Normal Basis Multiplication Algorithms for $GF(2^n)$ (Full Version)

Haining Fan, Duo Liu and Yiqi Dai
Department of Computer Science, Tsinghua University, Beijing 100084, People's Republic of China.
fan_haining@yahoo.com

Abstract - In this paper, we propose a new normal basis multiplication algorithm for $GF(2^n)$. This algorithm can be used to design not only fast software algorithms but also low complexity bit-parallel multipliers in some $GF(2^n)$s. Especially, for some values of n that no Gaussian normal basis exists in $GF(2^n)$, i.e., $8|n$, this algorithm provides an alternative way to construct low complexity normal basis multipliers. Two improvements on a recently proposed software normal basis multiplication algorithm are also presented. Time and memory complexities of these normal basis multiplication algorithms are compared with respect to software performance. It is shown that they have some specific behavior in different applications. For example, $GF(2^{571})$ is one of the five binary fields recommended by NIST for ECDSA (Elliptic Curve Digital Signature Algorithm) applications. In this field, our experiments show that the new algorithm is even faster than the polynomial basis Montgomery multiplication algorithm: 525 μs v. 819 μs.

Index Terms - Finite field, normal basis, Massey-Omura multiplier, optimal normal basis.

1. Introduction

$GF(2^n)$ arithmetic plays an important role in the implementation of cryptosystems. Among different types of field representations, the normal basis (NB) has received considerable attention because squaring in NB is simply a cyclic shift of the coordinates of the element and, thus, it has found applications in computing multiplicative inverses and exponentiations.

In software implementations, an algorithm for type-I ONB was shown in [11]. It can be further improved if the symmetric property proposed in [21] or [4] is employed. In [4] and [5], Reyhani-Masoleh and Hasan proposed a word-level NB multiplication algorithm over $GF(2^n)$, and we

---

This report compares three normal basis multiplication algorithms published in the following. It includes all experimental results omitted in [$].

[*] H. Fan and Y. Dai, "Two software normal basis multiplication algorithms for $GF(2^n)$, "
Cryptology ePrint Archive, Report 2004/126, 2004. http://eprint.iacr.org/. This paper is now published in:
H. Fan, D. Liu and Y. Dai, "Two Software Normal Basis Multiplication Algorithms for $GF(2^n)$, "
Tsinghua Science and Technology, vol. 11, no.3, pp. 264-270, 2006.

[$] H. Fan and Y. Dai, "Normal basis multiplication algorithm for $GF(2^n)$,"
IEE Electronics Letters, vol.40, no.18, pp. 1112-1113, 2004.

denote it as RH algorithm. RH algorithm is designed for working with all NB and its time complexity depends not only on n but also C_N (the complexity of selected NB [7]). The algorithm developed in [23] is efficient for composite finite fields. Since XOR and AND instructions take the same number of clock cycles on most modern CPUs, the time complexity of this algorithm is the same as RH algorithm in $GF(2^n)$ where n is prime. All these previously mentioned algorithms are word-level and more efficient than the bit-level NB multiplication algorithm presented in the IEEE standard P1363-2000 [12]. In [13], Ning and Yin presented an improved algorithm of [12], and we denote it as NY algorithm. Although NY algorithm is fast for ONB, it is slow for nonoptimal NB.

In this paper, a new NB multiplication algorithm is proposed for $GF(2^n)$. The field element is represented with a normal element of small multiplicative order. The complexity of the proposed algorithm does not depend on C_N, but the multiplicative order of the normal element. For some $GF(2^n)$s, this new algorithm can be used to design both fast software multiplication algorithms and low complexity bit-parallel multipliers. Especially, for some values of n that no Gaussian NB (GNB) exists in $GF(2^n)$, i.e., $8|n$, this algorithm is still applicable. For software implementations, we compare the proposed NB algorithm with the polynomial basis multiplication algorithm of [6], i.e., the finite field analogue of the Montgomery multiplication for integers. Our experimental results show that in some $GF(2^n)$s the new algorithm is faster than the Montgomery algorithm, e.g., $GF(2^{571})$, one of the five binary fields recommended by NIST for ECDSA (Elliptic Curve Digital Signature Algorithm) applications.

Two improvements on the RH algorithm are also presented. Time and memory complexities of these NB multiplication algorithms are compared with respect to their software performance. It is shown that they have specific behaviors in different applications.

This paper is organized as follows: In Section 2, we review the RH algorithm and its two improvements. Then, the proposed algorithm and software implementation are presented in Section 3. Some mathematical properties of the new algorithm are derived in Section 4. In Section 5, we show the new bit-parallel multiplier. Section 6 presents a summary and conclusion.

2. Preliminaries

Given a normal basis $\underline{N}$=$\{\beta^{2^0},\beta^{2^1},\beta^{2^2},...,\beta^{2^{n-1}}\}$ of $GF(2^n)$ over $GF(2)$, a field element A can be represented by a binary vector $(a_0,a_1,...,a_{n-1})$ with respect to (w.r.t.) this NB as $A=\sum_{i=0}^{n-1}a_i\beta^{2^i}=(a_0,a_1,...,a_{n-1})*N^T$, where $N=(\beta^{2^0},\beta^{2^1},\beta^{2^2},...,\beta^{2^{n-1}})$, * denotes the vector inner product and T denotes the vector transposition. β is called a normal element.

Throughout this paper, $<x>_i$ denotes the non-negative residue of x mod i, particularly, $<x>$ denotes the non-negative residue of x mod n. We define $v=(n-1)/2$ for n odd or $v=n/2$ for n even.

2.1 Computation of A^{2^i} $(1\le i\le n-1)$

For simplicity, let A_i= A^{2^i} $(0\le i\le n-1)$, i.e., A_i is the i-fold right cyclic shifts of the binary vector representation of A. Since A_i is used in software implementations, we now show how to compute it efficiently for $1\le i\le n-1$.

Let z be the full width of data-path of the general-purpose processor, e.g., z=32 for Pentium CPU. We assume that $z<v$ and define the array DA as follows:

$DA_0=(a_0,a_1,...,a_{n-1},a_0,a_1,...,a_{n-1})$ and

$DA_j=(a_j,...,a_{n-1},a_0,a_1,...,a_{n-1}a_0,...,a_{j-1})$, where $1\le j<z$, i.e., DA_j is the j-fold left cyclic shifts of the 2n-bit vector DA_0.

In software implementations, DA is defined as a 2-dimensional array $DA[z][\lceil\frac{2n}{z}\rceil]$, and each DA_j is stored in $\lceil\frac{2n}{z}\rceil$ successive computer words. So for $1\le i\le n-1$, A_{n-i} is stored in $\lceil\frac{n}{z}\rceil$ successive computer words starting from $DA[s][t]$ and ending at $DA[s][t+\lceil\frac{n}{z}\rceil-1]$, where $t=\lfloor i/z\rfloor$, $s=i$ & $(z-1)$ and & denotes the bit-wise AND. That is to say, two indexes of A_{n-i} can be computed at the cost of 1 binary shift ($t=\lfloor i/z\rfloor$) and 1 bit-wise AND. Moreover, the starting address of A_{n-i}, namely, the address of $DA[s][t]$, may be calculated in the precomputation procedure.

The time complexity to compute all DA_j is about 2z n-bit cyclic shift operations, and 2zn bits are required to store DA_j, where $0\le j<z$.

Now we introduce the RH algorithm of [4] and [5], and then we will present two slight improvements on this algorithm.

2.2 RH algorithm

For $1 \le i \le n-1$, let $\beta\beta^{2^i} = \sum_{j=0}^{n-1} \phi_{i,j} \beta^{2^j}$ be the expansion of $\beta\beta^{2^i}$ w.r.t. the NB $\underline{N}$, where $\phi_{i,j} \in GF(2)$. Let $S_i = \{ j \mid \phi_{i,j} = 1 \}$ and h_i=|S_i|. So we have $\beta\beta^{2^i} = \sum_{k \in S_i} \beta^{2^k}$. Note that for a particular normal basis $\underline{N}$ the representation of $\beta\beta^{2^i}$ is fixed.

D=AB can be computed by the following formula [3], [4], [5]:

$$D = AB = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j \beta^{2^i} \beta^{2^j} = \sum_{i=0}^{n-1} a_i b_i \beta^{(2^0+1)2^i} + \sum_{i=1}^{v}\sum_{j=0}^{n-1} (a_{\langle i+j \rangle} b_j + b_{\langle i+j \rangle} a_j) \beta^{(2^i+1)2^j} . \tag{1}$$

$$= \sum_{i=0}^{n-1} a_i b_i \beta^{2^{<i+1>}} + \sum_{i=1}^{v}\sum_{k \in S_i} \left[\sum_{j=0}^{n-1} \left(a_{\langle i+j \rangle} b_j + b_{\langle i+j \rangle} a_j \right) \beta^{2^{\langle j+k \rangle}} \right]. \tag{2}$$

Recall that A_i is defined as $A_i = A^{2^i}$, now we define B & A_{n-i} as follows:

$$B \,\&\, A_{n-i} = (a_i b_0, a_{\langle i+1 \rangle} b_1, ..., a_{\langle i-1 \rangle} b_{n-1}) . \tag{3}$$

Viewing B & A_{n-i} as a field element, we have $\sum_{j=0}^{n-1} a_{\langle i+j \rangle} b_j \beta^{2^{\langle j+k \rangle}} = (B \,\&\, A_{n-i})_k$ and (2) may be rewritten as follows:

$$D = (B \,\&\, A)_1 + \sum_{i=1}^{v}\sum_{k \in S_i} \left[(B \,\&\, A_{n-i}) + (A \,\&\, B_{n-i}) \right]_k . \tag{4}$$

Furthermore, we define $R[i]$ as follows:

$$\begin{cases} R[i] = (B \,\&\, A_{n-i}) + (A \,\&\, B_{n-i}) & \text{where } 1 \le i \le v-1, \\ R[v] = \begin{cases} (B \,\&\, A_{n-v}) + (A \,\&\, B_{n-v}) \\ A \,\&\, B_v \end{cases} & \begin{array}{l} \text{if } n \text{ is odd,} \\ \text{if } n \text{ is even.} \end{array} \end{cases}$$

Then we have

$$D = (B \,\&\, A)_1 + \sum_{i=1}^{v}\sum_{k \in S_i} (R[i])_k . \tag{5}$$

The RH algorithm for n odd is based on this formula, and it is presented in Appendix 1. For n even, (5) may be further improved by Lemma 1 of [3]. Please refer to [4] and [5] for more details. Now we present two improvements on the RH algorithm.

