corresponding binary valued coefficients. This is because modulo 2 addition and subtraction on the set $\{0, 1\}$ are both equivalent to the logical XOR operation.

The multiplication of polynomials over $\mathbb{F}_2$ requires both coefficient additions and coefficient multiplications. While coefficient additions are performed by logical XOR operations, as mentioned above, coefficient multiplications are performed by logical AND operations. This is because modulo 2 multiplication on the set $\{0, 1\}$ is equivalent to logical AND.

2.1. Multiprecision representation

Let w denote the size of a computer word. Then, each computer word can store w polynomial coefficients since the coefficients are binary 0 or 1. Thus, the polynomial

$$F_i = \sum_{j=0}^{w-1} f_{iw+j} x^j,$$

defined from the w-consecutive coefficients of a polynomial $f(x)$, can also be viewed as a w-bit word:

$$F_i = f_{iw+w-1} \ , \ \ldots \ , \ f_{iw+j} \ , \ \ldots \ , \ f_{iw+1} \ , \ f_{iw} \ .$$

Note that $f(x)$ has the multiprecision representation

$$f(x) = F_{\mathcal{N}-1} x^{(\mathcal{N}-1)w} + \ldots + F_1 x^w + F_0 \tag{1}$$

for some single word polynomials $F_{\mathcal{N}-1}$, ... , F_2, F_1, F_0. Naturally, $\mathcal{N}$ must satisfy the inequality $x^m \leq x^{w\mathcal{N}}$ for $m = \deg(f(x))$.

2.2. Multiplication by powers of x

This section shows how the multiplications by powers of x are performed by word and bit level shifts in practical implementations.

2.2.1. Computation $g(x) = x^k f(x)$ for $k < w$

Let $g(x) = x^k f(x) < x^{\mathcal{N}w}$. Then, we can use the notation $f(x) = \sum_{i=0}^{\mathcal{N}-1} F_i x^{iw}$ and $g(x) = \sum_{i=0}^{\mathcal{N}-1} G_i x^{iw}$ for some words F_i and G_i. The word level computations for the polynomial operations can be given as

$$\begin{aligned} G_i &= x^k F_i \bmod x^w + \lfloor x^k F_{i-1}/x^w \rfloor \\ &= (F_i \ll k) \text{ XOR } (F_{i-1} \gg (w-k)), \end{aligned}$$

where $F_{-1} = 0$ by the definition, while "$\ll$" and "$\gg$" denote logical left and right shifts, respectively. The pseudo code in Figure 1 implements this computation.

```
U = F_0
G_0 = U << k
for i = 1 to N - 1
    H = U >> (w - k)
    U = F_i
    G_i = (U << k) XOR H
```

Figure 1. Pseudo Code for the computation of $g(x) = x^k f(x)$, where $x^k < x^w$ and $x^k f(x) < x^{\mathcal{N}w}$.

2.2.2. Computation $g(x) = g(x) + f(x)x^{iw+k}$ for $k < w$

Let $f(x) = \sum_{j=0}^{\mathcal{N}-1} F_j x^{jw} < x^{\mathcal{N}w}$. Then,

$$g(x) = g(x) + \textstyle\sum_{j=0}^{\mathcal{N}-1} x^k F_j x^{(i+j)w}.$$

The word level computations for the polynomial operations can be given as

$$\begin{aligned} G_{i+j} &= G_{i+j} + x^k F_j \bmod x^w + \lfloor x^k F_{j-1}/x^w \rfloor \\ &= G_{i+j} \text{ XOR } (F_j \ll k) \text{ XOR } (F_{j-1} \gg (w-k)) \end{aligned},$$

where $F_{-1} = 0$ by the definition. The pseudo code in Figure 2 implements this computation.

Table 1 gives the numbers of the operations for the pseudo codes given in this section. Table 1 includes only the cost of the loads and stores required to access F_i and G_i. The cost of accessing variables U and H is excluded, since they can be stored in registers.

Table 1. The Complexities of the Pseudo Codes in Figure 1 and 2.

