Look-up Table Based Large Finite Field Multiplication in Memory Constrained Cryptosystems

M. A. Hasan

Abstract— **Many cryptographic systems use multiplication in the finite field GF(2^n) for their underlying computations. In the recent past, a number of look-up table based algorithms have been proposed for the software implementation of GF(2^n) multiplication. Look-up table based algorithms can provide speed advantages, but they either require a large memory space or do not fully utilize the resources of the processor on which the software is executed. In this work, an algorithm for GF(2^n) multiplication is proposed which can alleviate this problem. In each iteration of the proposed algorithm, a group of bits of one of the input operands are examined and two look-up tables are accessed. The group size determines the table sizes but does not affect the utilization of the processor resources. It can be used for both software and hardware realizations and is particularly suitable for implementations in memory constrained environment, such as, smart cards and embedded cryptosystems.**

Keywords— **Computer arithmetic, Galois (or finite) field multiplication, cryptographic systems, polynomial basis and look-up tables.**

I. Introduction

THE finite field GF(2^n) of characteristic two is used in many cryptosystems and has been included in the cryptographic standards of ANSI and IEEE. Computations in GF(2^n) are different from their counterparts in modular arithmetic, but they can potentially provide advantages for their implementations in resource constrained systems, such as, smart cards. In GF(2^n), addition and subtraction are the same and can be as simple as an XOR instruction of a general purpose processor. However, GF(2^n) multiplication is not so simple; nevertheless, there are cryptographic functions where a large number of multiplications are needed [2]. For example, using the Diffie-Hellman key exchange protocol on an elliptic curve defined over GF(2^{191}) as specified in ANSI X9.62, a single key establishment session may require about one thousand GF(2^{191}) multiplications. Thus, there is a need to develop efficient GF(2^n) multiplication algorithms which have lower computation time when implemented using available technologies.

In the literature, a number of GF(2^n) multiplication algorithms which rely on look-up tables have been proposed, for example, [3], [4], [5], [6]. The use of the tables, which are usually precomputed, reduces the number of operations needed during the execution time and consequently, reduces the effective computation time for multiplication. The sizes of these tables can be $2g2^g$ bits or more, where g corresponds to the number of the operand-bits examined at a time[1]. To take full advantage of these algorithms, the value of g needs to be equal to the width of the internal datapath (i.e., registers and arithmetic and logic units) of the processor in which the algorithms are implemented. These days, processors with 32 bit wide datapath can be found in many general purpose computing systems and the corresponding look-up table size would be 2^5 Gigabytes for [3], [4], [5] and 2^{37} Gigabytes for [6]. Such large memories of high speeds which can be used for look-up tables are however still not a commonplace, and consequently lower values of g (*e.g.*, 8) have been suggested. This, in turn, reduces the utilization of the processor's resources available. For example, when $g = 8$, only 8 bits out of the 32 bits of the ALU and bus of the processor are used in the arithmetic operations.

In this article a look-up table based algorithm for GF(2^n) multiplication is presented which can alleviate the above mentioned problem. For the multiplication of a and b in GF(2^n), this algorithm considers a group of g bits of b at a time and provides the product $a \cdot b$ in about n/g iterations. The algorithm is different from the other look-up table based multiplication algorithms in two important ways. First, it utilizes the full width of the datapath of the processor in which the algorithm is implemented; secondly, it uses two tables– one of which is precomputed during the field initialization process and the other is computed during the run (or, execution) time of the multiplication operation. Because of the run time generation of one table, it directly affects the multiplication operation and a mechanism is needed to quickly determine the table entries. Towards this end, a complementary algorithm for efficiently generating the table is presented, and this has enabled the multiplication algorithm to be implemented in memory constrained computing systems with a lower computation time.

The organization of this article is as follows. A brief discussion on the representation of the field elements and the basic field multiplication operation using the polynomial basis is given in Section II. A brief description of the conventional bit-level algorithm for GF(2^n) multiplication is also given in this section. Then the new look-up table based multiplication algorithm is presented in Section III.

Appeared in *IEEE Trans. Computers,* pp.749-758, July 2000. A preliminary version of this article was presented at the 7th IMA Conference on Cryptography and Coding [1].

The author is with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada. Email: ahasan@ece.uwaterloo.ca.

[1] For a *digit* serial/parallel multiplication algorithm, which groups g bits into a digit but does not apply look-up tables, the reader is referred to [7].