Since $0 \le k \le n-1$, we may interchange the order of the summation in (5) and obtain the formula:

$$D = (B \& A)_1 + \sum_{k=0}^{n-1} \left(\sum_{i:\ 1 \le i \le v \text{ and } k \in S_i} R[i] \right)_k \cdot \tag{6}$$

Based on this formula, an improved software multiplication algorithm, which is called A1, is presented in Appendix 2.

Now we show the second improvement. From (3) we know that

$(A \& B)_i$= $(A \& B)^{2^i} = (a_{n-i}b_{n-i}, a_{\langle n-i+1 \rangle}b_{\langle n-i+1 \rangle}, ..., a_{n-1}b_{n-1}, a_0 b_0, ..., a_{\langle n-i-1 \rangle}b_{\langle n-i-1 \rangle})$ =(A_i & B_i).

Thus (4) can be rewritten as:

$$D = (B_1 \& A_1) + \sum_{i=1}^{v} \sum_{k \in S_i} \left[(B_k \& A_{<k-i>}) + (A_k \& B_{<k-i>}) \right]. \tag{7}$$

Since elements of S_i are in the range of [0, n-1], (7) can be rewritten as:

$$D = (B_1 \& A_1) + \sum_{k=0}^{n-1} \sum_{i: 1 \le i \le v \text{ and } k \in S_i} \left[(B_k \& A_{<k-i>}) + (A_k \& B_{<k-i>}) \right]$$

$$= (B_1 \& A_1) + \sum_{k=0}^{n-1} \left(\left(B_k \& \left(\sum_{i: 1 \le i \le v \text{ and } k \in S_i} A_{<k-i>} \right) \right) + \left(A_k \& \left(\sum_{i: 1 \le i \le v \text{ and } k \in S_i} B_{<k-i>} \right) \right) \right). \tag{8}$$

Based on this formula, a software multiplication algorithm, which is called A2, is presented in Appendix 3.

Similarly, for even n, we have the following formula:

$$D = (B_1 \& A_1) + \sum_{i=1}^{v-1} \sum_{k \in S_i} \left[(B_k \& A_{<k-i>}) + (A_k \& B_{<k-i>}) \right] + \sum_{k \in S_v} (B_k \& A_{<k-v>}).$$

$$= (B_1 \& A_1) + \sum_{k=0}^{n-1} \left(\left(B_k \& \left(\sum_{i: 1 \le i \le v \text{ and } k \in S_i} A_{<k-i>} \right) \right) + \left(A_k \& \left(\sum_{i: 1 \le i \le v-1 \text{ and } k \in S_i} B_{<k-i>} \right) \right) \right). \tag{9}$$

3. Proposed Algorithm And Software Implementations

From now on, we assume that the multiplicative order of the normal element is m, i.e., m is the least positive integer such that $\beta^m = 1$.

The ordered cyclotomic cosets (OCC) mod m is defined as follows:

Definition 1. *Ordered cyclotomic cosets mod m are the following ordered sets*:

$C_0=\{<2^0>_m, <2^1>_m, <2^2>_m,...,<2^{n-1}>_m\}$, $<2^0>_m=1$ *is called the coset leader*;

$C_i=\{<(2^i+1)2^0>_m, <(2^i+1)2^1>_m, <(2^i+1)2^2>_m,...,<(2^i+1)2^{n-1}>_m\}$, *where* $0<i<n$ *and* $<(2^i+1)2^0>_m$ *is called the coset leader*.

In this definition, C_i is an ordered set and elements of C_i need not to be distinct. For example, $C_1=\{3, 6, 12, 3, 6, 12\}$ for $m=21$ and $n=6$. Sometimes we also view C_i as a set and call it the set C_i. The following lemma shows that there are at most $v+1$ distinct sets C_is.

Lemma 1. *For* $0<i<n$, *the set* C_i *and the set* C_{n-i} *are identical*.

Proof: It follows immediately from the fact $m|(2^n-1)$ and the congruence

$$2^i+1 \equiv 2^i+2^n \equiv (2^{n-i}+1)2^i \quad \text{mod } (2^n-1). \qquad \square$$

To clarify the description of the new algorithm, we first give a simple example.

3.1 An Example

$GF(2^8)$ has been standardized for space communication by ESA and NASA, and for use in CD players [20], so it is desirable to design low complexity multipliers for the field. Gaussian NB is widely used to design low complexity NB multipliers. But no GNB exists in $GF(2^8)$. In Section 5, we will show that the time complexity of the proposed GF(2^8) NB multiplier matches the best result available in the open literature and its gate complexity is less than the best result.

Let $\beta \in GF(2^8)$ be a root of the irreducible polynomial $f(x)=x^8+x^7+x^6+x^4+x^2+x+1$. From Appendix C of [14], we know that β is a normal element of multiplicative order 17.

Consider the following 3 OCCs mod 17:

$C_0 = \{2^0, 2^1, 2^2,...,2^7\} = \{1, 2, 4, 8, 16, 15, 13, 9\}$ with the coset leader $r_0=2^0=1$;

$C_1 = \{3\cdot2^0, 3\cdot2^1, 3\cdot2^2,...,3\cdot2^7\} = \{3, 6, 12, 7, 14, 11, 5, 10\}$ with the coset leader $r_1=3\cdot2^0=3$;

$C_4 = \{0, 0, 0, 0, 0, 0, 0, 0\}$ with the coset leader $r_4=0$.

For $0 \le i \le 4$, $<2^i+1>_{17}$ belongs to just one OCC, i.e.,

$i=0$: $<2^0+1>_{17}=2 \in C_0$ and $2^{f_i} \cdot r_{e_i} \equiv 2^i+1$ mod (17) holds for $e_0=0$ and $f_0=1$;

$i=1$: $<2^1+1>_{17}=3 \in C_1$ and $2^{f_i} \cdot r_{e_i} \equiv 2^i+1$ mod (17) holds for $e_1=1$ and $f_1=0$;

$i=2$: $<2^2+1>_{17}=5 \in C_1$ and $2^{f_i} \cdot r_{e_i} \equiv 2^i+1$ mod (17) holds for $e_2=1$ and $f_2=6$;

i=3: $<2^3+1>_{17}=9 \in C_0$ and $2^{f_i} \cdot r_{e_i} \equiv 2^i + 1$ mod (17) holds for e_3=0 and f_3=7;

i=4: $<2^4+1>_{17}=0 \in C_4$ and $2^{f_i} \cdot r_{e_i} \equiv 2^i + 1$ mod (17) holds for e_4=4 and f_4=0;

where the congruence $2^{f_i} \cdot r_{e_i} \equiv 2^i + 1$ mod (17) means that $<2^i+1>_{17}$ is the f_i-th element of the OCC C_{e_i} and f_i is the smallest nonnegative integer such that the congruence holds.

Associated with each C_j (j=0, 1, 4), a $GF(2^8)$ vector $N_j=((\beta^{r_j})^{2^0}, (\beta^{r_j})^{2^1}, (\beta^{r_j})^{2^2}, ..., (\beta^{r_j})^{2^{8-1}})$ is defined as follows:

$N_0=N=(\beta^{2^0}, \beta^{2^1}, \beta^{2^2}, ..., \beta^{2^{8-1}})$;

$N_1=(\gamma^{2^0}, \gamma^{2^1}, \gamma^{2^2}, ..., \gamma^{2^{8-1}})$; where $\gamma = \beta^3 = \beta^2 + \beta^{16} + \beta^{32} + \beta^{128}$ and

$N_4=\beta^{0\times 2^0} = (1,1,1,1,1,1,1,1)$.

Let $A = \sum_{i=0}^{7} a_i \beta^{2^i} = (a_0, a_1, ..., a_7) * N^T$ and $B = \sum_{i=0}^{7} b_i \beta^{2^i} = (b_0, b_1, ..., b_7) * N^T$ be two elements of $GF(2^8)$. $D=AB$ can be computed by the following formula:

$$D=AB=(a_7b_7, a_0b_0, a_1b_1, ..., a_6b_6)*N^T + (a_0b_1+b_0a_1, a_1b_2+b_1a_2, ..., a_7b_0+b_7a_0)*N_1^T +$$
$$(a_2b_4+b_2a_4, a_3b_5+b_3a_5, ..., a_1b_3+b_1a_3)*N_1^T + (a_1b_4+b_1a_4, a_2b_5+b_2a_5, ..., a_0b_3+b_0a_3)*N^T +$$
$$(a_0b_4, a_1b_5, a_2b_6, ..., a_7b_3)*N_4^T$$
$$=T_0*N^T+T_1*N_1^T+T_2*N_1^T+T_3*N^T+T_4*N_4^T.$$

The five inner products are denoted as T_0*N^T, $T_1*N_1^T$, $T_2*N_1^T$, T_3*N^T and $T_4*N_4^T$, as shown above. Arrays e_i and f_i ($0 \le i \le 4$) are used to determine these inner products, e.g.,

$$T_2 * N_1^T = \left((a_{8-f_2}b_{2+8-f_2} + b_{8-f_2}a_{2+8-f_2}), (a_3b_5 + b_3a_5), ..., (a_{8-f_2-1}b_{2+8-f_2-1} + b_{8-f_2-1}a_{2+8-f_2-1})\right) * N_{e_2}^T,$$

where f_2=6 and e_2=1.

Now T_0 and T_3 have already been represented in the NB $\underline{N}$. T_1 and T_2 can be combined into one vector. After converting this vector and T_4 into the $GF(2)$ vector representation w.r.t. $\underline{N}$, we get the $GF(2)$ vector representation of $D=AB$ w.r.t. $\underline{N}$.

3.2 Ordered Cyclotomic Cosets mod *m*

Lemma 1 shows that there are at most v+1 distinct sets C_is. However, the accurate number is often less than v+1, e.g., the example presented in Section 3.1. Throughout this paper, u denotes the number of different sets C_is. The following procedure determines the value of u and these sets.

INPUT: m (the order of the normal element).

OUTPUT: 1. integer u, OCC C_j's, where $0 \le j \le u-1$;

2. arrays r_j, e_i and f_i, where $0 \le j \le u-1$ and $0 \le i \le v$.

```
S1:  set $R = \varnothing$;
S2:  OCC $C_0 = \{\langle 2^0\rangle_m, \langle 2^1\rangle_m, \langle 2^2\rangle_m,\dots,\langle 2^{n-1}\rangle_m\}$;
S3:  $r_0=2^0=1$;   $e_0=0$;   $f_0=1$;   $R = R \cup C_0$;   $u=1$;
S4:  for $i=1$ to $v$ do
S5:      if $\langle 2^i+1\rangle_m \notin R$ then { /* a new OCC */
S6:          OCC $C_u = \{\langle(2^i+1)2^0\rangle_m, \langle(2^i+1)2^1\rangle_m, \langle(2^i+1)2^2\rangle_m,\dots,\langle(2^i+1)2^{n-1}\rangle_m\}$;
S7:          $r_u=\langle(2^i+1)2^0\rangle_m$;   $e_i=u$;   $f_i=0$;   $R = R \cup C_u$;   $u=u+1$;
         }else{
S8:          Find $j$ such that $0 \le j \le u-1$ and $\langle 2^i+1\rangle_m \in C_j$;
S9:          Let $f$ be the smallest nonnegative integer such that $2^f \cdot r_j \equiv 2^i+1 \mod (m)$;
S10:         $e_i=j$;   $f_i=f$; }
```

Notes: 1. Since the proposed algorithm is applicable only to $GF(2^n)$ that the value of m is small, one may first use reference [10] to find a small factor of 2^n-1, and then check the existence of a normal element whose multiplicative order equals this factor.