	XOR	Shift	Load/store
Pseudo code 1	$\mathcal{N}-1$	$2\mathcal{N}-1$	$2\mathcal{N}$
Pseudo code 2	$2\mathcal{N}$	$2\mathcal{N}$	$3\mathcal{N}+2$

$U = F_0$

$G_i = G_i$ XOR $(U \ll k)$

for $j = 1$ **to** $\mathcal{N} - 1$

$H = U \gg (w - k)$

$U = F_j$

$G_{i+j} = G_{i+j}$ XOR $(U \ll k)$ XOR H

$H = U \gg (w - k)$

$G_{i+\mathcal{N}} = G_{i+\mathcal{N}}$ XOR H

Figure 2. Pseudo Code for the computation of $g(x) = g(x) + f(x)x^{iw+k}$, where $x^k < x^w$ and $f(x) < x^{\mathcal{N}w}$.

3. Comb method

The comb method is a fast and efficient technique for multiplication in $\mathbb{F}_2[x]$ [4, 6]. Let $d(x) = a(x)b(x)$. If $a(x)$ and $b(x)$ are represented as shown in (1), then

$$\begin{aligned} d(x) &= \textstyle\sum_{i=0}^{n-1} A_i x^{iw} \sum_{j=0}^{n-1} B_j x^{jw} \\ &= \textstyle\sum_{i=0}^{n-1} A_i \sum_{j=0}^{n-1} B_j x^{(i+j)w} \\ &= \textstyle\sum_{k=0}^{w-1} x^k \sum_{i=0}^{n-1} \underbrace{a_{iw+k}}_{k\text{th bit of } A_i} \sum_{j=0}^{n-1} B_j x^{(i+j)w}. \end{aligned}$$

This formula leads to the left-to-right comb method given in Figure 3. The shift operation $d(x) = xd(x)$ in Step 6 of this method is implemented by the pseudo code in Figure 1 for $k = 1$, $\mathcal{N} = 2n$, and $g(x) = f(x) = d(x)$.

INPUT: $a(x)$ and $b(x)$ over $\mathbb{F}_2$ of degree less than nw.

OUTPUT: $d(x) = a(x)b(x)$ over $\mathbb{F}_2$ of degree less than $2nw$.

1. $D_i = 0$ for $0 \le i < 2n$
2. **for** $k = w - 1$ **downto** 0
3. **for** $i = 0$ **to** $n - 1$
4. **if** the kth bit of A_i is 1
5. $D_{i+j} = D_{i+j} + B_j$ for $0 \le j < n$
6. $d(x) = xd(x)$, if $k \neq 0$.

Figure 3. Left-to-right Comb Method.

In Figure 3, the bits (coefficients) of $a(x)$ are scanned one bit at a time in Step 4. Thus, its window size $\mathcal{W} = 1$. The algorithm in Figure 4 is a faster implementation of the comb method, scanning the bits of $a(x)$ with a larger window ($\mathcal{W} > 1$) and multiply them by $b(x)$ using a lookup table.

INPUT: $a(x) < x^{nw}$ and $b(x) < x^{nw-\mathcal{W}+1}$ over $\mathbb{F}_2$.

OUTPUT: $d(x) = a(x)b(x)$ over $\mathbb{F}_2$

1. $D_i = 0$ for $0 \leq i < 2n$
2. Compute and store $b'(x) = \varepsilon(x)b(x)$ for all $\varepsilon(x) < x^{\mathcal{W}}$
3. **for** $k = \mathcal{W}(\lceil w/\mathcal{W} \rceil - 1)$ **downto** 0 **by** $\mathcal{W}$
4. **for** $i = 0$ **to** $n - 1$
5. $\varepsilon(x) = \lfloor A_i/x^k \rfloor \bmod x^{\mathcal{W}}$
6. Find from lookup $b'(x) = \varepsilon(x)b(x)$
7. $D_{i+j} = D_{i+j} + B'_j$ for $0 \leq j < n$
8. $d(x) = x^{\mathcal{W}}d(x)$, if $k \neq 0$

Figure 4. Comb Method for $\mathcal{W} > 1$.