An efficient way to generate the look-up tables with fewer computations is developed in Section IV. Then Section V provides a numerical example of the overall operation of a multiplication operation. A comparison of the proposed algorithm with similar others is given in Section VI. A hardware architecture that can reduce the multiplication time using the look-up table based algorithm is presented in Section VII. Finally, concluding remarks are made in Section VIII.

II. Preliminaries

A. Field Element Representation

The finite field GF(2^n) has 2^n elements where n is a non-zero positive integer. Depending on the applications, the value of n can vary over a wide range. In cryptosystems, it can be as large as 1024 or more. Each of the 2^n elements of GF(2^n) can be uniquely represented with a polynomial of degree up to $n-1$ with coefficients from GF(2). For example, if a is an element in GF(2^n), then one can have

$$a \triangleq A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0. \quad (1)$$

This type of representation of the field elements is referred to as the *polynomial* or *standard basis* representation and has been used in many implementations.

For the polynomial basis representation as shown in (1), the addition of two field elements of GF(2^n) is simply bit-wise XOR operation of the coefficients of the equal powers of x, that is, if a and b are in GF(2^n), then

$$a + b \quad = \quad A(x) + B(x) = \sum_{i=0}^{n-1} (a_i + b_i)x^i$$

where the addition in the parenthesis indicates an XOR or modulo 2 addition operation. On the other hand, the multiplication of the field elements using the polynomial basis representation is much more complicated. It can be performed by first multiplying $A(x)$ with $B(x)$ and then taking modulo $F(x)$ on $A(x)B(x)$, *i.e.*, if p is the product of a and b then

$$p \triangleq P(x) = A(x)B(x) \bmod F(x). \quad (2)$$

In (2), $F(x)$ is a polynomial over GF(2) of degree n which defines the representation of the field elements. Such $F(x)$ has to be an irreducible polynomial which has the following form:

$$F(x) = x^n + f_{n-1}x^{n-1} + f_{n-2}x^{n-2} + \cdots + f_1 x + 1 \quad (3)$$

where the f_i's belong to $\{0, 1\}$. The choice of $F(x)$ can play an important role in determining the performance of the implementation of finite field multipliers. For example, using irreducible *trinomials* which have only three non-zero coefficients, several researchers have proposed multipliers which provide advantages in terms of both speed and space [8], [9]. Examples of other special forms of $F(x)$ include *all-one* polynomials and *equally-spaced* polynomials [10], [11].

B. Bit-Level Multiplication Algorithm & Its Complexity

Let $F(x)$ be the irreducible polynomial defining the representation of GF(2^n), and a, b, and p be any three elements of GF(2^n) such that $p = a \cdot b$ as defined earlier. Then

$$\begin{aligned} P(x) &= A(x)B(x) \bmod F(x) \\ &= A(x)\left(b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \right. \\ &\quad \left. \cdots + b_1 x + b_0\right) \bmod F(x) \\ &= ((\cdots(A(x)b_{n-1}x + A(x)b_{n-2})x + \\ &\quad \cdots)x + A(x)b_1)x + A(x)b_0 \bmod F(x) \\ &= ((\cdots(A(x)b_{n-1}x \bmod F(x) \\ &\quad + A(x)b_{n-2})x \bmod F(x) + \cdots)x \bmod F(x) \\ &\quad + A(x)b_1)x \bmod F(x) + A(x)b_0. \end{aligned} \quad (4)$$

In (4), the repeated operations involve multiplying $A(x)$ with x depending on the coefficients of $B(x)$ and then taking mod $F(x)$. These operations can be specified in terms of an algorithm as follows:

Algorithm 1: (Bit-Level Algorithm for GF(2^n) Multiplication)
Input: $A(x)$, $B(x)$ *and* $F(x)$
Output: $P(x) = A(x)B(x)$ *mod* $F(x)$
Step 1.1. If $b_{n-1} = 1$, $P(x) := A(x)$
else $P(x) := 0$
Step 1.2. For $i = n-2$ *to* 0 {
$P(x) := xP(x)$ *mod* $F(x)$
If $b_i = 1$, $P(x) := P(x) + A(x)$
} □