2. The following expression holds for $0 \le i \le v$:

$$\left\langle 2^i+1 \right\rangle_m = \left\langle 2^{f_i} \cdot r_{e_i} \right\rangle_m \in C_{e_i}. \tag{10}$$

Arrays e_i and f_i will be used in the proposed algorithm, and they have the following meaning: $\langle(2^i+1)2^0\rangle_m$ is the f_i-th element of the OCC C_{e_i}, where f_i is the smallest nonnegative integer such that (10) holds.

Associated with each OCC C_j, a $GF(2^n)$ vector N_j is defined as follows, where $0 \le j \le u-1$:

$$N_j = ((\beta^{r_j})^{2^0}, (\beta^{r_j})^{2^1}, (\beta^{r_j})^{2^2}, \dots, (\beta^{r_j})^{2^{n-1}}). \tag{11}$$

Since $r_0=1$, we have $N_0=N$. It is easy to see that the vector N_j possesses the following property:

$$\left((a_0, a_1, \dots, a_{n-1}) * N_j^T\right)^2 = (a_{n-1}, a_0, a_1, \dots, a_{n-2}) * N_j^T. \tag{12}$$

3.3 New Multiplication Algorithm

We assume that *n* is odd, unless otherwise stated. Recall that the multiplicative order of β is *m*. By substituting (10) into (1), we have

$$D = \sum_{j=0}^{n-1} a_j b_j \beta^{r_{e_0} \cdot 2^{f_0+j}} + \sum_{i=1}^{v}\sum_{j=0}^{n-1} (a_{\langle i+j\rangle} b_j + b_{\langle i+j\rangle} a_j)\beta^{r_{e_i} \cdot 2^{f_i+j}} \ . \qquad (13)$$

Now let $k=f_i+j$, (13) can be rewritten as:

$$D = \sum_{k=0}^{n-1} a_{<k-f_0>} b_{<k-f_0>} (\beta^{r_{e_0}})^{2^k} + \sum_{i=1}^{v}\sum_{k=0}^{n-1} (a_{<i-f_i+k>} b_{<k-f_i>} + b_{<i-f_i+k>} a_{<k-f_i>})(\beta^{r_{e_i}})^{2^k} \ . \qquad (14)$$

From (11) and (3), we know that

$$\sum_{k=0}^{n-1} (a_{<i-f_i+k>} b_{<k-f_i>})(\beta^{r_{e_i}})^{2^k} = \left(A_{<f_i-i>} \ \& \ B_{f_i}\right) * N_{e_i}{}^T \ . \qquad (15)$$

Applying (15) to (14), we have

$$D = \left(A_{f_0} \ \& \ B_{f_0}\right) * N_{e_0}{}^T + \sum_{i=1}^{v}\left(\left(A_{f_i} \ \& \ B_{<f_i-i>}\right) + \left(B_{f_i} \ \& \ A_{<f_i-i>}\right)\right) * N_{e_i}{}^T \ . \qquad (16)$$

Since $e_0=0$, $f_0=1$ and $0 \le e_i \le u-1$, (16) can be rewritten as follows:

$$D = \left(A_1 \ \& \ B_1\right) * N_0{}^T + \sum_{j=0}^{u-1}\left(\sum_{i:\ j=e_i \text{ and } 1\le i\le v}\left(\left(A_{f_i} \ \& \ B_{<f_i-i>}\right) + \left(B_{f_i} \ \& \ A_{<f_i-i>}\right)\right)\right) * N_j{}^T \ . \qquad (17)$$

Now we define the binary vectors D_j as follows:

$$\begin{cases} D_0 = (A_{f_0} \ \& \ B_{f_0}) + \displaystyle\sum_{i:\ 0=e_i \text{ and } 1\le i\le v}\left(\left(A_{f_i} \ \& \ B_{<f_i-i>}\right) + \left(B_{f_i} \ \& \ A_{<f_i-i>}\right)\right) , \\ D_j = \displaystyle\sum_{i:\ j=e_i \text{ and } 1\le i\le v}\left(\left(A_{f_i} \ \& \ B_{<f_i-i>}\right) + \left(B_{f_i} \ \& \ A_{<f_i-i>}\right)\right) , \end{cases} \qquad (18)$$

where $1 \le j \le u-1$.

Applying this definition to (17), we have

$$D = D_0 * N_0{}^T + \sum_{j=1}^{u-1} D_j * N_j{}^T \ . \qquad (19)$$

Since $N_0=N$, D_0 is just the *GF*(2) vector representation of D_0*N^T w.r.t. the NB $\underline{N}$. Each $D_j*N_j^T$ ($0<j<u$) is also a field element, but D_j is not the *GF*(2) vector representation of $D_j*N_j^T$ w.r.t. $\underline{N}$. So we need to convert $D_j*N_j^T$ into the *GF*(2) vector representation w.r.t. $\underline{N}$. This may be done by the table lookup approach in software implementations. Now view D_j as an integer. Let $(d_{j,n-1},...,d_{j,1},$

$d_{j,0})_2$ be the binary representation of D_j and w ($0<w<n$) be the block size. We first divide D_j into $\lceil \frac{n}{w} \rceil$ blocks $D_{j,h}$ as follows:

$$D_j * N_j^T = \sum_{i=0}^{n-1} d_{j,i}\, \gamma^{2^i} = \sum_{h=0}^{\lceil \frac{n}{w} \rceil - 1} \sum_{i=0}^{w-1} d_{j,wh+i}\, \gamma^{2^{wh+i}} = \sum_{h=0}^{\lceil \frac{n}{w} \rceil - 1} \left(\sum_{i=0}^{w-1} d_{j,wh+i}\, \gamma^{2^i} \right)^{2^{wh}} = \sum_{h=0}^{\lceil \frac{n}{w} \rceil - 1} D_{j,h}^{\;2^{wh}}\ , \qquad (20)$$

where $\gamma = \beta^{r_j}$ and the binary representation of the block $D_{j,h}$ is $(d_{j,wh+w-1},\ldots,d_{j,wh+1}, d_{j,wh})_2$. In (20), the term $d_{j,i}$ is defined as $d_{j,i}=0$ for $n \le i \le w\lceil \frac{n}{w} \rceil - 1$.

The $GF(2)$ vector representation of $D_j*N_j^T$ w.r.t. $\underline{N}$ can be computed by the $GF(2^n)$ addition operation if the $GF(2)$ vector representation of each $D_{j,h}$ w.r.t. $\underline{N}$ is available. In software implementations, the $GF(2)$ vector representation of $D_{j,h}$ is stored in the precomputation table TB, which is defined as follows:

```
for  $j$=1  to  $u$-1  do
    for  $c$=0  to  $2^w$-1  do {
        Let $(c_{w-1}c_{w-2} \ldots c_1c_0)_2$ be the binary representation of the integer $c$;
        $TB[j][c]$ = $GF(2)$ vector representation of $\sum_{i=0}^{w-1} c_i\, \beta^{r_j 2^i}$  w.r.t. $\underline{N}$;}
```

Note that the table TB depends only on $\underline{N}$, i.e., once the NB is chosen the table can be created during the precomputation stage.

Since the binary representation of the block $D_{j,h}$ is $(d_{j,wh+w-1},\ldots,d_{j,wh+1}, d_{j,wh})_2$, (20) can be rewritten as follows:

$$D_j * N_j^T = \sum_{h=0}^{\lceil \frac{n}{w} \rceil - 1} TB[j][(d_{j,wh+w-1},\ldots,d_{j,wh+1},d_{j,wh})_2]^{2^{wh}}\ . \qquad (21)$$

Thus D=AB can be computed by the following formula.

$$D = AB = D_0 + \sum_{j=1}^{u-1} \sum_{h=0}^{\lceil \frac{n}{w} \rceil - 1} TB[j][(d_{j,wh+w-1},\ldots,d_{j,wh+1},d_{j,wh})_2]^{2^{wh}}\ . \qquad (22)$$

Based on (18), (19), (22) and Section 2.1, it is straightforward to present the following multiplication algorithm for odd values of n.

Multiplication Algorithm A3 for *n* Odd:

INPUT: A, B, w, u, e_i, f_i, and $TB[j][c]$, where $0 \le i \le v$, $0 \le c \le 2^w - 1$ and $1 \le j \le u - 1$.

```
OUTPUT:   $D$=$AB$.
  S1:  Compute $DA_i$ and $DB_i$ for $0 \le i < z$;
  S2:  $D_0 = A_{f_0} \& B_{f_0}$ =$(A\&B)_1$;
  S3:  for $j = 1$ to $u$-1 do $D_j$=(0,0,...,0);
  S4:  for $i = 1$ to $v$ do $D_{e_i} = D_{e_i} \oplus \left(A_{f_i} \& B_{<f_i-i>}\right) \oplus \left(B_{f_i} \& A_{<f_i-i>}\right)$;
  S5:  for $j = 1$ to $u$-1 do
  S6:       for $h = 0$ to $\lceil \frac{n}{w} \rceil$-1 do $D_0 = D_0 \oplus (TB[j][D_{j,h}])_{wh}$;
  S7:  Output $D_0$;
```

Notes: 1. Similar to Algorithm A1, starting addresses of $A_{f_i}, A_{<f_i-i>}, B_{f_i}, B_{<f_i-i>}$ and D_{e_i} in S4 may be stored sequentially in a 1-dimensinal array for $1 \le i \le v$. The size of this array is 5*vz* bits.

2. Since $TB[j][D_{j,h}]$ is a $GF(2^n)$ element represented in the NB $\underline{N}$, $(TB[j][D_{j,h}])_{wh}$ may also be precomputed by the method introduced in Section 2.1. In S6, only *wh*-fold cyclic shifts of the binary vector representation of $TB[j][D_{j,h}]$ are required, thus it is easy to see that when w is selected as a multiple of the minimal width of the data-path of the CPU, *TB* has the smallest size: $2n(u\text{-}1)2^w$ bits. In our experiment, the value of w is 8.