Step 2 in Figure 4 multiplies the input $b(x)$ by all the polynomials $\varepsilon(x) < x^{\mathcal{W}}$ and stores the results $b'(x) = \varepsilon(x)b(x) < x^{nw}$ into a lookup table. Note that this table has n-word entries, i.e. $b'(x) = \sum_{j=0}^{n-1} B'_j x^{jw}$.

Step 5 extracts the $\mathcal{W}$ term $\varepsilon(x)$ from A_i as follows:

$$\begin{aligned} \varepsilon(x) &= \lfloor A_i/x^k \rfloor \bmod x^{\mathcal{W}} \\ &= \lfloor (\textstyle\sum_{j=0}^{w-1} a_{iw+j}x^j)/x^k \rfloor \bmod x^{\mathcal{W}} \\ &= \textstyle\sum_{j=0}^{\mathcal{W}-1} a_{iw+k+j}x^j \ . \end{aligned}$$

Because A_i and $\varepsilon(x)$ are stored as sequences of w and $\mathcal{W}$ bits, respectively,

$$\lfloor A_i/x^k \rfloor \bmod x^{\mathcal{W}} \equiv (A_i \gg k) \text{ AND } (2^{\mathcal{W}} - 1) = I \ .$$

Here, I is the lookup table entry storing the product $b'(x) = \varepsilon(x)b(x)$. Steps 6 and 7 take this product from the lookup and add it to the running sum. Step 8 computes $d(x) = x^{\mathcal{W}}d(x)$ by carrying out the pseudo code in Figure 1 for $k = \mathcal{W}$, $\mathcal{N} = 2n$, and $g(x) = f(x) = d(x)$.

4. Operand scanning method

In the usual operand scanning multiplication, the terms of one operand are scanned successively from the lowest order to the highest order. Each $\mathcal{W}$ terms of this operand are multiplied by the other operand. The resulting partial products are appropriately shifted and accumulated to a running sum. Since the terms are scanned in successive order, it is easy to multiply the consecutive terms stored in the same computer word together and interleave the resulting partial products. In contrary, the comb method does not scan the polynomial terms in successive order, as can be seen from Figures 3 and 4. In these algorithms, bit position k varies in the most outer loop. The inner loops just perform the multiplication for a fixed k value. That is the polynomial terms stored in the same bit positions are scanned together to reduce the bit level shifts.

Let $d(x) = a(x)b(x)$. If $a(x)$ and $b(x)$ are represented as shown in (1), then

$$\begin{aligned} d(x) &= \sum_{i=0}^{n-1} A_i x^{iw} b(x) \\ &= \sum_{i=0}^{n-1} \sum_{k=0}^{w-1} \underbrace{a_{iw+k}}_{k^{\text{th}} \text{ bit of } A_i} b(x) x^{iw+k} \end{aligned}$$

This formula leads to the operand scanning method in Figure 5.

INPUT: $a(x)$ and $b(x)$ over $\mathbb{F}_2$ of degree less than nw.

OUTPUT: $d(x) = a(x)b(x)$ over $\mathbb{F}_2$ of degree less than $2nw$.

1. $D_i = 0$ for $0 \leq i < 2n$
2. **for** $i = 0$ **to** $n - 1$
3. $\quad A = A_i$
4. $\quad$ **for** $k = 0$ **to** $w - 1$
5. $\quad\quad$ **if** the kth bit of A is 1
6. $\quad\quad\quad d(x) = d(x) + b(x)x^{iw+k}$

Figure 5. Operand Scanning Method for $\mathcal{W} = 1$.

Practical implementations obtain $d(x) = d(x) + b(x)x^{iw+k}$ in Step 6 of this method, using the pseudo code in Figure 2 for $\mathcal{N} = n$, $f(x) = b(x)$, and $g(x) = d(x)$.

The algorithm in Figure 6 is a faster implementation of the operand scanning multiplication, using a larger window ($\mathcal{W} > 1$) and a lookup table. As in the comb method given in Figure 4, Step 3 of this algorithm computes all the possible products $\varepsilon(x)b(x)$ for the polynomials $\varepsilon(x)$ of degree less than $\mathcal{W}$ and stores the resulting polynomials into a lookup table. Then, the remaining steps multiply $b(x)$ by the terms of $a(x)$, $\mathcal{W}$ terms at a time, using the lookup table.