After the final iteration, $P(x)$ is the polynomial basis representation of the required product $a \cdot b$. In Algorithm 1, a coefficient, *i.e.*, a single bit of $B(x)$ is checked in each iteration. The loop in Step 1.2 is executed $n-1$ times. Its operation $P(x) := xP(x) \bmod F(x)$ can be realized by left shifting the coefficients of $P(x)$ to obtain $xP(x)$, and then subtracting/adding $F(x)$ from it if the nth coefficient of $xP(x)$ is 1. Assuming that 0 and 1 appear as the coefficients of $P(x)$ with an equal probability, the operation $P(x) := xP(x) \bmod F(x)$ takes $\frac{n-1}{2}$ polynomial additions on average. Similarly, there are $\frac{n-1}{2}$ more polynomial additions on average for the other operation $P(x) := P(x) + A(x)$. Thus, in Algorithm 1, there are approximately $n-1$ polynomial additions on average. If we assume that the time to add two polynomials is T_{poly_add} and that to multiply a polynomial by x (i.e., to shift the coefficients) is T_{poly_shift}, then the average computation time of a GF(2^n) multiplication can be approximated as:

$$T_{multi\_in\_GF(2^n)} \approx (n-1) \cdot (T_{poly\_add} + T_{poly\_shift}). \quad (5)$$

For a cryptosystem which uses a large value of n, the above bit-level algorithm may not be the best solution especially when speed is of major concern.

C. Multi-Precision Arithmetic and Look-up Tables

In Algorithm 1, the overall computation time of a GF(2^n) multiplication can be reduced by reducing both

the polynomial addition time and the total number of iterations. In order to reduce the polynomial addition time, especially when a general purpose processor is used, the coefficients of the polynomials can be divided into $\lceil \frac{n}{w} \rceil$ units each consisting of w bits, where the latter corresponds to the word size of the processor. For example, on a 32 bit processor, six units each of 32 bits are needed for the polynomial basis representation of an element in GF(2^{191}). Assuming that the processor can perform bit-wise exclusive-or of two 32-bit operands using one single XOR instruction, the addition of two polynomials of degree up to 190 will take six such XOR instructions. In general, the number of XOR instructions needed for adding two polynomials of degree up to $n-1$ with coefficients from GF(2) is

$$\#XOR_{poly\_add} = \left\lceil \frac{n}{w} \right\rceil . \tag{6}$$

If T_{XOR} is the time needed for executing an XOR instruction, then

$$T_{poly\_add} = \#XOR_{poly\_add} \cdot T_{XOR} = \left\lceil \frac{n}{w} \right\rceil T_{XOR}. \tag{7}$$

Using such multi-precision arithmetic, a number of GF(2^n) multiplication algorithms that can be implemented in a general purpose processor have been proposed [3], [4], [6]. These algorithms are based on look-up tables of up to $g2^{2g}$ bits which are basically used to multiply two polynomials of degree up to $g-1$ [6]. The processor's resources are best utilized when g is equal to w (or, a multiple of w). Thus, for $g = w$ and for a 32 bit processor which has become a commonplace these days, the table size is 2^{36} Gigabytes which is too large to be implemented in any practical system with today's memory technologies. One solution is to use a smaller value of g which in turn would reduce the table size. This however may cause a lower utilization of the processor's resources, especially a significant portion of the processor's datapath may remain unused with each XOR instruction.

What follows below is a GF(2^n) multiplication algorithm which attempts to better utilize the processor's resources even when a smaller group size is used. Additionally, unlike the algorithms of [3] and [4] which have the limitation that n be multiple of the group size, the algorithm can be used for any n. Thus, this algorithm could be useful when a prime n is sought, for example, in the implementation of high speed elliptic curve cryptosystems using Koblitz curves [12].

III. Group-Level Look-up Table Based Multiplication

Let us divide the n coefficient bits of $B(x)$ into s groups of $g \geq 2$ bits each[2]. If n is not a multiple of g, then the number of bits in the most significant group is taken as $n \bmod g$. Thus,

$$\begin{aligned} B(x) &= x^{(s-1)g}B_{s-1}(x) + x^{(s-2)g}B_{s-2}(x) + \cdots + \\ &\quad x^g B_1(x) + B_0(x) \end{aligned}$$

where

$$B_i(x) = \begin{cases} \sum_{j=0}^{g-1} b_{ig+j}x^j & 0 \leq i \leq s-2, \\ \sum_{j=0}^{(n \bmod g)-1} b_{ig+j}x^j & i = s-1. \end{cases} \tag{8}$$