Similarly, we have the following formula for even n:

$$D = \left(A_1 \& B_1\right) * N_0{}^T + \left(A_{f_v} \& B_{<f_v-v>}\right) * N_{e_v}{}^T + \sum_{i=1}^{v-1} \left(\left(A_{f_i} \& B_{<f_i-i>}\right) + \left(B_{f_i} \& A_{<f_i-i>}\right)\right) * N_{e_i}{}^T . \quad (23)$$

Recall that the multiplicative order of the normal element β is m, and m is a primitive factor of 2^n-1=$(2^v$-1$)(2^v$+1), i.e., $m|2^v$+1, we know that e_v=u-1, f_v=0 and N_{u-1}=(1,1,...,1) in the output of the procedure of Section 3.2. So (23) may be rewritten as

$$D = \left(A_1 \& B_1\right) * N_0{}^T + \left(A_0 \& B_v\right) * N_{u-1}{}^T + \sum_{i=1}^{v-1} \left(\left(A_{f_i} \& B_{<f_i-i>}\right) + \left(B_{f_i} \& A_{<f_i-i>}\right)\right) * N_{e_i}{}^T , \quad (24)$$

where $0 \le e_i \le u-2$ for $1 \le i \le v-1$

We may also compute the product D=AB using a lookup table. Since N_{u-1}={1,1,...,1}, $(A_0\&B_v)*N_{u-1}{}^T$ is actually an element of $GF(2)$, which is equal to the parity of $(A_0\&B_v)$. This observation may reduce the size of the lookup table.

3.4 Analysis and Comparison of Software Implementations

The time complexity of the RH algorithm is determined in [4] and [5]. It needs n n-bit AND operations and $\frac{n-1}{2}+\frac{C_N-1}{2}=(C_N+n-2)/2$ n-bit XOR operations. The number of n-bit cyclic shift operations of the RH algorithm is equal to $(C_N+2n-1)/2$.

Ning and Yin presented two algorithms in [13]: one is for ONB and the other is for nonoptimal normal bases. The algorithm for nonoptimal normal bases is slow, and we only consider the one for ONB. Since no description of the precomputation procedure was presented in [13] (parts of the NY algorithm were described in a patent application), we assume that the method introduced in Section 2.1 is used to perform this precomputation. For the NY algorithm, DA_j and DB_j are defined as $z\lceil \frac{n}{z} \rceil$ -bit vectors, thus the total number of cyclic shift operations is about $2z$. Based on this assumption, we know that the NY algorithm 3 of [13] requires about $2n$ n-bit XOR, $n+1$ n-bit AND and $2z$ n-bit cyclic shift operations.

Now we analyze the space complexity of Algorithm A3.

In line S1, $4zn$ bits are required to store arrays DA_j and DB_j ($0 \le j < z$). In line S2 and S3, variables D_js ($0 \le j < u-1$) require un bits. Note 2 shows that the table $TB[j][D_{j,h}]$ requires $2n(u-1)2^w$ bits. Therefore the space complexity of Algorithm A3 is $4zn+un+2n(u-1)2^w$ bits.

This size is reasonable on modern 32-bit computers. For example, if w=8 then it is 2709966 bits ($\approx$331 Kbytes) for $GF(2^{571})$ (u=10).

In Algorithm A3, starting addresses of $A_{f_i}, B_{<f_i-i>}, B_{f_i}, B_{<f_i-i>}, D_{e_i}$, and $(TB[j][D_{j,h}])_{wh}$ may be computed in the precomputation procedure, so it is easy to determine the time complexity.

The cyclic shift operation is required only in line S1. From Section 2.1, we know that the total number of cyclic shift operations in the algorithm is $4z$.

The numbers of AND operations in line S2 and S4 are 1 and $2v=n-1$, respectively. Thus the algorithm requires n AND operations.

For simplicity, we suppose that the statement $D_j=(0,0,...,0)$ of S3 can be performed at the cost of 1 n-bit XOR operation. In line S4, $2v=n-1$ XOR operations are required. The XOR operation of line S6 is repeated $(u\text{-}1)\lceil \frac{n}{w} \rceil$ times. Therefore the total number of XOR operations used in the algorithm is $(\lceil \frac{n}{w} \rceil+1)(u\text{-}1)+n\text{-}1$.

Table 1 compares the time and space complexities of these NB algorithms in $GF(2^n)$ where n is odd. In the table, #Variables denotes the size of the temporary variables, and #Pointers denotes the size of the 1-dimensional array discussed in Note 1 of Algorithms A1, A2 and A3.

TABLE 1: Comparison of NB multiplication algorithms (n odd).

Algorithm	No. of important n-bit operations			Memory (in bits)	
	XOR	AND	Cyclic shift	#Variables	#pointers
RH	$(C_N+n\text{-}2)/2$	n	$(C_N+2n\text{-}1)/2$	$3n+z(C_N+n\text{-}2)/2$	0
NY (ONB)	$2n$	$n+1$	$2z$	$4zn$	0
A1	$(C_N+n\text{-}2)/2$	n	$\leq n+3z$	$4zn+n(n+1)/2$	$z(2+2v)+z(C_N\text{-}1)/2$
A2	$C_N\text{-}1$	$\leq 2n+1$	$4z$	$5zn+2n$	$2zn+z(C_N\text{-}1)$
A3	$(\lceil \frac{n}{w} \rceil+1)(u\text{-}1)+n\text{-}1$	n	$4z$	$4zn+un+n(u\text{-}1)2^{w+1}$	$5zv$

We assume that the general-purpose processor can perform 1 n-bit XOR or AND operation using 1 n-bit operation. As defined in [5], we also assume that 1 cyclic shift operation needs ρ n-bit operations. Our experiments and [5] show that the value of ρ is typically 4 for the C programming language if only simple logical instructions, such as AND, SHIFT and OR are used to emulated a k-fold cyclic shifts. Now we compare these algorithms. We assume that w=8, ρ =4 and z=32, and obtain the following results:

1. Algorithm A1 is faster than the RH algorithm if $C_N > 193$.
2. Algorithm A1 is faster than Algorithm A2 if $C_N > 7n - 258$.
3. Algorithm A3 is faster than the NY algorithm if $n > 292$ and $u = \begin{cases} 2 & n \text{ odd,} \\ 3 & n \text{ even.} \end{cases}$.
4. Algorithm A3 is faster than Algorithm A1 if

$$u < 1 + \frac{C_N + 7n - 256}{2\lceil \frac{n}{8} \rceil + 2}. \tag{25}$$

Appendix 4 contains 204 values of n such that $100<n<1001$ and the inequality (25) is satisfied.

For memory constrained systems, Hasan proposed a lookup table-based polynomial basis multiplication algorithm in [9]. The size of the tables is $(w+d)2^w$, where d is the degree of the second leading coefficient of the field-defining polynomial. Now we consider these NB algorithms in such systems. For Algorithms A1, A2 and A3, we will not use arrays *DA*, *DB* and the 1-dimensional array introduced in Note 1 of these algorithms. Furthermore, we assume that values of e_k and cyclic shift count are hard-coded [5], i.e., they are stored as program codes. Table 2 compares the time and memory complexities of these NB algorithms. From the table, we conclude that RH and NY algorithms are better choices in memory constrained systems.

TABLE 2: Comparison of NB algorithms in memory constrained systems (n odd).

Algorithm	No. of important n-bit operations			Memory (in bits)
	XOR	AND	Cyclic shift	
RH	$(C_N+n-2)/2$	n	$(C_N+2n-1)/2$	$3n$
NY (ONB)	$2n$	$n+1$	$2z$	$4zn$
A1	$(C_N+n-2)/2$	n	$\leq 2n$	$n(n+3)/2$
A2	C_N-1	$\leq 2n+1$	$2n+C_N$	$2n$
A3	$(\lceil \frac{n}{w} \rceil+1)(u-1)+n-1$	n	$2n-1$	$un+n(u-1)2^{w+1}$

We implement these NB multiplication algorithms in ANSI C using Microsoft Visual C++ 6.0 complier. For comparison, we also implement the polynomial basis multiplication algorithm presented in [6], i.e., the finite field analogue of the Montgomery multiplication for integers. For simplicity, the Montgomery multiplication algorithm is implemented in $GF(2^k)$ instead of $GF(2^n)$, where $k = w\lceil \frac{n}{w} \rceil$. Reference [6] indicates that the case w=8 results in the fastest implementation on modern 32-bit computers. So we also select w=8, and employ the table lookup approach, which is shown to be the best choice to perform word-level multiplications [6].

Our experiments are performed on an IBM ThinkPad 770X notebook with a 300 MHz Pentium II CPU running Windows NT 4.0. Experimental results are listed in Table 3, where the first five fields are the binary fields recommended by NIST for ECDSA applications. Since the NY algorithm is slow for nonoptimal normal bases, we do not list timings for these fields. For $GF(2^n)$ that the value of u is large, the timing of A3 do not listed in the table either. In Table 3, Algorithm A2 is slightly slower than the Montgomery algorithm for $GF(2^{409})$. So for applications that many squaring operations are required, e.g. exponentiation, Algorithm A2 is a better choice.

TABLE 3: Timings for some $GF(2^n)$s (μs)

n	GNB type	u	RH	NY	Montgomery	A1	A2	A3
163	4	-	164	-	57	123	112	-
233	2	6	180	56	127	140	98	87
283	6	34	516	-	190	318	339	455
409	4	-	671	-	421	506	441	-
571	10	10	2454	-	819	1481	1684	525
251	2	2	192	64	147	151	105	56
491	2	2	590	226	612	493	321	178
577	4	6	1231	-	865	993	825	395
599	8	-	2241	-	896	1444	1524	-
563	14	-	3308	-	801	1782	2198	-

4. Mathematical Properties

We now present some mathematical properties of the new algorithm.

Recall the definition of the primitive factor of 2^n-1 in [15, p.173].

Definition 2. *The prime p is said to be a primitive factor of* 2^n-1 *iff the order of* 2 *mod p is n*.

Lemma 2. *If p is a primitive factor of* 2^n-1, *then* $\langle p\rangle_n=1$. *If n odd*, $\langle p\rangle_8=\pm 1$; *if* $\langle n\rangle_8=\pm 2$, $\langle p\rangle_8=1$ *or* 3; *if* $\langle n\rangle_8=0$, $\langle p\rangle_{2n}=1$ [15].

Given two $GF(2^n)$ normal elements A and B of multiplicative orders a and b, respectively. If $a|b$ then the number of different sets C_is mod a is not greater than that of b. Thus selecting A results in a better implementation of the algorithm. From [25 Theorem V], we know that 2^n-1 (n>1) posses at least one primitive factor with the single exception n=6. So we only consider the normal element of the prime order in this paper. We define the type of a normal element as follows:

Definition 3. *A normal element of* $GF(2^n)$ *over* $GF(2)$ *is said to be of type k if its multiplicative order is* $kn+1$ *and* $kn+1$ *is a primitive factor of* 2^n-1.

