Designs, Codes and Cryptography, 14(1), 57–69 (April 1998)
© April 1998 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Montgomery Multiplication in GF(2^k) * **

ÇETIN K. KOÇ AND TOLGA ACAR {koc,acar}@ece.orst.edu
Electrical & Computer Engineering
Oregon State University, Corvallis, Oregon 97331

Editor:

Abstract. We show that the multiplication operation $c = a \cdot b \cdot r^{-1}$ in the field GF(2^k) can be implemented significantly faster in software than the standard multiplication, where r is a special fixed element of the field. This operation is the finite field analogue of the Montgomery multiplication for modular multiplication of integers. We give the bit-level and word-level algorithms for computing the product, perform a thorough performance analysis, and compare the algorithm to the standard multiplication algorithm in GF(2^k). The Montgomery multiplication can be used to obtain fast software implementations of the discrete exponentiation operation, and is particularly suitable for cryptographic applications where k is large.

Keywords: Finite fields, multiplication, cryptography.

1. Introduction

The arithmetic operations in the Galois field GF(2^k) have several applications in coding theory, computer algebra, and cryptography. We are especially interested in cryptographic applications where k is very large. Examples of the cryptographic applications are the Diffie-Hellman key exchange algorithm (Diffie and Hellman 1976) based on the discrete exponentiation and elliptic curve cryptosystems (Koblitz 1994) over the field GF(2^k). The Diffie-Hellman algorithm requires implementation of the exponentiation g^e, where g is a fixed primitive element of the field and e an integer. The exponentiation operation can be implemented using a series of squaring and multiplication operations in GF(2^k) using the binary method (Knuth 1981).

Cryptographic applications require fast hardware and software implementations of the arithmetic operations in GF(2^k) for large values of k. An important advance in this field has been the Massey-Omura algorithm (Omura and Massey 1986) which is based on the normal bases. Subsequently, the optimal normal bases were introduced (Mullin, Onyszchuk, Vanstone and Wilson 1988), and their hardware (Agnew, Mullin and Vanstone 1993) and software (Schroeppel, O'Malley, Orman and Spatscheck 1995) implementations were given. While the hardware implementations are compact and fast, they are also inflexible and expensive. The change of the field in a hardware implementation requires a complete redesign. Software implementations, on the other hand, are perhaps slower, but they are cost-effective and flexible, i.e., the algorithms and the field parameters can easily be modified without requiring redesign. Recently, there has been a growing interest to develop software methods for implementing GF(2^k) arithmetic operations for cryptographic applications (Schroeppel, O'Malley, Orman and Spatscheck 1995; De Win, Bosselaers, Vandenberghe, De Gersem and Vandewalle 1996).

* This research is supported in part by Intel Corporation.
** This work was presented at the *Third Annual Workshop on Selected Areas in Cryptography*, Queen's University, Kingston, Ontario, Canada, August 15-16, 1996.

In this paper, we present an algorithm for multiplication in GF(2^k), which is significantly faster than the standard multiplication, and is particularly useful for obtaining fast software implementation of the discrete exponentiation operation. The algorithm is based on Montgomery's method for computing the modular multiplication operation. We use the polynomial representation of the field GF(2^k), and show that Montgomery's technique is also applicable here. We have performed a thorough analysis of the Montgomery multiplication algorithm, and compared it to the standard multiplication algorithm in GF(2^k). We show that this operation would be significantly faster in software with the availability of a fast method for multiplying two w-bit polynomials defined over GF(2), where w is the wordsize. For example, the Montgomery multiplication is about 5 times faster for $w = 8$, and about 20 times faster for $w = 32$.

2. Polynomial Representation

The elements of the field GF(2^k) can be represented in several different ways (McEliece 1987; Menezes 1993; Lidl and Niederreiter 1994). We find the polynomial representation useful and suitable for software implementation. The algorithm for the Montgomery multiplication described in this paper is based on the polynomial representation. According to this representation an element a of GF(2^k) is a polynomial of length k, i.e., of degree less than or equal to $k-1$, written as

$$a(x) = \sum_{i=0}^{k-1} a_i x^i = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0 \ ,$$

where the coefficients $a_i \in$ GF(2). These coefficients are also referred as the bits of a, and the element a is represented as $a = (a_{k-1}a_{k-2}\cdots a_1 a_0)$. In the word-level description of the algorithms, we partition these bits into blocks of equal length. Let w be the wordsize of the computer, also assume that $k = sw$. We can write a as an sw-bit number consisting of s blocks, where each block is of length w. Thus, we have $a = (A_{s-1}A_{s-2}\cdots A_1 A_0)$, where each A_i is of length w such that

$$A_i = (a_{iw+w-1}a_{iw+w-2}\cdots a_{iw+1}a_{iw}) \ .$$

In the polynomial case, this is equivalent to

$$\begin{aligned} a(x) &= \sum_{i=0}^{s-1} A_i(x) x^{iw} \\ &= A_{s-1}(x)x^{(s-1)w} + A_{s-2}(x)x^{(s-2)w} + \cdots + A_1(x)x^w + A_0(x) \ , \end{aligned}$$

where $A_i(x)$ is a polynomial of length w such that

$$A_i(x) = \sum_{j=0}^{w-1} a_{iw+j}x^j = a_{iw+w-1}x^{w-1} + a_{iw+w-2}x^{w-2} + \cdots + a_{iw+1}x + a_{iw} \ .$$