Then

$$\begin{aligned} P(x) &= A(x)B(x) \bmod F(x) \\ &= A(x)\left(x^{(s-1)g}B_{s-1}(x) + x^{(s-2)g}B_{s-2}(x) + \cdots + \right. \\ &\quad \left. x^g B_1(x) + B_0(x)\right) \bmod F(x) \\ &= (\cdots(\ (\ A(x)B_{s-1}(x) \bmod F(x)\ )\, x^g \bmod F(x) \\ &\quad + A(x)B_{s-2}(x) \bmod F(x)\ )\, x^g \bmod F(x) \\ &\quad \vdots \\ &\quad + A(x)B_1(x) \bmod F(x)\ )\, x^g \bmod F(x) \\ &\quad + A(x)B_0(x) \bmod F(x). \end{aligned} \tag{9}$$

Based on (9), now we have the following group-level intermediate algorithm for GF(2^n) multiplication.

Algorithm 2: (Group-Level GF(2^n) Multiplication)
Input: $A(x)$, $B(x)$ *and* $F(x)$
Output: $P(x) = A(x)B(x)$ *mod* $F(x)$
Step 2.1. $P(x) := B_{s-1}(x)A(x)$ *mod* $F(x)$
Step 2.2. For $k = s-2$ *to* 0 {
$P(x) := x^g P(x)$ *mod* $F(x)$
$P(x) := P(x) + B_k(x)A(x)$ *mod* $F(x)$
} □

After the final iteration of the above algorithm, the coefficients of $P(x)$ correspond to the polynomial basis representation of the product $a \cdot b$. The loop in Step 2.2 is executed $s-1$ times. Since $P(x) \triangleq \sum_{i=0}^{n-1} p_i x^i$ is a polynomial of degree up to $n-1$ with coefficients from GF(2), the first operation of Step 2.2 can be written as follows:

$$P(x) := \underbrace{x^g \sum_{i=0}^{n-1-g} p_i x^i}_{\tau_1} + \underbrace{x^g \sum_{i=n-g}^{n-1} p_i x^i \bmod F(x)}_{\tau_2} . \tag{10}$$

In (10), τ_1 is a g-fold left shift of the least significant $n-g$ coefficients of $P(x)$ and the other term τ_2 depends on the g most significant bits of $P(x)$ as well as the coefficients of $F(x)$. In practice, $F(x)$ does not change in a single cryptographic session, and in many cases, it remains unchanged as long as the dimension of the field does not change. In such circumstances, a table which is hereafter referred to as the M (or, modulo) table can be created to store $x^g \sum_{i=n-g}^{n-1} p_i x^i \bmod F(x)$. The table entries are precomputed as part of the field initialization process.

[2] Unlike [3], [4], [6], the group size g in this work is expected to have a much smaller value. For the convenience of implementation, the value of g should divide w.

In a straightforward realization, the M table may have 2^g entries each with n bits resulting in a memory requirement of $n2^g$ bits. If $F(x) = x^n + x^d + \sum_{i=1}^{d-1} f_i x^i + 1$, with $f_i \in \{0,1\}$ for $0 < i < d$, *i.e.*, the degree of the second leading coefficient of $F(x)$ is d, then the effective size of each table entry is $d+g$ bits. Thus, an irreducible polynomial with a smaller value of d results in a smaller look-up table. An example with $F(x) = x^{191} + x^9 + 1$ resulting in a table width of 16 bits is shown in Table I where the polynomials are shown in hexadecimal notation, *i.e.*, 0201 of entry 1 corresponds to $x^9 + 1$. Also, examples of trinomials and pentanomials with small d which will result in a look-up table of width 16 bits or less for $g = 4$ are given in Table II. The table lists only polynomials with degree n where n is a prime in the range $[100, 512]$. This range covers most of the current applications that may potentially incorporate elliptic curve cryptosystems.

TABLE I

An example of table M with $F(x) = x^{191} + x^9 + 1$ and $g = 4$.