Step 4 loads A_i to A and Step 6 splits A into the polynomials

$$\begin{aligned} \varepsilon(x) &= \lfloor A/x^k \rfloor \bmod x^{\mathcal{W}} \\ &= \lfloor (\sum_{j=0}^{w-1} a_{iw+j} x^j)/x^k \rfloor \bmod x^{\mathcal{W}} \\ &= \sum_{j=0}^{\mathcal{W}-1} a_{iw+k+j} x^j, \end{aligned}$$

for all $k = 0, \mathcal{W}, 2\mathcal{W}, \ldots, (\lceil w/\mathcal{W} \rceil - 1)\mathcal{W}$. Because A and $\varepsilon(x)$ are stored as sequences of w and $\mathcal{W}$ bits, respectively, in practice, the computation

$$\lfloor A/x^k \rfloor \bmod x^{\mathcal{W}} \equiv (A \gg k) \text{ AND } (2^{\mathcal{W}} - 1) = I \ .$$

Here, I is the lookup table entry storing the product $b'(x) = \varepsilon(x)b(x) < x^{nw}$. The steps 7 and 8 take this product from the lookup and add it to the running sum. The computation $d(x) = d(x) + b'(x)x^{iw+k}$ in Step 8 is achieved by carrying out the pseudo code in Figure 2 for $\mathcal{N} = n$, $f(x) = b'(x)$, and $g(x) = d(x)$.

INPUT: $a(x) < x^{nw}$ and $b(x) < x^{nw-\mathcal{W}+1}$ over $\mathbb{F}_2$.

OUTPUT: $d(x) = a(x)b(x)$ over $\mathbb{F}_2$

1. $D_i = 0$ for $0 \leq i < 2n$
2. Compute and store $b'(x) = \varepsilon(x)b(x)$ for all $\varepsilon(x) < x^{\mathcal{W}}$
3. **for** $i = 0$ **to** $n - 1$
4. $\quad A = A_i$
5. $\quad$ **for** $k = 0$ **to** $\mathcal{W}(\lceil w/\mathcal{W} \rceil - 1)$ **by** $\mathcal{W}$
6. $\quad\quad \varepsilon(x) = \lfloor A/x^k \rfloor \bmod x^{\mathcal{W}}$
7. $\quad\quad$ Find from lookup $b'(x) = \varepsilon(x)b(x)$
8. $\quad\quad d(x) = d(x) + b'(x)x^{iw+k}$

Figure 6. Operand Scanning Method for $\mathcal{W} > 1$.

4.1. Interleaving partial product accumulation

Steps 5, 6, 7 and 8 of the algorithm in Figure 6 can be modified to accumulate Λ partial products together, as seen in Figure 7.

for $k = 0$ **to** $\Lambda\mathcal{W}(\lceil w/\Lambda\mathcal{W} \rceil - 1)$ **by** $\Lambda\mathcal{W}$

6. $\quad \varepsilon^{(\lambda)}(x) = \lfloor A/x^{k+\lambda\mathcal{W}} \rfloor \bmod x^{\mathcal{W}}, \;\; \lambda = 0, \ldots, \Lambda - 1$
7. $\quad$ Find all $b'^{(\lambda)}(x) = \varepsilon^{(\lambda)}(x)b(x)$ in the lookup.
8. $\quad d(x) = d(x) + \sum_{\lambda=0}^{\Lambda-1} b'^{(\lambda)}(x)x^{iw+k+\lambda\mathcal{W}}$

Figure 7. Interleaving the Accumulation of the Partial Products.

Step 6 in Figure 7 processes the terms of A from x^k to $x^{k+\Lambda\mathcal{W}-1}$ in each iteration. Step 6 actually computes the following Λ integers

$$I^{(\lambda)} = (A \gg (k + \lambda\mathcal{W})) \text{ AND } (2^{\mathcal{W}} - 1)$$

for $\lambda = 0, 1, \ldots, \Lambda - 1$ in practical implementations in place of the polynomials $\varepsilon^{(\lambda)}(x)$. $I^{(\lambda)}$ is the entry number for the polynomial $b'^{(\lambda)}(x)$ stored in the lookup.