It seems that the smaller multiplicative order may result in the smaller value of u, which in turn results in a faster implementation of the proposed algorithm. But this is not always true. For example, 328177=318*1032+1 and 359137=348*1032+1 are two primitive factors of 2^{1032}-1, and in $GF(2^{1032})$ we have found two normal elements of orders 328177 and 359137, respectively. Our experiments show that their values of u are 268 and 267, respectively.

Now we show the relation between the GNB and the type k normal element. The type-I GNB are always called the type-I ONB, and they are determined by the following lemma [7]:

Lemma 3. $GF(2^n)$ *contains a type-I ONB iff* $n+1$ *is a prime and* 2 *is a primitive root mod* $n+1$.

From this lemma, the following theorem holds immediately.

Proposition 1. $GF(2^n)$ *contains a type* 1 *normal element iff* $n+1$ *is a prime and* 2 *is a primitive root mod* $n+1$.

All type-II GNB, which are called type-II ONB, are described by the following lemma [7,8]:

Lemma 4. *If*

(1) 2 *is primitive in* $\mathbf{Z}_{2n+1}$, *or*

(2) $2n+1$ *is a prime congruent to* 3 *modulo* 4 *and* 2 *generates the quadratic residues in* $\mathbf{Z}_{2n+1}$,

Let β *be a primitive* $(2n+1)$*st root of unity in* $GF(2^{2n})$ *and* $\gamma=\beta+\beta^{-1}$. *Then* γ *is a type*-II *ONB generator*.

If condition (1) holds then $(2n+1)\nmid 2^n-1$, and there is no element of order $2n+1$ in $GF(2^n)$. We call the normal basis generated by condition (2) the second class of type-II ONB. In this case, (2n+1) is a primitive factor of 2^n-1. We may expect that either β or β^{-1}, but not both, is a normal element of type 2. But this is not true. Our experiments show that for n<2000 such that the second class of type-II ONB exists in $GF(2^n)$, there are three fields that do not contain type 2 normal elements: $GF(2^{119})$, $GF(2^{791})$ and $GF(2^{1659})$.

On the other hand, there exist type 2 normal elements in some $GF(2^n)$s but no type-II ONB exists in these fields. For example, Appendix 4 lists some values of n such that 8|n and type 2 normal elements exist in $GF(2^n)$. But there is no GNB exists in these fields.

We now present some additional differences between the type t GNB and type t normal elements:

1. For some values of n and t (t>2) there exists type t GNB in $GF(2^n)$ but no normal element with u=t exists in the field, and vice versa.
2. For a given $GF(2^n)$ and the value of t, the field can have, at most, one type t GNB, but the field may contain more than one type t normal elements up to conjugacy, e.g., $GF(2^{468})$ have 6 type 16 normal elements up to conjugacy.
3. GNB generators always have high multiplicative orders, and are very often primitive [19].

5. New Bit-Parallel Multiplier

5.1 Architecture

We now assume that n is odd, unless otherwise stated. Because we have described the proposed algorithm for software implementations in detail, it is easy to understand its application to VLSI multipliers. The new multiplier works in a similar way of software implementations: it first computes each $D_j=(d_{j,0},d_{j,1},...,d_{j,n-1})$ by (18) and then converts each $d_{j,k}$ into the coordinate w.r.t. the selected NB $\underline{N}$.

The new multiplier first computes a_ib_j ($0 \le i, j \le n-1$), which is called a term, using exactly n^2 two-input AND gates. This operation requires a single AND gate delay T_A due to the parallelism. Then it computes coordinates of $D_j=(d_{j,0},d_{j,1},...,d_{j,n-1})$ by (18) as follows:

$$\begin{cases} d_{0,k} = a_{<k-f_0>}b_{<k-f_0>} + \sum\limits_{i:\ 0=e_i \text{ and } 1\le i\le v} \left(\left(a_{<k-f_i>}b_{<k-f_i+i>}\right)+\left(b_{<k-f_i>}a_{<k-f_i+i>}\right)\right), \\ d_{j,k} = \sum\limits_{i:\ j=e_i \text{ and } 1\le i\le v} \left(\left(a_{<k-f_i>}b_{<k-f_i+i>}\right)+\left(b_{<k-f_i>}a_{<k-f_i+i>}\right)\right), \end{cases} \tag{26}$$

where $0 \le k \le n-1$ and $1 \le j \le u-1$.

For a fixed *j* and $0 \le k \le n-1$, each $d_{j,k}$ has the same number of terms, which is denoted as $|d_{j,k}|$. Since each term a_ib_j ($0 \le i, j \le n-1$) is XORed only once, the total number of two-input XOR gates required in this step is n^2-nu.

For $0 \le j \le u-1$, $d_{j,0}$ may be computed using a binary XOR tree, and the time delay is at most $\lceil \log_2 Max(|d_{j,0}|) \rceil$ T_X due to the parallelism, where T_X is the delay of one two-input XOR gate.

Since $d_{0,k}$ is already the coordinate representation w.r.t. $\underline{N}$, we now need to convert each $d_{j,k}$ ($1 \le j \le u-1$) into the coordinate representation w.r.t. $\underline{N}$.

For $1 \le j \le u-1$, let $\beta^{r_j} = \sum_{t=0}^{n-1} \phi_{j,t}\beta^{2^t}$ be the expansion of β^{r_j} w.r.t. $\underline{N}$, where $\phi_{j,t} \in GF(2)$ and (10) holds for some *i* such that $1 \le i \le v$ and e_i=*j*. Let $H_j = \{t \mid \phi_{j,t} = 1\}$ and h_j=$|H_j|$. We may rewrite H_j as $H_j = \{w_{j,1}, w_{j,2}, ..., w_{j,h_j}\}$, where $0 \le w_{j,1} < w_{j,2} < ... < w_{j,h_j} \le n-1$, and obtain the following identity:

$$\beta^{r_j} = \sum_{k \in H_j} \beta^{2^k} = \sum_{k=1}^{h_j} \beta^{2^{w_{j,k}}} . \qquad (27)$$

By (11), we know that

$$D_j * N_j^T = \sum_{t=0}^{n-1} d_{j,t}\, \beta^{r_j \cdot 2^t} = \sum_{t=0}^{n-1} d_{j,t} \sum_{k=1}^{h_j} \beta^{2^{t+w_{j,k}}} = \sum_{k=1}^{h_j} \sum_{t=0}^{n-1} d_{j,t}\, \beta^{2^{t+w_{j,k}}} . \qquad (28)$$

The coefficient of β in (28) is $\sum_{k=1}^{h_j} d_{j,n-w_{j,k}}$. Therefore, from (19) we know that the expression of d_0, which is the coefficient of β in the final result of *D*=*AB*=(d_0,d_1,...,d_{n-1}), is

$$d_0 = d_{0,0} + \sum_{j=1}^{u-1} \sum_{k=1}^{h_j} d_{j,n-w_{j,k}} . \qquad (29)$$

The term d_0 may be computed by a binary XOR tree, which requires $\sum_{j=1}^{u-1} h_j$ XOR gates and $\left\lceil \log_2 \left(1 + \sum_{j=1}^{u-1} h_j \right) \right\rceil$ T_X gate delay. Therefore the total number of XOR gates required in this conversion step is $n \sum_{j=1}^{u-1} h_j$. Since the XOR gate delay to generate $d_{j,0}$ may be different for different values of *j*, the delay of $\left\lceil \log_2\left(1+\sum_{j=1}^{u-1} h_j\right)\right\rceil T_X$ in this conversion step is an upper bound. Take $GF(2^{468})$ (u=17) for example. For $0 \le j \le u-1$, values of $|d_{j,0}|$ are 29, 30, 34, 30, 32, 32, 22, 24, 24, 22, 26, 36, 36, 26, 36, 28 and 1, respectively. There are 6 type 16 normal elements up to conjugacy, and values of $\sum_{j=1}^{u-2} h_j$ for these normal elements are 3424, 3444, 3420, 3448, 3468, and 3456, respectively. We select the normal element of $\sum_{j=1}^{u-2} h_j$ =3420 to represent field elements. For this normal element, values of h_j are 232, 253, 233, 236, 225, 241, 216, 228, 229, 221, 225, 228, 216, 228 and 209, respectively, where $1 \le j \le u-2$.

From values of $|d_{j,0}|$, it is easy to see that generating $d_{0,k}$, $d_{1,k}$, $d_{3,k}$, $d_{4,k}$, $d_{5,k}$, $d_{6,k}$, $d_{7,k}$, $d_{8,k}$, $d_{9,k}$, $d_{10,k}$, $d_{13,k}$ and $d_{15,k}$ requires $5T_X$ and generating $d_{2,k}$, $d_{11,k}$, $d_{12,k}$ and $d_{14,k}$ requires $6T_X$, where $0 \le k \le n-1$. In (29), terms generated in $5T_X$ may be first XORed pairwisely, and then these summations and terms generated in $6T_X$ are XORed. The accurate time delay of this computation procedure is $\lceil \log_2(934+2486/2)\rceil T_X$=$12T_X$, where 934=$h_2$+$h_{11}$+$h_{12}$+$h_{14}$=253+225+228+228 and 2486=$\sum_{j=1}^{u-2} h_j$ -934=3420-934.

For even values of *n*, we have the following formula from (24):

$$\begin{cases} d_{0,k} = a_{<k-f_0>} b_{<k-f_0>} + \sum_{i:\ 0=e_i \text{ and } 1\le i\le v-1} \left(\left(a_{<k-f_i>} b_{<k-f_i+i>}\right) + \left(b_{<k-f_i>} a_{<k-f_i+i>}\right)\right), \\ d_{j,k} = \sum_{i:\ j=e_i \text{ and } 1\le i\le v-1} \left(\left(a_{<k-f_i>} b_{<k-f_i+i>}\right) + \left(b_{<k-f_i>} a_{<k-f_i+i>}\right)\right), \\ d_{u-1,k} = a_k b_{<k+v>}, \end{cases} \tag{30}$$

where $0 \le k \le n-1$ and $1 \le j \le u-2$.

The conversion step for *n* even is similar to that of *n* odd. The only difference is $g = D_{u-1} * N_{u-1}^T$. From the discussion after (24), we know that N_{u-1}=(1,1,...,1) and $g \in GF(2)$. Therefore we may first compute *g* using *n*-1 XOR gates and then XOR this single bit to each d_k ($0 \le k \le n-1$) at the cost of *n* XOR gates. This method needs only 2*n*-1 XOR gates.