0	$x^n \cdot 0 \bmod F(x)$	$= 0000$
1	$x^n \cdot 1 \bmod F(x)$	$= 0201$
2	$x^n \cdot x \bmod F(x)$	$= 0402$
3	$x^n \cdot (x+1) \bmod F(x)$	$= 0603$
4	$x^n \cdot x^2 \bmod F(x)$	$= 0804$
5	$x^n \cdot (x^2+1) \bmod F(x)$	$= 0a05$
6	$x^n \cdot (x^2+x) \bmod F(x)$	$= 0c06$
7	$x^n \cdot (x^2+x+1) \bmod F(x)$	$= 0e07$
8	$x^n \cdot x^3 \bmod F(x)$	$= 1008$
9	$x^n \cdot (x^3+1) \bmod F(x)$	$= 1209$
10	$x^n \cdot (x^3+x) \bmod F(x)$	$= 140a$
11	$x^n \cdot (x^3+x+1) \bmod F(x)$	$= 160b$
12	$x^n \cdot (x^3+x^2) \bmod F(x)$	$= 180c$
13	$x^n \cdot (x^3+x^2+1) \bmod F(x)$	$= 1a0d$
14	$x^n \cdot (x^3+x^2+x) \bmod F(x)$	$= 1c0e$
15	$x^n \cdot (x^3+x^2+x+1) \bmod F(x)$	$= 1e0f$

TABLE II

Examples of irreducible polynomials which will require a look-up table of width of 16 bits or less for $g = 4$. For each listed polynomial, the table gives the corresponding non-zero coefficients excluding the constant term which is always 1.

101,7,6,1	103, 9	107,9,7,4	109,5,4,2
113,9	127,1	131,8,3,2	139,8,5,3
149,10,9,7	151,3	157,6,5,2	163,7,6,3
167,6	173,8,5,2	179,4,2,1	181,7,6,1
191,9	197,9,4,2	211,11,10,8	227,10,9, 4
229,10,4,1	251,7,4,2	269,7,6,1	293,11,6,1
307,8,4,2	311,7,5,3	317,7,4,2	331,10,6,2
347,11,10,3	349,6,5,2	373,8,7,2	379,10,8,5
389,10,9,5	421,5,4,2	443,10,6,1	461,7,6,1
467,11,6,1	491,11,6,1	499,11,6,5	503,3
509,8,7,3			

Referring to the second operation $P(x) := P(x) + B_k(x)A(x) \bmod F(x)$ in Step 2.2, let

$$\tau_3 = B_k(x)A(x) \bmod F(x). \tag{11}$$

The product $B_k(x)A(x)$ results in a polynomial of degree $\leq n-g-2$. In order to reduce the degrees of x^{n+g-2}, x^{n+g-3}, $\cdots$, x^n of $B_k(x)A(x)$, we need polynomial shifts and additions. For reasonable values of g, the term τ_3 can however be directly read from a precomputed look-up table (hereafter referred to as T) and thus the above shift and addition operations can be avoided. Since in practice $n >> g$, the table can be more conveniently built to store $B_k(x)A(x) \bmod F(x)$ for all possible $B_k(x)$'s. The size of the table would then be $n2^g$ bits. However, unlike the previous M table, this table needs to be created on the fly each time a new $A(x)$ is chosen, and care must be taken to reduce the task to compute the table entries as it lies in the critical path of the loop in Algorithm 2.

Algorithms for generating the tables are given in Section IV. We wind up this section by incorporating the M and T tables into Algorithm 2. In this regard let $e \triangleq \sum_{i=0}^{g-1} e_i 2^i$ be an integer in the range $[0, 2^g - 1]$ and let the contents of the e-th entry of the M and T tables be

$$M[e] = \left(\sum_{i=0}^{g-1} e_i x^i\right) x^n \bmod F(x), \text{ and}$$

$$T[e] = \left(\sum_{i=0}^{g-1} e_i x^i\right) A(x) \bmod F(x),$$

respectively, then we have the following algorithm where $P_{s-1}(x) \triangleq \sum_{i=0}^{g-1} p_{n-g+i} x^i$.