The addition of two elements a and b in GF(2^k) is performed by adding the polynomials $a(x)$ and $b(x)$, where the coefficients are added in the field GF(2). This is equivalent to bit-wise

XOR operation on the vectors a and b. In order to multiply two elements a and b in GF(2^k), we need an irreducible polynomial of degree k. Let $n(x)$ be an irreducible polynomial of degree k over the field GF(2). The product $c = a \cdot b$ in GF(2^k) is obtained by computing

$$c(x) = a(x)b(x) \bmod n(x) \ ,$$

where $c(x)$ is a polynomial of length k, representing the element $c \in$ GF(2^k). Thus, the multiplication operation in the field GF(2^k) is accomplished by multiplying the corresponding polynomials modulo the irreducible polynomial $n(x)$.

3. Montgomery Multiplication in GF(2^k)

Instead of computing $a \cdot b$ in GF(2^k), we propose to compute $a \cdot b \cdot r^{-1}$ in GF(2^k), where r is a special fixed element of GF(2^k). A similar idea was proposed by Montgomery in (Montgomery 1985) for modular multiplication of integers. We show that Montgomery's technique is applicable to the field GF(2^k) as well. The selection of $r(x) = x^k$ turns out to be very useful in obtaining fast software implementations. Thus, r is the element of the field, represented by the polynomial $r(x) \bmod n(x)$, i.e., if $n = (n_k n_{k-1} \cdots n_1 n_0)$, then $r = (n_{k-1} \cdots n_1 n_0)$. The Montgomery multiplication method requires that $r(x)$ and $n(x)$ are relatively prime, i.e., $\gcd(r(x), n(x)) = 1$. For this assumption to hold, it suffices that $n(x)$ be not divisible by x. Since $n(x)$ is an irreducible polynomial over the field GF(2), this will always be case. Since $r(x)$ and $n(x)$ are relatively prime, there exist two polynomials $r^{-1}(x)$ and $n'(x)$ with the property that

$$r(x)r^{-1}(x) + n(x)n'(x) = 1 \ , \tag{1}$$

where $r^{-1}(x)$ is the inverse of $r(x)$ modulo $n(x)$. The polynomials $r^{-1}(x)$ and $n'(x)$ can be computed using the extended Euclidean algorithm (Lidl and Niederreiter 1994; McEliece 1987). The Montgomery multiplication of a and b is defined as the product

$$c(x) = a(x)b(x)r^{-1}(x) \bmod n(x) \ , \tag{2}$$

which can be computed using the following algorithm:

```
Algorithm for Montgomery Multiplication
Input:    $a(x), b(x), r(x), n'(x)$
Output:   $c(x) = a(x)b(x)r^{-1}(x) \bmod n(x)$

Step 1.   $t(x) := a(x)b(x)$
Step 2.   $u(x) := t(x)n'(x) \bmod r(x)$
Step 3.   $c(x) := [t(x) + u(x)n(x)]/r(x)$
```

In order to prove the correctness of the above algorithm, we note that $u(x) = t(x)n'(x) \bmod r(x)$ implies that there is a polynomial $K(x)$ over GF(2) with the property

$$u(x) = t(x)n'(x) + K(x)r(x) \ . \tag{3}$$

We write the expression for $c(x)$ in Step 3, and then substitute $u(x)$ with the expression (3) as

$$\begin{aligned} c(x) &= \frac{1}{r(x)}\,[t(x) + u(x)n(x)] \\ &= \frac{1}{r(x)}\,[t(x) + t(x)n'(x)n(x) + K(x)r(x)n(x)] \end{aligned}$$

Furthermore, we have $n'(x)n(x) = 1 + r(x)r^{-1}(x)$ according to (1). Thus, $c(x)$ is obtained as

$$\begin{aligned} c(x) &= \frac{1}{r(x)}\,[t(x) + t(x)[1 + r(x)r^{-1}(x)] + K(x)r(x)n(x)] \\ &= \frac{1}{r(x)}\,[t(x)r(x)r^{-1}(x) + K(x)r(x)n(x)] \\ &= t(x)r^{-1}(x) + K(x)n(x) \\ &= a(x)b(x)r^{-1}(x) \bmod n(x)\ , \end{aligned}$$