In [22], this idea was generalized. Converting $D_j * N_j^T$ ($1 \le j \le u-1$) into the coordinate representation w.r.t. $\underline{N}$ requires nh_j XOR gates in (28). If nh_j>n+n(n-h_j), i.e., h_j>n/2, then we have the following variants of (27) and (28):

$$\beta^{r_j} = \sum_{k=0}^{n-1}\beta^{2^k} + \sum_{k=0}^{n-1}\beta^{2^k} + \sum_{k\in H_j}\beta^{2^k} = 1 + \sum_{0\le k\le n-1 \text{ and } k\notin H_j}\beta^{2^k} . \tag{31}$$

$$D_j * N_j^T = \sum_{t=0}^{n-1} d_{j,t}\ \beta^{r_j\cdot 2^t} = \sum_{t=0}^{n-1} d_{j,t} + \sum_{t=0}^{n-1}\ \sum_{0\le k\le n-1 \text{ and } k\notin H_j} d_{j,t}\beta^{2^{t+k}} . \tag{32}$$

Formula (32) requires $n+n(n-h_j)$ XOR gates to convert $D_j{*}N_j^T$.

5.2 Analysis and Comparison of Bit-Parallel Multipliers

The gate and time complexities of the new multiplier are listed in Table 4, where complexities of the multiplier in [3] are also listed. In the table, $Max(|d_{j,0}|)$ denotes the maximum value of $|d_{j,0}|$ for $0 \le j \le u-1$.

TABLE 4: Comparison of two NB multipliers.

Multipliers	#AND	#XOR	Time Delay
Proposed (n odd)	n^2	$n^2-nu+n\sum_{j=1}^{u-1} h_j$	$T_A+\lceil \log_2 Max(\lvert d_{j,0}\rvert)\rceil T_X+\left\lceil \log_2\left(1+\sum_{j=1}^{u-1} h_j\right)\right\rceil T_X$
Proposed (n even)	n^2	$n^2-nu+2n-1+n\sum_{j=1}^{u-2} h_j$	$T_A+\lceil \log_2 Max(\lvert d_{j,0}\rvert)\rceil T_X+\left\lceil \log_2\left(2+\sum_{j=1}^{u-2} h_j\right)\right\rceil T_X$
[3] Theorem 2	n^2	$n(C_N+n-2)/2$	$T_A+\lceil \log_2(C_N+1)\rceil T_X$

Since it is not easy to determine the explicit expression of h_j, we assume that $h_j=n/2$ and $C_N=tn$ for simplicity. Based on this assumption, we know that numbers of XOR gates required by two multipliers are about $\frac{(u+1)}{2}n^2 - nu$ and $\frac{(t+1)}{2}n^2 - n$, respectively. Therefore the new algorithm may be used to design multipliers in $GF(2^n)$s such that $u<t$. As enumerated in Appendix 4 ($100<n<1001$), there are 25 n values such that $8\nmid n$ and $u<t$.

Now we analyze the time complexity of the new multiplier. In Table 4, $Max(|d_{j,0}|)$ is used to determine the time delay. For a fixed j and $0 \le k \le n-1$, each $d_{j,k}$ has the same number of terms, and $|d_{j,k}|=n/u$ on average. Thus, time complexities of two multipliers are about $T_A+\lceil \log_2(n^2/2)\rceil T_X$ and $T_A+\lceil \log_2 tn\rceil T_X$, respectively. For the GNB of small value of t, the time delay of the new multiplier is larger than [3]. For a randomly selected NB, the average value of t is $n/2$ and the time delay of two multipliers are equal. So for $GF(2^n)$ that no GNB exists ($8|n$) and the value of u is small, the algorithm provides an new way to design multipliers in these fields. In Appendix 4 there are 22 n values such that $u<12$ and $8|n$.

We now show the performance of the new multiplier with four examples.

In [1] and [3], two type-I ONB multipliers are presented. Both multipliers require n^2-1 XOR gates and n^2 AND gates. Their gate delays are $T_A + (2 + \lceil \log_2(n-1) \rceil)T_X$ and $T_A + (1 + \lceil \log_2(n-1) \rceil)T_X$, respectively. For $GF(2^n)$ where type-I ONB exists, $N_0=N$ and $N_1=(1,1,...,1)$. Since $d_{1,0}$ consists of a single term, no XOR gate is required to compute D_1. It is easy to see that $n(n-2)$ XOR gates are required to compute D_0. Computing $g=D_1*N_1^T$, which is the parity of D_1, requires n-1 XOR gates, and the gate delay is $\lceil \log_2 n \rceil T_X$. The final result is obtained by adding this bit to every bits of D_0, and this needs n XOR gates and $1T_X$ gate delay. Thus the total number of XOR gates is n^2-1 and the gate delay is $T_A + (1 + \lceil \log_2 n \rceil)T_X$. Since $\lceil \log_2 n \rceil = \lceil \log_2(n-1) \rceil$ for $n>3$ and $8 \nmid n$, the new multiplier matches the best result presented in Table 3 of [3].

Table 5 shows a comparison of the new multiplier with the multiplier of [3] in three fields: $GF(2^{370})$, $GF(2^{468})$ and $GF(2^{571})$. From [24], we know that there exist cryptographically interesting elliptic curves in $GF(2^{468})$. Since the signal reuse method introduced in [3] is applicable to both multipliers, it is not used here. In Table 5, t denotes the GNB type and h denotes the minimal value of $\sum_{j=1}^{u-1} h_j$ (n odd) or $\sum_{j=1}^{u-2} h_j$ (n even).

TABLE 5: Comparison of NB multipliers for $GF(2^{370})$, $GF(2^{468})$ and $GF(2^{571})$

n	t	C_N	u	h	Multiplier	#AND	#XOR	Delay
370	6	2199	5	544	Proposed	370^2	337069	T_A+17T_X
					[3] Theorem 2	370^2	474895	T_A+12T_X
468	21	9899	17	3420	Proposed	468^2	1812563	T_A+18T_X
					[3] Theorem 2	468^2	2425410	T_A+14T_X
571	10	5637	10	2586	Proposed	571^2	1796937	T_A+18T_X
					[3] Theorem 2	571^2	1771813	T_A+13T_X

5.3 An Example

Now we illustrate the proposed multiplier for $GF(2^8)$. From Section 2.1, we have the following formulas for D_0, D_1 and D_2:

$D_0=(d_{0,0}, d_{0,1},...,d_{0,7})=T_0+T_3=(x_{1,4}+a_7b_7, x_{2,5}+a_0b_0 ,...,x_{0,3}+a_6b_6)$,

$D_1=(d_{1,0}, d_{1,1},...,d_{1,7})=T_1+T_2=(x_{2,4}+x_{0,1}, x_{3,5}+x_{1,2} ,...,x_{1,3}+x_{7,0})$ and

$D_2=(d_{2,0}, d_{2,1},...,d_{2,7})=T_4=(a_0b_4, a_1b_5, a_2b_6,...,a_7b_3)$, where $x_{i,j}=a_ib_j+ a_jb_i$.

Computing D_0 and D_1 requires $2n+3n=40$ XOR gates, and no XOR gate is required to compute D_2. Recall that $N_2=\{1,1,...,1\}$, the value of $g=D_2*N_2^T$ is equal to the parity of D_2, which can be computed using n-1=7 XOR gates. Since $N_1=\{\gamma^{2^0},\gamma^{2^1},...,\gamma^{2^{8-1}}\}$, where $\gamma=\beta^{2^1}+\beta^{2^4}+\beta^{2^5}+\beta^{2^7}$, we have the following formulas for $D=AB=(d_0, d_1,...,d_7)$:

$d_0=d_{0,0}+g+d_{1,7}+d_{1,4}+d_{1,3}+d_{1,1}=d_{0,0}+g+(d_{1,3}+d_{1,7})+d_{1,1}+d_{1,4}$,

$d_1=d_{0,1}+g+d_{1,0}+d_{1,5}+d_{1,4}+d_{1,2}=d_{0,1}+g+(d_{1,0}+d_{1,2})+d_{1,4}+d_{1,5}$,

$d_2=d_{0,2}+g+d_{1,1}+d_{1,6}+d_{1,5}+d_{1,3}=d_{0,2}+g+(d_{1,3}+d_{1,6})+(d_{1,1}+d_{1,5})$,

$d_3=d_{0,3}+g+d_{1,2}+d_{1,7}+d_{1,6}+d_{1,4}=d_{0,3}+g+(d_{1,2}+d_{1,7})+(d_{1,4}+d_{1,6})$,

$d_4=d_{0,4}+g+d_{1,3}+d_{1,0}+d_{1,7}+d_{1,5}=d_{0,4}+g+(d_{1,3}+d_{1,7})+d_{1,0}+d_{1,5}$,

$d_5=d_{0,5}+g+d_{1,4}+d_{1,1}+d_{1,0}+d_{1,6}=d_{0,5}+g+(d_{1,4}+d_{1,6})+d_{1,0}+d_{1,1}$,

$d_6=d_{0,6}+g+d_{1,5}+d_{1,2}+d_{1,1}+d_{1,7}=d_{0,6}+g+(d_{1,1}+d_{1,5})+(d_{1,2}+d_{1,7})$ and

$d_7=d_{0,7}+g+d_{1,6}+d_{1,3}+d_{1,2}+d_{1,0}=d_{0,7}+g+(d_{1,0}+d_{1,2})+(d_{1,3}+d_{1,6})$.

The signal reuse technique of [3] is employed in these formulas. Partial sums in the brackets can be reused, e.g., both d_0 and d_4 have common terms $(d_{1,3}+d_{1,7})$. Therefore the total number of XOR gates used in this conversion step is reduced from 40 to 34, and the total number of XOR gates of the multiplier is 40+7+34=81.

Generating g needs 1 T_A and 3 T_X time delay, and generating each $d_{i,j}$ needs 1 T_A and 2 T_X due to the parallelism. So the total gate delay of the multiplier is $1T_A+5T_X$.

From Table 2 of [21], we know that the minimal value of C_N is 21 for $GF(2^8)$, and the formula for $D=AB=(d_0, d_1,...,d_7)$ is

$d_i=a_{<i+7>}b_{<i+7>}+x_{i+4,\, i+5}+x_{i+6,\, i+7}+x_{i+2,\, i+4}+x_{i+3,\, i+5}+x_{i+7,\, i+1}+$

$x_{i+3,\,i+6} + x_{i+6,\,i+1} + x_{i+0,\,i+3} + x_{i+2,\,i+6} + x_{i+3,\,i+7}$, where $0 \le i \le 7$ and $x_{k,j}=a_{<k>}b_{<j>}+a_{<j>}b_{<k>}$.

Using the symmetry property and the signal reuse technique of [3], we know that the $GF(2^8)$ multiplier of [3] requires 96 XOR gates and the total gate delay is $1T_A+5T_X$.