Step 8 in Figure 7 accumulates the Λ partial products together. Usually, the word size $w = 32$ bits. Also, $\mathcal{W} = 4$ bits is an appropriate choice for the window size. As an example, let $\Lambda = 4$. Then, $\Lambda\mathcal{W} = 16$ and the modified step 8 becomes

$$d(x) = d(x) + \sum_{\lambda=0}^{3} b'^{(\lambda)}(x)x^{iw+k+\lambda 4}$$

for $k = 0$ and $k = 16$. Let $b'^{(\lambda)}(x) = \sum_{j=0}^{n-1} B_j'^{(\lambda)} x^{jw}$. Then, $d(x)$ can be computed in Step 8 as:

$H = 0$
for $j = 0$ **to** $n - 1$
$U = B_j'^{(0)}, \quad L = (U \ll (k + 0))$ XOR H
$H = (U \gg (32 - k))$
$U = B_j'^{(1)}, \quad L = L$ XOR $(U \ll (k + 4))$
$H = H$ XOR $(U \gg (28 - k))$
$U = B_j'^{(2)}, \quad L = L$ XOR $(U \ll (k + 8))$
$H = H$ XOR $(U \gg (24 - k))$
$U = B_j'^{(3)}, \quad L = L$ XOR $(U \ll (k + 16))$
$H = H$ XOR $(U \gg (16 - k))$
$D_{i+j} = D_{i+j}$ XOR L
$D_{i+n} = D_{i+n}$ XOR H

The pseudo code in Figure 8 performs the modified step 8 for any Λ.

$H = 0$
for $j = 0$ **to** $n - 1$
$U = B_j'^{(0)}$
$L = (U \ll k)$ XOR H
$H = (U \gg (w - k))$
for $\lambda = 1$ **to** $\Lambda - 1$
$U = B_j'^{(\lambda)}$
$L = L$ XOR $(U \ll (k + \lambda\mathcal{W}))$
$H = H$ XOR $(U \gg (w - k - \lambda\mathcal{W}))$
$D_{i+j} = D_{i+j}$ XOR L
$D_{i+n} = D_{i+n}$ XOR H

Figure 8. Pseudo Code to compute $d(x) = d(x) + \sum_{\lambda=0}^{\Lambda-1} b'^{(\lambda)}(x) x^{iw+k+\lambda\mathcal{W}}$.

The complexity of the pseudo code in Figure 8 is given in Table 2. Table 2 includes only the cost of the loads and stores required for accessing to $B_j'^{(\lambda)}$ and D_{i+j}. The cost of accessing U, L, and H is excluded since they can be stored in registers.

Table 2. Complexity of the Pseudo Code in Figure 8.

XOR	Shift	Load/store
$2n\Lambda + 1$	$2n\Lambda$	$n\Lambda + 2n + 2$

4.2. Advantage of new method

In the usual multiplication, the words of the running sum $d(x)$ need to be loaded and stored, every time a partial product is added to $d(x)$. Thus, accumulating Λ partial products requires loading and storing the words of $d(x)$ Λ times. However, if the partial product accumulation is interleaved as illustrated in Figure 8, accumulating Λ partial products requires not Λ times but one time memory access to $d(x)$.

The same kind of improvement for the comb method is difficult for the following reasons:

1. The second loop in the proposed method (Figure 6) generates $\lceil w/\mathcal{W} \rceil$ of the partial products $\varepsilon(x)b(x)$, where the word size w is a fixed number ($w = 16$ or $w = 32$ in practice). However, the second loop in the comb method (Figure 4) generates n partial products where the operand size n is a varying number. The fixed number of the partial products in the proposed method can easily be clustered into the groups of Λ elements. However, the varying number of partial products in the comb method can not be clustered into groups at compile time.

2. The partial products generated in the second loop of our method (Figure 6) are easily interleaved since they are added to the same words of $d(x)$ in Step 8. In this step, x^{iw} is fixed, and thus the partial products are added to $d(x)$ starting from its i^{th} word. However, i is the loop variable of the second loop in the comb method (Figure 4), and thus all the partial products generated in the second loop are added to $d(x)$ starting from a different word. That is, these partial products are misaligned. Thus, accumulating a group of them together is a tedious job.