as required. The above algorithm is similar to the algorithm given for the Montgomery multiplication of integers. The only difference is that the final subtraction step required in the integer case is not necessary in the polynomial case. This is proved by showing that the degree of the polynomial $c(x)$ computed by this algorithm is less than or equal to $k-1$. Since the degrees of $a(x)$ and $b(x)$ are both less than or equal to $k-1$, the degree of $t(x) = a(x)b(x)$ will be less than or equal to $2(k-1)$. Also note that the degrees of $n(x)$ and $r(x)$ are both equal to k. The degree of $u(x)$ computed in Step 2 will be strictly less than k since the operation is performed modulo $r(x)$. Thus, the degree of $c(x)$ as computed in Step 3 of the algorithm is found as

$$\begin{aligned} \deg\{c(x)\} &\leq \max[\deg\{t(x)\}\ ,\ \deg\{u(x)\} + \deg\{n(x)\}] - \deg\{r(x)\} \\ &\leq \max[2k-2\ ,\ k-1+k] - k \\ &\leq k-1 \end{aligned}$$

Thus, the polynomial $c(x)$ is already reduced.

4. Computation of Montgomery Multiplication

The computation of $c(x)$ involves a regular multiplication in Step 1, a modulo $r(x)$ multiplication in Step 2, and finally a regular multiplication and a division by $r(x)$ operation in Step 3. The modular multiplication and division operations in Steps 2 and 3 are intrinsically fast operations since $r(x) = x^k$. The remainder operation in modular multiplication using the modulus x^k is accomplished by simply ignoring the terms which have powers of x larger than or equal to k. Similarly, division of an arbitrary polynomial by x^k is accomplished by shifting the polynomial to the right by k places. The precomputation of $n'(x)$ required in Step 2 constitutes an overhead for computing $c(x)$. However, it turns out the computation of $n'(x)$ can be completely avoided if the coefficients of $a(x)$ are scanned one bit at a time. Furthermore, the word-level algorithm requires the computation of only the least significant word $N_0'(x)$ instead of the whole $n'(x)$.

Recall that we need to compute $c(x) = a(x)b(x)r^{-1}(x) \bmod n(x)$, where $r(x) = x^k$. This product can be written as

$$c(x) = x^{-k}a(x)b(x) = x^{-k} \sum_{i=0}^{k-1} a_i x^i b(x) \bmod n(x) \ .$$

The product

$$t(x) = (a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0)b(x)$$

can be computed by starting from the most significant digit, and then proceeding to the least significant, as follows:

```
$t(x) := 0$
for $i = k-1$ to 0
    $t(x) := t(x) + a_i b(x)$
    $t(x) := xt(x)$
```

The shift factor x^{-k} in $x^{-k}a(x)b(x)$ reverses the direction of summation. Since

$$\begin{aligned} x^{-k}(a_{k-1}x^{k-1} &+ a_{k-2}x^{k-2} + \cdots + a_1 x + a_0) = \\ & a_{k-1}x^{-1} + a_{k-2}x^{-2} + \cdots + a_1 x^{-k+1} + a_0 x^{-k} \ , \end{aligned}$$

we start processing the coefficients of $a(x)$ from the least significant, and obtain the following bit-level algorithm in order to compute $t(x) = a(x)b(x)x^{-k}$.

```
$t(x) := 0$
for $i = 0$ to $k-1$
    $t(x) := t(x) + a_i b(x)$
    $t(x) := t(x)/x$
```

This algorithm computes the product $t(x) = x^{-k}a(x)b(x)$, however, we are interested in computing $c(x) = x^{-k}a(x)b(x) \bmod n(x)$. Following the analogy to the integer algorithm, we achieve this computation by adding $n(x)$ to $c(x)$ if c_0 is 1, making the new $c(x)$ divisible by x since $n_0 = 1$. If c_0 is already 0 after the addition step, we do not add $n(x)$ to it. Therefore, we are computing $c(x) := c(x) + c_0 n(x)$ after the addition step. After this computation, $c(x)$ will always be divisible by x. We can compute $c(x) := c(x)x^{-1} \bmod n(x)$ by dividing $c(x)$ by x since $c(x) = xu(x)$ implies $c(x)x^{-1} = xu(x)x^{-1} = u(x) \bmod n(x)$. The bit-level algorithm is given below:

```
Bit-Level Algorithm for Montgomery Multiplication
Input:    $a(x), b(x), n(x)$
Output:   $c(x) = a(x)b(x)x^{-k} \bmod n(x)$
Step 1.   $c(x) := 0$
Step 2.   for $i = 0$ to $k-1$ do
Step 3.       $c(x) := c(x) + a_i b(x)$
Step 4.       $c(x) := c(x) + c_0 n(x)$
Step 5.       $c(x) := c(x)/x$
```