6. Conclusions

We have proposed a new NB multiplication algorithm for $GF(2^n)$. The proposed algorithm provides a new way to perform NB multiplication in some $GF(2^n)$s. The complexity of the proposed algorithm does not depend on C_N but the multiplicative order of the normal element. For these $GF(2^n)$s, the new algorithm can be used to design not only fast software multiplication algorithms but also low complexity bit-parallel VLSI multipliers. Especially, for some values of n that no Gaussian NB exists in $GF(2^n)$, i.e., $8|n$, the algorithm can still be used to design low complexity VLSI multipliers.

The proposed algorithm speedups software implementations by look-up table. But for VLSI implementations, the time complexity of the new multiplier is higher than the multiplier of [3] in $GF(2^n)$ where low complexity NB exists.

Two improvements on RH algorithm have been shown. Both theoretical and experimental comparisons of these NB algorithms have been presented. It is shown that they have specific behaviors in specific applications.

Appendix 1:

RH Multiplication Algorithm for *n* odd:

INPUT: A, B, S_i, where $1 \le i \le v$.

OUTPUT: $D=AB$.

```
S1:  $D = (A \,\&\, B)_1$;
S2:  for  $i = 1$  to  $v$  do {
S3:        $R = (B \,\&\, A_{n-i}) \oplus (A \,\&\, B_{n-i})$;
S4:        for each $k \in S_i$ do $D = D \oplus R_k$; }
S5:  Output $D$ ;
```

Appendix 2:

Multiplication Algorithm A1 for *n* Odd:

For each $0 \le k \le n-1$, the following precomputation procedure is to find all values of *i* such that $1 \le i \le v$ and $k \in S_i$.

A2.1 Precomputation:

INPUT: n, S_i, where $1 \le i \le v$.

OUTPUT: e_k and $m[k][j]$, where $0 \le k \le n-1$ and $0 \le j \le e_k - 1$.

```
S1:  for  $k = 0$  to  $n$-1 do  $e_k = 0$;
S2:  for  $i = 1$  to  $v$  do {
S3:      for  each  $k \in S_i$ do {
S4:          $m[k][e_k] = i$ ;
S5:          $e_k = e_k + 1$;  }  }
```

This procedure outputs e_k and $m[k][j]$, where $0 \le k \le n-1$ and $0 \le j \le e_k - 1$. e_k is the total number of *i*s such that $1 \le i \le v$ and $k \in S_i$, and $m[k][0]$ to $m[k][e_k-1]$ store these *i*s, i.e., $k \in S_{m[k][j]}$ for $0 \le j \le e_k - 1$.

A2.2 Algorithm A1 for *n* Odd:

INPUT: A, B, e_k and $m[k][j]$, where $0 \le k \le n-1$ and $0 \le j \le e_k - 1$.

OUTPUT: $D=AB$.

```
S1:  Compute $DA_i$ and $DB_i$ for $0 \le i < z$ ;
S2:  $D = A_1$ & $B_1$= $(A$ & $B)_1$;
S3:  for  $i = 1$  to  $v$  do    $R[i] = (B$ & $A_{n-i}) \oplus (A$ & $B_{n-i})$;
S4:  for  $k = 0$  to  $n$-1 do
S5:      if $e_k > 0$  then{
S6:          $C = R[m[k][0]]$;
S7:          for  $j = 1$  to  $e_k$-1 do   $C = C \oplus R[m[k][j]]$;
S8:          $D = D \oplus (C >> k)$;  }        // $C$>>$k$ denotes $k$-fold right cyclic shifts of $C$.
S9:  Output $D$ ;
```

Notes 1: For each $1 \le i \le v$, A_{n-i} is stored in $\lceil \frac{n}{z} \rceil$ successive computer words starting from $DA[s][t]$ and ending at $DA[s][t+\lceil \frac{n}{z} \rceil -1]$, where $t = \lfloor i/z \rfloor$, $s = i$ & $(z-1)$ and & denotes the bit-wise AND. In our implementation, these address computations are performed in the precomputation procedure, and starting addresses of A_1, B_1, A_{n-i}, B_{n-i} and $R[m[k][j]]$ are stored sequentially in a 1-dimensional array, where $1 \le i \le v$, $0 \le k \le n-1$ and $0 \le j \le e_k - 1$. Since $\sum_{k=0}^{n-1} e_k = \sum_{i=1}^{v} h_i = (C_N - 1)/2$, the size of this 1-dimensinal array is $z(2+2v) + z(C_N-1)/2$ bits.

2: Algorithm A1 may be implemented without computing arrays *DA* and *DB*. Experimental results show that arrays *DA* and *DB* speedup Algorithm A1 by no more than 10% for $GF(2^n)$s listed in Table 3 of Section 3. For example, Algorithm A1 without computing arrays *DA* and *DB* performs one multiplication operation in 1566 μs over $GF(2^{571})$.

A2.3 Complexity Analysis:

First we show that there is no need to define array DA_j as a $2n$-bit vector in line S1. Since only i-fold left cyclic shifts of A is required in line S3, where $1 \le i < v$, we may define array DA_j as a $(n+v)$-bit vector.

$DA_0 = (a_0,a_1,\ldots,a_{n-1},a_0,a_1,\ldots,a_{v-1})$ and

$DA_j = (a_j,\ldots,a_{n-1},a_0,a_1,\ldots,a_{v-1}a_0,\ldots,a_{j-1})$, where $1 \le j < z$.

From this definition, we know that the time complexity to compute DA_j's and DB_j's ($0 \le j < z$) is about $3z$ n-bit cyclic shift operations, and $3zn$ bits are required to store these arrays.

The cyclic shift operation is also required in line S8. Since e_k may be zero for some values of k, one can see that the total number of cyclic shift operations in Algorithm A1 is at most $n+3z$. For example, our experiments show that for $100<n<1000$, where n is odd and the type-II ONB exists in $GF(2^n)$, the minimal, average and maximal percentages that $e_k=0$ are 22.9%, 25.0% and 27.2% respectively.

XOR operations are required in S3, S7 and S8. In S3, the XOR operation is repeated $v=(n-1)/2$ times. Since $\sum_{k=0}^{n-1} e_k = (C_N - 1)/2$, the total number of XOR operations in line S7 and S8 is $(C_N - 1)/2$.

Therefore the total number of XOR operations in Algorithm A1 is $\frac{n-1}{2} + \frac{C_N - 1}{2} = (C_N + n - 2)/2$.

In line S2 and S3, 1 and $2v=n-1$ AND operations are required, respectively. Therefore Algorithm A1 requires n AND operations.

Now we analyze the space complexity of Algorithm A1.

In line S1, arrays DA_j and DB_j $(0 \le j < z)$ need $3zn$ bits. In line S3, the array $R[i]$ $(1 \le i < v)$ requires vn bits. In line S5, the array e_k $(0 \le k < n-1)$ requires zn bits. The temporary variable C of S6 needs n bits. From Note 1, we know that there is no need to store the array $m[k][j]$ in Algorithm A1. Therefore the space complexity of the Algorithm A1 is $3zn+vn+zn+n$ = $4zn+n(n+1)/2$ bits.

Appendix 3:

Multiplication Algorithm A2 for *n* Odd:

We only present the algorithm for *n* odd. In this case, both Algorithm A1 and A2 share the same precomputation procedure, which have been presented in Appendix A2.1.

A3.1 Algorithm A2 for *n* Odd:

INPUT: A, B, e_k and $m[k][j]$, where $0 \le k \le n-1$ and $0 \le j \le e_k - 1$.

OUTPUT: $D=AB$.

```
S1:  Compute $DA_i$ and $DB_i$ for $0 \le i < z$ ;
S2:  $D = A_1 \,\&\, B_1 = (A \,\&\, B)_1$;
S3:  for $k = 0$ to $n$-1 do
S4:      if $e_k > 0$ then{
S5:          $UA = A_{<k-m[k][0]>}$;   $UB = B_{<k-m[k][0]>}$;
S6:          for $j = 1$ to $e_k$-1 do { $UA = UA \oplus A_{<k-m[k][j]>}$;   $UB = UB \oplus B_{<k-m[k][j]>}$; }
S7:          $D = D \oplus (B_k \,\&\, UA) \oplus (A_k \,\&\, UB)$; }
S8:  Output $D$ ;
```

Note 1: In our implementation, starting addresses of A_k, B_k $A_{<k-m[k][j]>}$, and $B_{<k-m[k][j]>}$ in Algorithm A2 are also computed in the precomputation procedure and stored in a 1-dimensional array, where $0 \le k \le n-1$ and $0 \le j \le e_k - 1$. Since $\sum_{k=0}^{n-1} e_k = (C_N - 1)/2$, the size of this array is $2zn+z(C_N-1)$ bits.

A3.2 Complexity Analysis:

Since DA_j and DB_j are defined as $2n$-bit vectors, the number of cyclic shift operations, which are required only in line S1, is $4z$.

AND operations are required in S2 and S7. In line S2, one AND operation is required. Since e_k may be zero for some values of k, one can see that at most $2n$ AND operations are required in S7. Therefore the total number of AND operations in Algorithm A2 is at most $2n+1$.

Since $\sum_{k=0}^{n-1} e_k = (C_N - 1)/2$, the total number of XOR operations in Algorithm A2 is C_N-1.

Now we analyze the space complexity of Algorithm A2.

Since DA_j and DB_j ($0 \le j < z$) are defined as 2*n*-bit vectors in line S1, 4*zn* bits are required to store them. In line S4, the array e_k ($0 \le k < n-1$) requires *zn* bits. Temporary variables *UA* and *UB* of S5 need 2*n* bits. From Note 1, we know that there is no need to store the array *m*[*k*][*j*] in Algorithm A2. Therefore the space complexity of Algorithm A2 is 4*zn*+*zn*+2*n* = 2*n*+5*zn* bits.

Appendix 4:

This appendix lists all values of n such that $100<n<1001$ and one of the following two conditions is satisfied:

(1). $u<12$ and $8|n$;

(2). the inequality (25) is satisfied and type t GNB exists in $GF(2^n)$;

The table contains 22 values of n such that (1) is satisfied and 204 values of n such that (2) is satisfied. In the table, The value of u is the smallest value for $GF(2^n)$. Because type-I normal elements can be determined by theorem 1, they are not listed here.

(n, t, u) denotes that both the type t GNB and a normal element of u OCCs exists in $GF(2^n)$.

(n, -, u) denotes that a normal element of u OCCs exists in $GF(2^n)$ ($8|n$).