5. Complexity comparisons

The Karatsuba-based multiplication methods have an asymptotic time complexity of $\mathcal{O}(n^{1.58})$, while the proposed method and the comb method have quadratic time complexities. Thus, Karatsuba-based multiplication methods perform better for large polynomials whose degrees are greater than a threshold. Of course, this threshold varies from platform to platform.

It is easy to compare the time complexities of the comb method and the proposed method. In this comparison, we omit the complexities of initializing the output $d(x)$ by zeros, constructing the lookup table, computing the polynomials $\varepsilon(x)$, and locating the partial products $\varepsilon(x)b(x)$ in the lookup table. This is because the costs of these operations are the same for both methods. Actually, both methods have the same lookup table size and generate the same number of $\varepsilon(x)$ from the bits of input $a(x)$. The main difference between these methods is just the order in which they scan the input bits.

The comb method (Figure 4) needs $(n^2 + 6n - 2)\lceil \frac{w}{\mathcal{W}} \rceil$ shifts and XORs, as well as $(3n^2 + 5n)\lceil \frac{w}{\mathcal{W}} \rceil$ loads and stores, excluding the omitted costs.

This complexity is calculated as follows:

- Accessing A_i in Step 5 takes $n\lceil\frac{w}{\mathcal{W}}\rceil$ loads.
- $D_{i+j} = D_{i+j} + B'_j$ in Step 7 takes $n^2\lceil\frac{w}{\mathcal{W}}\rceil$ XORs and $3n^2\lceil\frac{w}{\mathcal{W}}\rceil$ load/stores.
- $d(x) = x^{\mathcal{W}}d(x)$ in Step 8 takes $(2n-1)\lceil\frac{w}{\mathcal{W}}\rceil$ XORs, $(4n-1)\lceil\frac{w}{\mathcal{W}}\rceil$ shifts, and $4n\lceil\frac{w}{\mathcal{W}}\rceil$ load/stores.

Here, $d(x) = x^{\mathcal{W}}d(x)$ is performed by the pseudo code in Figure 1 for $f(x) = d(x)$, $k = \mathcal{W}$, and $\mathcal{N} = 2n$. Its complexity is obtained from Table 1.

Our method (the algorithm in Figure 6, with the modification shown in Figure 2) needs $(4n^2\Lambda+n)\lceil\frac{w}{\Lambda\mathcal{W}}\rceil$ shifts and XORs, as well as $(n^2\Lambda + 2n^2 + 2n)\lceil\frac{w}{\Lambda\mathcal{W}}\rceil + n$ loads and stores, excluding the omitted costs. This complexity is calculated as follows:

- Accessing A_i in Step 4 takes n loads.
- $d(x) = d(x) + \sum_{\lambda=0}^{\Lambda-1} b'^{(\lambda)}(x)x^{iw+k+\lambda\mathcal{W}}$ in the modified step 8 takes $(2n^2\Lambda + n)\lceil\frac{w}{\Lambda\mathcal{W}}\rceil$ XORs, $2n^2\Lambda\lceil\frac{w}{\Lambda\mathcal{W}}\rceil$ shifts, and $(n^2\Lambda + 2n^2 + 2n)\lceil\frac{w}{\Lambda\mathcal{W}}\rceil$ load/stores.

Here, $d(x) = d(x) + \sum_{\lambda=0}^{\Lambda-1} b'^{(\lambda)}(x)x^{iw+k+\lambda\mathcal{W}}$ is performed by the pseudo code in Figure 8, whose complexity is given from Table 2.

Table 3 gives the complexities of the comb method and the proposed method where $\lceil\frac{w}{\Lambda\mathcal{W}}\rceil = \frac{1}{\Lambda}\lceil\frac{w}{\mathcal{W}}\rceil$. As seen from the table, the comb method requires fewer number of shifts and XORs than the proposed method. On the other hand, the proposed method requires fewer load and stores especially for large Λ.