The bit-level algorithm for the Montgomery multiplication given above is generalized to the word-level algorithm by proceeding word by word, where the wordsize is $w \geq 2$ and $k = sw$. Let $A_i(x)$ represent one word of the polynomial $a(x)$. The addition step is performed by multiplying $A_i(x)$ by $b(x)$ at the ith step. We then need to multiply the partial product $c(x)$ by x^{-w} modulo $n(x)$. In order to perform this step using division, we add a multiple of $n(x)$ to $c(x)$ so that the least significant w coefficients of $c(x)$ will be zero, i.e., $c(x)$ will be divisible by x^w. Thus, if $c(x) \neq 0 \bmod x^w$, then we find $M(x)$ (which is a polynomial of degree $\leq w-1$) such that $c(x)+M(x)n(x) = 0 \pmod{x^w}$. Let $C_0(x)$ and $N_0(x)$ be the least significant words of $c(x)$ and $n(x)$, respectively. We calculate $M(x)$ as

$$M(x) = C_0(x)N_0^{-1}(x) \bmod x^w \ .$$

We note that $N_0^{-1}(x) \bmod x^w$ is equal to $N_0'(x)$ since the property (1) implies that

$$\begin{aligned} x^{sw}x^{-sw} + n(x)n'(x) &= 1 \pmod{x^w} \\ N_0(x)N_0'(x) &= 1 \pmod{x^w} \end{aligned}$$

The word-level algorithm for the Montgomery multiplication is obtained as

```
Word-Level Algorithm for Montgomery Multiplication
Input:    $a(x), b(x), n(x), N_0'(x)$
Output:   $c(x) = a(x)b(x)x^{-k} \bmod n(x)$

Step 1.   $c(x) := 0$
Step 2.   for $i = 0$ to $s-1$ do
Step 3.       $c(x) := c(x) + A_i(x)b(x)$
Step 4.       $M(x) := C_0(x)N_0'(x) \pmod{x^w}$
Step 5.       $c(x) := c(x) + M(x)n(x)$
Step 6.       $c(x) := c(x)/x^w$
```

The word-level algorithm requires the computation of the w-length polynomial $N_0'(x)$ instead of the entire polynomial $n'(x)$ which is of length $k = sw$. It turns out that the short algorithm developed for computing n_0' in the integer case (Dussé and Kaliski 1990) can also be generalized to the polynomial case. The inversion algorithm is based on the observation that the polynomial $N_0(x)$ and its inverse satisfy

$$N_0(x)N_0^{-1}(x) = 1 \pmod{x^i}$$

for $i = 1, 2, \ldots, w$. In order to compute $N_0'(x)$, we start with $N_0'(x) = 1$, and proceed as

```
Inversion Algorithm
Input:    $w, N_0(x)$
Output:   $N_0'(x) = N_0^{-1} \bmod x^w$

Step 1.   $N_0'(x) := 1$
Step 2.   for $i = 2$ to $w$
Step 3.       $t(x) := N_0(x)N_0'(x) \bmod x^i$
Step 4.       if $t(x) \neq 1$ then $N_0'(x) := N_0'(x) + x^{i-1}$
```

5. Computation of Montgomery Squaring

The computation of the Montgomery multiplication for $a(x) = b(x)$ can optimized due to the fact that cross terms disappear because they come in pairs and the underlying field is GF(2). It is easy to show that

$$a^2(x) = \sum_{i=0}^{k-1} a_i x^{2i} \ ,$$

and thus, the multiplication steps in the bit-level and word-level algorithms can be skipped. The Montgomery squaring algorithm starts with the degree $2(k-1)$ polynomial $c(x) = a^2(x)$, i.e.,

$$\begin{aligned} c(x) &= a_{k-1}x^{2(k-1)} + a_{k-2}x^{2(k-2)} + \cdots + a_1x^2 + a_0 \\ &= (a_{k-1}\mathbf{0}a_{k-2}\mathbf{0}\cdots\mathbf{0}a_1\mathbf{0}a_0) \ , \end{aligned}$$

and then reduces $c(x)$ by computing $c(x) := c(x)x^{-k} \bmod n(x)$. The steps of the bit-level algorithm are illustrated below:

```
Bit-Level Algorithm for Montgomery Squaring
Input:    $a(x), n(x)$
Output:   $c(x) = a^2(x)x^{-k} \bmod n(x)$
Step 1.   $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$
Step 2.   for $i = 0$ to $k-1$ do
Step 3.       $c(x) := c(x) + c_0 n(x)$
Step 4.       $c(x) := c(x)/x$
```

Similarly, the word-level algorithm starts with the same polynomial $c(x)$, however, then performs the reduction steps by proceeding word by word, as follows:

```
Word-Level Algorithm for Montgomery Squaring
Input:    $a(x), n(x), N_0'(x)$
Output:   $c(x) = a^2(x)x^{-k} \bmod n(x)$
Step 1.   $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$
Step 2.   for $i = 0$ to $s-1$ do
Step 3.       $M(x) := C_0(x)N_0'(x) \pmod{x^w}$
Step 4.       $c(x) := c(x) + M(x)n(x)$
Step 5.       $c(x) := c(x)/x^w$
```