(110, 6, 24)	(114, 5, 6)	(117, 8, 8)	(121, 6, 6)	(131, 2, 2)	(135, 2, 2)
(146, 2, 13)	(152, -, 9)	(153, 4, 6)	(154, 25, 5)	(155, 2, 2)	(156, 13, 3)
(159, 22, 38)	(161, 6, 8)	(166, 3, 4)	(179, 2, 2)	(183, 2, 2)	(188, 5, 21)
(190, 10, 13)	(191, 2, 2)	(194, 2, 6)	(197, 18, 34)	(198, 22, 28)	(201, 8, 8)
(214, 3, 4)	(215, 6, 8)	(217, 6, 21)	(219, 4, 18)	(221, 2, 6)	(222, 10, 16)
(224, -, 3)	(230, 2, 4)	(231, 2, 2)	(233, 2, 6)	(236, 3, 6)	(237, 10, 6)
(239, 2, 2)	(243, 2, 2)	(244, 3, 4)	(245, 2, 6)	(251, 2, 2)	(258, 5, 5)
(262, 3, 5)	(266, 6, 18)	(270, 2, 4)	(274, 9, 5)	(284, 3, 3)	(286, 3, 8)
(295, 16, 16)	(298, 6, 5)	(299, 2, 2)	(303, 2, 2)	(305, 6, 6)	(308, 15, 28)
(317, 26, 30)	(318, 11, 20)	(323, 2, 2)	(325, 4, 22)	(332, 3, 4)	(333, 24, 6)
(336, -, 7)	(337, 10, 50)	(340, 3, 4)	(356, 3, 4)	(357, 10, 14)	(359, 2, 2)
(362, 5, 6)	(363, 4, 24)	(365, 24, 24)	(370, 6, 5)	(371, 2, 2)	(375, 2, 2)
(377, 14, 14)	(380, 5, 3)	(384, -, 3)	(386, 2, 18)	(397, 6, 6)	(400, -, 5)
(404, 3, 3)	(407, 8, 8)	(411, 2, 2)	(419, 2, 2)	(422, 11, 12)	(426, 2, 13)
(428, 5, 3)	(431, 2, 2)	(436, 13, 14)	(443, 2, 2)	(445, 6, 6)	(461, 6, 6)
(463, 12, 24)	(464, -, 3)	(468, 21, 17)	(472, -, 5)	(475, 4, 10)	(477, 46, 182)
(482, 5, 6)	(483, 2, 2)	(484, 3, 8)	(487, 4, 10)	(488, -, 3)	(491, 2, 2)
(492, 13, 7)	(495, 2, 2)	(497, 20, 14)	(500, 11, 15)	(506, 5, 9)	(515, 2, 2)

(519, 2, 2)	(525, 8, 8)	(526, 3, 4)	(531, 2, 2)	(532, 3, 4)	(534, 7, 8)
(537, 8, 8)	(542, 3, 4)	(543, 2, 2)	(544, -, 11)	(545, 2, 6)	(547, 10, 10)
(549, 14, 66)	(551, 6, 8)	(557, 6, 6)	(560, -, 9)	(566, 3, 4)	(568, -, 5)
(571, 10, 10)	(574, 3, 4)	(575, 2, 2)	(576, -, 7)	(577, 4, 6)	(594, 17, 41)
(596, 3, 4)	(598, 15, 5)	(601, 6, 6)	(605, 6, 6)	(611, 2, 2)	(615, 2, 2)
(624, -, 9)	(627, 20, 18)	(635, 8, 8)	(636, 13, 17)	(637, 4, 6)	(639, 2, 2)
(644, 3, 4)	(648, -, 3)	(650, 2, 6)	(651, 2, 2)	(659, 2, 2)	(662, 3, 9)
(663, 14, 72)	(667, 6, 18)	(679, 10, 10)	(680, -, 3)	(683, 2, 2)	(687, 10, 10)
(688, -, 7)	(697, 4, 24)	(704, -, 3)	(711, 8, 8)	(719, 2, 2)	(723, 2, 2)
(724, 13, 14)	(734, 3, 4)	(737, 6, 6)	(743, 2, 2)	(748, 7, 8)	(753, 16, 6)
(755, 2, 2)	(761, 2, 6)	(762, 10, 5)	(764, 3, 6)	(771, 2, 2)	(777, 16, 6)
(779, 2, 2)	(782, 3, 4)	(783, 2, 2)	(784, -, 5)	(788, 11, 7)	(790, 3, 8)
(793, 6, 6)	(794, 14, 18)	(803, 2, 2)	(806, 11, 9)	(810, 2, 9)	(814, 15, 20)
(831, 2, 2)	(836, 15, 9)	(838, 7, 8)	(839, 12, 32)	(842, 5, 6)	(848, -, 3)
(849, 8, 22)	(857, 8, 8)	(861, 28, 6)	(868, 19, 37)	(871, 6, 18)	(874, 9, 10)
(877, 16, 40)	(879, 2, 2)	(881, 18, 30)	(883, 4, 10)	(890, 5, 9)	(891, 2, 2)
(904, -, 11)	(908, 21, 7)	(911, 2, 2)	(916, 3, 4)	(917, 6, 6)	(923, 2, 2)
(929, 8, 14)	(931, 10, 16)	(935, 2, 2)	(939, 2, 2)	(941, 6, 8)	(943, 6, 10)
(955, 10, 18)	(958, 6, 5)	(967, 16, 24)	(972, 5, 4)	(975, 2, 2)	(986, 2, 6)
(992, -, 7)	(996, 43, 3)	(999, 8, 8)	(1000, -, 5)		

References

[1] C.K. Koc and B. Sunar, "Low-Complexity Bit-Parallel Canonical and Normal Basis Multipliers for a Class of Finite Fields," *IEEE Trans. Computers*, vol. 47, no. 3, pp. 353-356, Mar. 1998.

[2] C.K. Koc and B. Sunar, "An Efficient Optimal Normal Basis Type II Multiplier," *IEEE Trans. Computers*, vol. 50, no. 1, pp. 83-87, Jan. 2001.

[3] A. Reyhani-Masoleh and M.A. Hasan, "A New Construction of Massey-Omura Parallel Multiplier over $GF(2^m)$," *IEEE Trans. Computers*, vol. 51, no. 5, pp. 511-520, May 2002.

[4] A. Reyhani-Masoleh and M.A. Hasan, "Fast Normal Basis Multiplication Using General Purpose Processors," *IEEE Trans. Computers*, vol. 52, no. 11, pp. 1379-1390, Nov. 2003.

[5] A. Reyhani-Masoleh and M.A. Hasan, "Fast Normal Basis Multiplication Using General Purpose Processors," *Technical Report CORR 2001-25, Dept. of C&O, University of Waterloo, Canada*, Apr. 19, 2001.

[6] C. Koc and T. Acer, "Montgomery Multiplication in $GF(2^k)$," *Design,Codes and Cryptography*, vol. 14, no.1, pp. 57-69, Apr. 1998.

[7] R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R.M. Wilson, "Optimal Normal Bases in $GF(p^n)$," *Discrete Applied Mathematics*, vol. 22, pp. 149-161, 1988/89.

[8] S. Gao and Jr. H.W. Lenstra, "Optimal Normal Bases," *Design, Codes and Cryptography*, vol. 2 pp. 315-323, 1992.

[9] M.A. Hasan, "Look-Up Table-Based Large Finite Field Multiplication in Memory Constrained Cryptosystems," *IEEE Trans. Computers*, vol. 49, no. 7, pp. 749-758, July 2000.

[10] J. Brillhart, D. H. Lehmer J. L. Selfridge, B. Tuckerman, and S. S. Wagstaff, Jr., "Factorizations of $b^n \pm 1$, b= 2, 3, 5, 6, 7,10, 11, 12 up to High Powers," third ed. *Contemporary Mathematics*, vol. 22, American Mathematical Society, Providence, Rhode Island, Available at http://www.ams.org.

[11] H.N. Fan, "Simple Multiplication Algorithm for a Class of $GF(2^n)$," *Electronics Letters*, vol. 32, no.7, pp. 636-637, 1996.

[12] IEEE P1363-2000. "Standard Specifications for Public Key Cryptography," Aug. 2000.

[13] P. Ning and Y.L. Yin, "Efficient Software Implementation for Finite Field Multiplication in

Normal Basis," In *LNCS 2229 as Proceedings of 3rd International Conference on Information and Communications Security* (*ICICS*) *2001*, pp.177-188, Xian, China, 2001. Springer Verlag.

[14] W.W. Peterson and E.J. Weldon, Jr., *Error-Correcting Codes*. second ed. Cambridge, Mass.: The M.I.T. Press, 1972.

[15] E.R.Berlekamp, *Algebraic Coding Theory*. revised ed. Laguna Hills, CA: Aegean Park Press, 1984.

[16] O. Moreno, "On Primitive Elements of Trace Equal to 1 in $GF(2^n)$," *Discrete Mathematics*, vol. 41, pp. 53-56, 1982.

[17] D.Y. Pei, C.C. Wang and J.K. Omura, "Normal Basis of Finite Field $GF(2^m)$," *IEEE Trans. Information Theory*, vol. 32, no. 2, pp. 285-287, 1986.

[18] G. Seroussi, "Table of Low-Weight Binary Irreducible Polynomials," *Technical Report HPL*-98-135, *Hewlett-Packard Laboratories,* Palo Alto, Calif., Aug. 1998, Available at http://www.hpl.hp.com/techreports/98/HPL-98-135.html.

[19] S.H. Gao, J.V.Z.Gathen and D. Panario, "Gauss Periods: Orders and Cryptographical Applications," *Mathematics of Computation*, vol. 67, no. 221 pp. 343-352, 1998.

[20] C. Paar, P. Fleischmann and P. Roelse, "Efficient Multiplier Architectures for Galois Fields $GF(2^{4m})$," *IEEE Trans. Computers*, vol. 47, no. 2, pp. 162-170, Feb. 1998.

[21] Chung-Chin Lu, "A Search of Minimal Key Functions for Normal Basis Multipliers," *IEEE Trans. Computers,* vol. 46, no. 5, pp. 588-592, May 1997.

[22] H.N. Fan and Y.Q. Dai, "Key Function of Normal Basis Multipliers in $GF(2^n)$," *Electronics Letters*, vol. 38, no.23, pp. 1431-1432, 2002.

[23] A. Reyhani-Masoleh and M.A. Hasan, "Efficient Multiplication Beyond Optimal Normal Bases," *IEEE Trans. Computers*, vol. 52, no. 4, pp. 428-439, Apr. 2003.

[24] M. Maurer, A. Menezes and E. Teske, "Analysis of the GHS Weil Descent Attack on the ECDLP over Characteristic Two Finite Fields of Composite Degree," *Technical Report CORR 2001-59, Dept. of C&O, University of Waterloo, Canada,* Oct. 12, 2001.

[25] G.D. Birkhoff and H.S. Vandiver, "On the Integral Divisors of a^n-b^n," *Annals of Mathematics*, vol. 5, no.2, pp. 173-180, 1904.