Table 3. The method complexities assuming $\lceil\frac{w}{\Lambda\mathcal{W}}\rceil = \frac{1}{\Lambda}\lceil\frac{w}{\mathcal{W}}\rceil$.

	XOR/Shift	Load/store
Comb	$(n^2 + 6n - 2)\lceil\frac{w}{\mathcal{W}}\rceil$	$(3n^2 + 5n)\lceil\frac{w}{\mathcal{W}}\rceil$
Proposed	$(4n^2 + \frac{n}{\Lambda})\lceil\frac{w}{\mathcal{W}}\rceil$	$(n^2 + \frac{2n^2+2n}{\Lambda})\lceil\frac{w}{\mathcal{W}}\rceil + n$

6. Simulation results and discussion

The comb method [6], the Karatsuba algorithm [7, 8, 5], and the Karatsuba-like formulae [9] are often used in embedded cryptosystems. We simulate the performances of these algorithms and the proposed algorithm for the ARM7TDMI processor, using the Keil μvision development environment (version 3.51).

Table 4 gives the timing results of the proposed method, comb method, and the Karatsuba algorithm for the polynomial degrees usually encountered in Elliptic Curve Cryptosystems. The proposed method performs better than these algorithms clearly. The speed improvements obtained by the proposed method are given in the table. Our simulations show that the optimum window size for the comb method is $\mathcal{W} = 4$ on the ARM7TDMI platform, while the optimum parameters for the proposed method are $\mathcal{W} = 4$ and $\Lambda = 8$. Table 4 also shows that the Karatsuba algorithm always outperforms the comb method. This result is also demonstrated by a previous work [11, Figure 1].

Table 4. Computation Times in Cycles for the ARM 7 platform.

n words	$32n$ bits	Proposed Method	Comb Method	Speed up	Karatsuba Algorithm	Speed up
6	192	3422	5982	42.8%	3451	0.8%
7	224	4359	7810	44.2%	4907	11.2%
8	256	5398	9360	42.3%	5628	4.1%
9	288	6551	11604	43.5%	8203	20.1%
10	320	7806	13442	41.9%	9327	16.3%
11	352	9175	16102	43.0%	10469	12.4%
12	384	10646	18229	41.6%	11025	3.4%
13	416	12231	21305	42.6%	14000	12.6%
14	448	13918	23719	41.3%	15488	10.1%
15	480	15719	27211	42.2%	16994	7.5%
16	512	17622	29913	41.1%	17731	0.6%

Table 5 compares the timing results of the proposed method and an implementation using the Karatsuba-like formulae given in [9]. As seen, their performances are very similar. On the other hand, the proposed method requires several times less memory. The Karatsuba-based implementation, whose timings are given in Table 5, is a combination of Karatsuba-like formulae and the Karatsuba algorithm. This implementation uses the recursive Karatsuba algorithm to multiply the large inputs. The Karatsuba algorithm recursively defines a large product in terms of some smaller products. Eventually, the products with small inputs are multiplied by the nonrecursive methods. Our implementation uses the fast nonrecursive multiplication methods given in [9], each of which is designed for a specific input size (3, 4, 5, 6, and 7 words). Also, the nonrecursive methods multiplying the one and two word polynomials are included in the implementation. Though, these special multiplication methods improve the performance, they increase the code size significantly. As a result, the overall code size of this implementation is more than 6 kbytes and larger than the code sizes of the other implementations. On the other hand, the code size of the proposed method is less than 2 kbytes.

Table 5. Computation Times in Cycles for the ARM 7 platform.

n words	$32n$ bits	Proposed Method	Karatsuba-like formulae	Speed up
6	192	3422	3295	-3.9%
7	224	4359	4224	-3.2%
8	256	5398	5173	-4.3%
9	288	6551	7138	8.2%
10	320	7806	7956	1.9%
11	352	9175	9714	5.6%
12	384	10646	10586	-0.6%
13	416	12231	12502	2.2%
14	448	13918	13469	-3.3%
15	480	15719	15425	-1.9%
16	512	17622	16396	-7.5%

As seen in Tables 4 and 5, speed up values with respect to Karatsuba-based multiplication methods vary significantly. This is because the performance of the Karatsuba algorithm is sensitive to input size. Karatsuba-based methods work better when the input size is a power of two. Karatsuba algorithm recursively defines

smaller products from the lower and higher halves of their inputs and finds its output from these smaller products. If the input size is not an even number, the inputs are divided into two approximate halves, and this degrades the performance. The performance varies depending on how the inputs are divided into smaller inputs and which Karatsuba-like formulae are used to eventually multiply the small inputs.