Example: We take the field GF(2^4) to illustrate the Montgomery product computation. The irreducible polynomial is selected to be $n(x) = x^4 + x + 1$. Furthermore, we have $k = 4$ and $r(x) = x^4$. Since $n = (10011)$, the special field element r is (0011). The inverse of $r(x)$ modulo $n(x)$ is computed as $r^{-1}(x) = x^3 + x^2 + x = (1110)$ using the extended Euclidean algorithm. This result is easily verified by computing

$$\begin{aligned} r(x)r^{-1}(x) &= x^4(x^3+x^2+x) && \pmod{x^4+x+1} \\ &= x^7+x^6+x^5 && \pmod{x^4+x+1} \\ &= 1 && \pmod{x^4+x+1} \end{aligned}$$

Furthermore, we compute $n'(x)$ using the property (1) as

$$\begin{aligned} n'(x) &= \frac{1 + r(x)r^{-1}(x)}{n(x)} = \frac{1 + x^4(x^3+x^2+x)}{x^4+x+1} = \frac{x^7+x^6+x^5+1}{x^4+x+1} \\ &= x^3 + x^2 + x + 1 \ . \end{aligned}$$

Let $a(x) = x^3 + x^2 + 1 = (1101)$ and $b(x) = x^3 + 1 = (1001)$. In order to compute the product $c = a \cdot b \cdot r^{-1}$ in GF(2^4), we use the algorithm for the Montgomery multiplication, and compute $t(x)$, $u(x)$, and $c(x)$ as follows:

$$\begin{array}{rrcl}
\text{Step 1:} & t(x) & = & a(x)b(x) = (x^3+x^2+1)(x^3+1) \\
 & & = & x^6+x^5+x^2+1 \ . \\
\text{Step 2:} & u(x) & = & t(x)n'(x) = (x^6+x^5+x^2+1)(x^3+x^2+x+1) \\
 & & = & x^9+x^4+x+1 = x+1 \pmod{x^4} \ . \\
\text{Step 3:} & c(x) & = & [t(x)+u(x)n(x)]/r(x) \\
 & & = & [(x^6+x^5+x^2+1)+(x+1)(x^4+x+1)]/x^4 \\
 & & = & (x^6+x^4)/x^4 = x^2+1 \ .
\end{array}$$

Thus, we conclude that $a \cdot b \cdot r^{-1}$ is equal to $c = (0101)$. This result is obtained using the bit-level algorithm without computing $n'(x)$ or $r^{-1}(x)$. The bit-level algorithm starts with $c(x) = 0$, and obtains $c(x) = x^2 + 1$ using the steps given on Table 1.

Table 1. The bit-level computation of $a(x)b(x)x^{-4}$ mod $n(x)$.

i	a_i	$a_ib(x)$	Step 3 $c(x) := c(x) + a_ib(x)$	c_0	Step 4 $c(x) := c(x) + c_0n(x)$	Step 5 $c(x) := c(x)/x$
0	1	x^3+1	x^3+1	1	x^4+x^3+x	x^3+x^2+1
1	0	0	x^3+x^2+1	1	$x^4+x^3+x^2+x$	x^3+x^2+x+1
2	1	x^3+1	x^2+x	0	x^2+x	$x+1$
3	1	x^3+1	x^3+x	0	x^3+x	x^2+1

We now illustrate the steps of the word-level algorithm to compute $c(x)$. The word-level algorithm first computes $N_0'(x)$ using the inversion algorithm. Let $w = 2$. Since $n(x) = x^4 + x + 1$, we have $N_0(x) = x + 1$. The inversion algorithm starts with $N_0'(x) = 1$, and then computes

$$t(x) = N_0(x)N_0'(x) = (x+1)(1) = x+1 \pmod{x^2} \ .$$

Since $t(x) \neq 1$, the value of $N_0'(x)$ is updated as $N_0'(x) = N_0'(x) + x = 1 + x$. Therefore, we obtain $N_0'(x) = x + 1$ using the inversion algorithm. This result is easily verified since

$$N_0(x)N_0'(x) = (x+1)(x+1) = x^2+1 = 1 \pmod{x^2} \ .$$

The word-level algorithm starts with $c(x) = 0$. Since $a(x) = (1101)$, we have $A_0(x) = (01) = 1$ and $A_1(x) = (11) = x + 1$. Furthermore, $N_0'(x) = (11) = x + 1$. The steps of the word-level algorithm for computing the result $c(x) = x^2 + 1$ are given below:

$$\begin{array}{lll}
i=0 & \text{Step 3:} & c(x) = c(x) + A_0(x)b(x) = (0) + (1)(x^3+1) = x^3+1 \\
& \text{Step 4:} & M(x) = C_0(x)N_0'(x) = (1)(x+1) = x+1 \pmod{x^2} \\
& \text{Step 5:} & c(x) = c(x) + M(x)n(x) = (x^3+1) + (x+1)(x^4+x+1) \\
& & = x^5+x^4+x^3+x^2 \\
& \text{Step 6:} & c(x) = c(x)/x^2 = (x^5+x^4+x^3+x^2)/x^2 = x^3+x^2+x+1 \\
\\
i=1 & \text{Step 3:} & c(x) = c(x) + A_1(x)b(x) = (x^3+x^2+x+1) + (x+1)(x^3+1) \\
& & = x^4+x^2 \\
& \text{Step 4:} & M(x) = C_0(x)N_0'(x) = (0)(x+1) = 0 \pmod{x^2} \\
& \text{Step 5:} & c(x) = c(x) + M(x)n(x) = (x^4+x^2) + (0)(x^4+x+1) \\
& & = x^4+x^2 \\
& \text{Step 6:} & c(x) = c(x)/x^2 = (x^4+x^2)/x^2 = x^2+1
\end{array}$$

Finally, we give an example illustrating the word-level Montgomery squaring algorithm. We compute $c = a \cdot a \cdot r^{-1}$ where $a = (1101) = x^3+x^2+1$. The word-level Montgomery algorithm starts with $c(x) = a^2(x) = x^6+x^4+1$, and performs the following steps in order to compute the final result.

$$\begin{array}{lll}
i=0 & \text{Step 3:} & M(x) = C_0(x)N_0'(x) = (1)(x+1) = x+1 \pmod{x^2} \\
& \text{Step 4:} & c(x) = c(x) + M(x)n(x) \\
& & = (x^6+x^4+1) + (x+1)(x^4+x+1) = x^6+x^5+x^2 \\
& \text{Step 5:} & c(x) = c(x)/x^2 = (x^6+x^5+x^2)/x^2 = x^4+x^3+1 \\
\\
i=1 & \text{Step 3:} & M(x) = C_0(x)N_0'(x) = (1)(x+1) = x+1 \pmod{x^2} \\
& \text{Step 4:} & c(x) = c(x) + M(x)n(x) \\
& & = (x^4+x^3+1) + (x+1)(x^4+x+1) = x^5+x^3+x^2 \\
& \text{Step 5:} & c(x) = c(x)/x^2 = (x^5+x^3+x^2)/x^2 = x^3+x+1
\end{array}$$

□

6. Analysis of the Word-Level Algorithm

In this section, we give a rigorous analysis of the word-level algorithm for computing the Montgomery product. We calculate the number of word-level GF(2) addition and multiplication operations. The word-level addition is simply the bitwise `XOR` operation which is a readily available instruction on most general purpose microprocessors and signal processors. The word-level multiplication operation receives two 1-word (w-bit) polynomials $A(x)$ and $B(x)$ defined over the field GF(2), and computes the 2-word polynomial $C(x) = A(x)B(x)$. The degree of the product polynomial $C(x)$ is $2(w-1)$. For example, given $A = (1101)$ and $B = (1010)$, this operation computes C as

$$\begin{aligned}
A(x)B(x) &= (x^3+x^2+1)(x^3+x) \\
&= x^6+x^5+x^4+x \\
&= (0111\ 0010)\ .
\end{aligned}$$

Unfortunately, none of the general purpose processors contains an instruction to perform the above operation. The implementation of this operation, which we call `MULGF2`, can be performed in two distinctly different ways:

- Emulation using `SHIFT` and `XOR` operations.
- Table lookup approach.

The emulation approach is usually slower than the table lookup approach, particularly for $w \geq 8$. The following function can be used to compute the 2-word result `H,L` given the inputs `A` and `B`. The `MULGF2` algorithm given below requires $2w$ `SHIFT` and w `XOR` operations.

```
H := 0 ; L := 0
for j=w-1 downto 0 do
    L := SHL(L,1)
    H := RCL(H,1)
    if BIT(B,j)=1 then L := L XOR A
```

Here, `SHL` shifts its first operand to the left by the number of bits given in the second operand. `RCL` is a rotate (circular shift) instruction shifting the first operand to the left circularly by the number of bits given in the second operand.

On the other hand, a simple method for implementing the table lookup approach is to use 2 tables, one for computing `H` and the other for computing `L`. The tables are addressed using the bits of `A` and `B`, and thus, each table is of size $2^w \times 2^w \times w$ bits. We obtain the values of `H` and `L` using two table reads. Other approaches are also possible. However, we note that these tables are different from the tables in (Harper, Menezes and Vanstone 1992; De Win, Bosselaers, Vandenberghe, De Gersem and Vandewalle 1996), which are used to implement GF(2^w) multiplications. Here we are using the tables to multiply two $(w-1)$-degree polynomials over the field GF(2) to obtain the product polynomial which is of degree $2(w-1)$. In Table 2, we give the the number of `MULGF2` and `XOR` operations required in each step of the word-level Montgomery multiplication algorithm.

Table 2. Operation counts for the word-level Montgomery multiplication algorithm.