In cryptographic applications, polynomial multiplication is usually used to perform field multiplication. Thus, multiplication results are reduced modulo to an irreducible polynomial. For efficiency, a sparse irreducible polynomial such as a trinomial or pentanomial is chosen to generate the field. We have implemented the polynomial reduction for pentanomials. The multiplications in Table 4 and 5 have 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, and 32 word results (two times the input size). We have found that the time required to reduce these multiplication results are 307, 348, 389, 430, 472, 510, 551, 592, 633, 674, and 717 cycles, respectively. Our measurements show that polynomial reduction accounts for 4% to 8% of the total computation time required for the field multiplication when the polynomial multiplication is performed by the proposed method.

7. Conclusion

We propose an efficient software method to implement the binary polynomial multiplication on embedded processors. The proposed method is similar to the comb method in many ways. However, our method aims at decreasing the number of load and store operations during partial product accumulation. Thus, our method is very advantageous for embedded processors with limited memory bandwidth. Our simulations on the ARM7TDMI platform show that the proposed method performs very well for the polynomial degrees usually encountered in the practical cryptosystems such as elliptic curve cryptography. The proposed method outperforms the comb method and the Karatsuba algorithm in this range while it shows a similar performance the Karatsuba-based multiplication method given in [9]. However, this method uses several algorithms customized for particular input lengths to avoid the iterations and recursions. Hence, it needs much larger code size than the proposed method.

The architecture of the embedded processor has an important role in the performances of the algorithms. The load and store instructions take more cycles than the arithmetic and logic instructions in low-end Arm processors such as Arm7. Also, low-end Arm processors have no cache. As a result, the proposed method, which saves on the load and store operations, shows a good performance on Arm7. Also, shift operations are free in Arm processors since they can be combined with other arithmetic and logic instructions. As a result, it is not advantageous to use the comb method, which saves on the shift operations, in Arm processors. Finally, high-end Arm processors have both instruction and data cache. Also, they can execute load-store operations in a single cycle. For this kind of devices, the Karatsuba-based multiplication methods are very affordable and preferable. The Karatsuba-based multiplication methods are asymptotically faster than the comb method and the proposed method, which are quadratic in time.

References

[1] J. von zur Gathen and J. Gerhard, "Arithmetic and factorization of polynomial over F_2" (extended abstract), In *ISSAC*, pages 1–9, 1996.

[2] J. von zur Gathen and J. Gerhard, "Polynomial factorization over F_2", *Mathematics of Computation*, 71(240):1677–1698, 2002.

[3] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, *Discrete Mathematics and its Applications*, Chapman & Hall/CRC, 2005.

[4] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag New York, Inc., 2004.

[5] A. Weimerskirch, D. Stebila, and S. C. Shantz, "Generic GF(2) arithmetic in software and its application to ECC", In *ACISP*, pages 79–92, 2003.

[6] J. López and R. Dahab, "High-speed software multiplication in F_{2^m}", In *INDOCRYPT*, pages 203–212, 2000.

[7] A. A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata", *Soviet Physics-Doklady (English translation)*, 7(7):595–596, 1963.

[8] D. E. Knuth, *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*, Addison-Wesley, Reading, MA, 3rd edition, 1997.

[9] P. L. Montgomery, "Five, six, and seven-term Karatsuba-like formulae", *IEEE Transactions on Computers*, 54(3):362–369, 2005.

[10] R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann, "Faster multiplication in GF(2)[x]", In *ANTS*, pages 153–166, 2008.

[11] S. Bartolini, I. Branovic, R. Giorgi, and E. Martinelli, "Effects of instruction-set extensions on an embedded processor: A case study on elliptic-curve cryptography over GF(2^m)", *IEEE Transactions on Computers*, 57(5):672–685, 2008.