	MULGF2	XOR
`for i=0 to s do`	-	-
`   C[i]:=0`	-	-
`for i=0 to s-1 do`	-	-
`   for j=0 to s-1 do`	-	-
`      MULGF2(H,L,A[j],B[i])`	s^2	-
`      C[j]:=C[j] XOR L`	-	s^2
`      C[j+1]:=C[j+1] XOR H`	-	s^2
`   MULGF2(H,M,C[0],N0')`	s	-
`   MULGF2(P,L,M,N[0])`	s	-
`   for j=1 to s-1 do`	-	-
`      MULGF2(H,L,M,N[j])`	$s^2 - s$	-
`      C[j-1]:=C[j] XOR L XOR P`	-	$2s^2 - 2s$
`      P:=H`	-	-
`   C[s-1]:=C[s] XOR P XOR M`	-	$2s$
`   C[s]:=0`	-	-
	$2s^2 + s$	$4s^2$

We now compare the word-level Montgomery multiplication algorithm to the standard GF(2^k) multiplication using the polynomial representation. The standard GF(2^k) multiplication can be accomplished in several different ways. We select the word-level interleaving and reduction method for comparison since this algorithm is very similar to the word-level algorithm for the Montgomery multiplication in terms of its data structure and the general flow. This algorithm computes $c = a \cdot b$ using the polynomial representation by computing $c(x) = a(x)b(x) \pmod{n(x)}$.

```
Word-Level Standard Multiplication Algorithm
Input:    $a(x), b(x), n(x)$
Output:   $c(x) = a(x)b(x) \bmod n(x)$

Step 1.   $c(x) := 0$
Step 2.   for $i = s - 1$ downto 0 do
Step 3.       $c(x) := c(x)x^w$
Step 4.       $c(x) := c(x) + B_i(x)a(x)$
Step 5.       $c(x) := c(x) \pmod{n(x)}$
```

In Step 5, the modular reduction is performed by aligning the most significant word of $n(x)$ with the most significant word of $c(x)$, and then by performing a series of bit-level right shift and polynomial additions until the most significant word of $c(x)$ becomes zero. Table 3 gives the number of MULGF2, XOR, and SHIFT operations required in each step of the word-level standard multiplication algorithm.

Table 3. Operation counts for the word-level standard multiplication algorithm.

	MULGF2	XOR	SHIFT
`for i=0 to s do`	-	-	-
`C[i]:=0`	-	-	-
`for i=s-1 downto 0 do`	-	-	-
`P:=0`	-	-	-
`for j=s-1 downto 0 do`	-	-	-
`MULGF2(H,L,A[j],B[i])`	s^2	-	-
`C[j+1]:=C[j] XOR H XOR P`	-	$2s^2$	-
`P:=L`	-	-	-
`C[0]:=P`	-	-	-
`for j=s downto 1 do`	-	-	-
`U[j]:=SHL(N[j],w-1) XOR SHR(N[j-1],1)`	-	s^2	$2s^2$
`U[0]:=SHL(N[0],w-1)`	-	-	s
`for j=w-1 downto 0 do`	-	-	-
`if DEGREE(C)>=DEGREE(U) then`	-	-	-
`for k=0 to s do`	-	-	-
`C[k]:=C[k] XOR U[k]`	-	$sw(s+1)/2$	-
`for k=0 to s-1 do`	-	-	-
`U[k]:=SHR(U[k],1) XOR SHL(U[k+1],w-1)`	-	s^2w	$2s^2w$
`U[s]:=SHR(U[s],1)`	-	-	sw
	s^2	$3s^2(w/2+1)+$ $sw/2$	$2s^2(w+1)+$ $s(w+1)$

As can be seen from Tables 1 and 2, the word-level Montgomery multiplication algorithm performs about twice as many MULGF2 operations as the standard algorithm, however, it requires much fewer XOR operations and no SHIFT operation. Thus, if the MULGF2 operation can be performed in a few clock cycles, the word-level Montgomery multiplication algorithm would be significantly faster. In Table 4, we tabulate the total number of operations for the Montgomery and standard multiplication algorithms for $w = 8, 16, 32$ for comparison purposes. We have also implemented these two algorithms in C, and obtained timings on a

Table 4. Comparing the Montgomery and standard multiplication.

	Emulation			Table Lookup		
w	Standard	Montgomery	Speedup	Standard	Montgomery	Speedup
8	$57s^2 + 13s$	$52s^2 + 24s$	1.09	$34s^2 + 13s$	$6s^2 + s$	5.67
16	$109s^2 + 25s$	$100s^2 + 48s$	1.09	$62s^2 + 25s$	$6s^2 + s$	10.33
32	$213s^2 + 49s$	$196s^2 + 96s$	1.09	$118s^2 + 49s$	$6s^2 + s$	19.67

100-MHz Intel 486DX4 processor running the NextStep 3.3 operating system. We summarize the experimental speedup values in Table 5.

As was mentioned, the table lookup approach can be implemented using 2 tables each of which is of size $2^w \times 2^w \times w$ bits. For $w = 8$, each of the tables is of size 64 Kilobytes, which is quite reasonable. However, for $w = 16$, the table size increases to $2^{16} \times 2^{16} \times 16$ bits, or 8 Gigabytes. Thus, we have implemented the table lookup `MULGF2` algorithm for only $w = 8$. For $w = 16$ and $w = 32$, we have also implemented the `MULGF2` operation using an hybrid approach: 8-bit tables coupled with emulation to obtain the 16-bit or 32-bit result.

As the theoretical data in Table 4 and the experimental data in Table 5 indicate, the Montgomery multiplication algorithm is about 2–5 times faster than the standard multiplication for $w = 8$. Table lookup approach for $w \geq 16$ seems unrealistic due to the size of the tables. An efficient way to implement the `MULGF2` operation is to add an instruction to the processor to perform this multiplication. The availability of a 16-bit or 32-bit `MULGF2` instruction would make the Montgomery multiplication about 10–20 times faster than the standard multiplication.

Table 5. Experimental speedup values.

w	Method $k \rightarrow$	64	128	256	512	1024	1536	2048
8	Table Lookup	4.51	3.82	3.17	2.94	2.83	2.43	2.45
16	Hybrid (8)	4.26	2.83	2.15	2.05	2.02	2.07	2.00
32	Hybrid (8)	4.49	2.99	2.24	2.08	2.27	1.59	1.53
8	Emulation	3.89	3.26	2.67	2.63	2.41	2.37	2.25
16	Emulation	4.70	2.92	2.11	1.99	2.09	1.93	2.01
32	Emulation	4.20	2.50	1.48	1.36	1.46	1.54	1.53

7. Conclusions

We have described the bit-level and word-level algorithms for computing the Montgomery product $a \cdot b \cdot r^{-1}$ in the field GF(2^k). It turns out that this operation would be significantly faster in software with the availability of a fast method for multiplying two w-bit polynomials defined over GF(2), where w is the wordsize. This can be achieved using a table lookup approach when the wordsize is small; another method is to implement an instruction on the underlying processor for performing this operation which is much simpler than the integer multiplication due to the lack of carry propagation.

The Montgomery multiplication can be used to obtain fast software implementation of the exponentiation over the field GF(2^k). Let the field element a and the m-bit positive integer e be given. In order to compute $c = a^e \in$ GF(2^k), we can use the binary method (Knuth 1981). The algorithm first computes $\bar{c} = 1 \cdot r$ and $\bar{a} = a \cdot r$ using the standard multiplication, and then proceeds to compute c using only Montgomery squarings and multiplications.

```
for $i = m - 1$ downto 0 do
    $\bar{c} := \bar{c} \cdot \bar{c} \cdot r^{-1}$
    if $e_i = 1$ then $\bar{c} := \bar{c} \cdot \bar{a} \cdot r^{-1}$
$c := \bar{c} \cdot 1 \cdot r^{-1}$
```

Our findings regarding the efficiency of the Montgomery exponentiation algorithm are summarized in (Koç and Acar 1997). We are currently working on extending the Montgomery multiplication and squaring to the normal bases.

References

G. B. Agnew, R. C. Mullin, I. Onyszchuk, and S. A. Vanstone. An implementation for a fast public-key cryptosystem. *Journal of Cryptology*, 3(2):63–79, 1991.

G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology – EUROCRYPT 90, Lecture Notes in Computer Science, No. 473*, pages 230–244, New York, NY, 1990. Springer-Verlag.

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, New York, NY, 1992. Springer-Verlag.

D. .E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, MA, Second edition, 1981.

N. Koblitz. *A Course in Number Theory and Cryptography.* New York, NY:Springer-Verlag, New York, NY, Second edition, 1994.

Ç. K. Koç and T. Acar. Fast software exponentiation in GF(2^k). In *Proceedings, 9th Symposium on Computer Arithmetic*, pages 225–231, Asilomar, California, July 6–9, 1997.

R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications.* Cambridge University Press, Cambridge, UK, 1994.

R. J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 1987.

A. J. Menezes, editor. *Applications of Finite Fields.* Kluwer Academic Publishers, Boston, MA, 1993.

A. J. Menezes. *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic Publishers, Boston, MA, 1993.

P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

R. Mullin, I. Onyszchuk, S. Vanstone, and R. Wilson. Optimal normal bases in GF(p^n). *Discrete Applied Mathematics*, 22:149–161, 1988.

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

R. Schroeppel, S. O'Malley, H. Orman, 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, New York, NY, 1995. Springer-Verlag.

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 *Advances in Cryptology — ASIACRYPT 96*, Lecture Notes in Computer Science, No. 1163, pages 65–76. New York, NY: Springer-Verlag, 1996.