Algorithm 3: (Look-up Table Based Group-Level Multiplication)
Input: $A(x)$, $B(x)$, $F(x)$, and the M table
Output: $P(x) = A(x)B(x)$ mod $F(x)$
Step 3.1. Generate table T
Step 3.2. $P(x) := T[B_{s-1}(x=2)]$
Step 3.3. For $k = s-2$ to 0 {
$\tau_1 := x^g \sum_{i=0}^{n-1-g} p_i x^i$
$\tau_2 := M[P_{s-1}(x=2)]$
$\tau_3 := T[B_k(x=2)]$
$P(x) := \tau_1 + \tau_2 + \tau_3$
} □

Note that steps 3.2 and 3.3 of Algorithm 3 correspond to steps 2.1 and 2.2 of Algorithm 2. In Algorithm 3, the numbers of n-bit word read from the T and M tables are s and $s-1$, respectively. The algorithm requires $2(s-1)$ polynomial additions, or $2(s-1)\lceil \frac{n}{w} \rceil$ XOR instructions. For τ_1, one needs $(s-1)\lceil \frac{n}{w} \rceil$ SHIFT instructions. For the evaluation of the indices of the two tables, one can use $2s-1$ SHIFT and $2s-1$ AND instructions. The cost of computing $T[B_i(x=2)]$, for $i = s-1, \cdots, 1, 0$, is given in the following section.

IV. Table Generation Algorithms

In this section, we consider algorithms for generating the T table. This table needs to be created with minimum pos-

sible delay to make its use feasible in the GF(2^n) multiplication operation. This algorithm is equally applicable to M where the speed of the table generation is however not that critical.

There are g bits in $B_k(x)$ and table T has 2^g entries. The e-th entry of T is

$$T[e] = e(x) \cdot A(x) \bmod F(x) \tag{12}$$

where $e(x) \triangleq \sum_{i=0}^{g-1} e_i x^i$ as assumed earlier. Thus, once the table has been generated, $B_k(x)A(x) \bmod F(x)$ is obtained by reading the table entry at

$$B_k(x=2) = \sum_{j=0}^{g-1} b_{kg+j} 2^j.$$

In the sequel, the following g entries of the table, namely, $T[1], T[2], T[2^2], \cdots$, and $T[2^{g-1}]$ are referred to as the *base* entries. From (12), the j-th base entry is

$$T[2^j] = x^j A(x) \bmod F(x),$$

from which one can write

$$T[2^{j+1}] = xT[2^j] \bmod F(x). \tag{13}$$

Thus, given the j-th base entry, the computation of the $(j+1)$-st base entry takes a maximum of one polynomial addition[3]. Thus, if the first base entry $T[2^0]$ is initialized with $A(x)$, then the maximum number of XOR instructions needed to compute the remaining $g-1$ base entries is $(g-1)\lceil \frac{n}{w} \rceil$, and the corresponding average number is $\frac{1}{2}(g-1)\lceil \frac{n}{w} \rceil$.

Lemma 1: If $A(x) \neq 0$, then all the entries except $T[0]$ contain non-zero polynomials of degree up to $n-1$.

Proof: From (12), $T[0]$=0 and $T[1] = A(x)$. Each of the subsequent entry contains the mod $F(x)$ of the product $A(x) \cdot e(x)$, where both $A(x) \neq 0$ and $e(x) \neq 0$. Thus the entry corresponds to the product of two non-zero elements of GF(2^n) which is also a non-zero element in the field. Thus, the lemma holds. ■

In order to compute the *regular* (i.e., non-zero and non-base) entries of the table, below we present two schemes. Both assume that the g base entries have already been computed and use these entries to compute the regular entries of the table.

A. Entry Computation on Demand

When $s < 2^g$, only a part of the table is read when Algorithm 3 is executed. In such cases, instead of computing all the regular entries and storing them, it is advantageous to compute the entries as they are needed. In this effect, the following algorithm can be used.

Algorithm 4: (Regular Entry Computation on Demand)
Input: Index e, and base entries $T[2^i], i = 0, 1, \cdots, g-1$
Output: The e-th entry $T[e]$
Step 4.1. If $e_0 = 0$, $tmp := 0$
else $tmp := T[1]$
Step 4.2. For 1 to $g-1$ {
If e_i=1, $tmp := tmp + T[2^i]$
}
Step 4.3. $T[e] := tmp$ □

The number of polynomial additions needed in Algorithm 4, depends on the Hamming weight of the binary representation of e. On average, the algorithm requires $(g-1)/2$ polynomial additions. Thus, the cost of computing $T[B_i(x=2)]$, $0 \leq i \leq s-1$, in Algorithm 3 is approximated as $s(g-1)/2$ polynomial additions. The cost of creating all the regular entries of T, which will be used in our forthcoming discussions, is given below.

Corollary 1: The generation of all the regular entries of the T table using Algorithm 4 requires $2^{g-1}(g-2)+1$ polynomial additions.

Proof: The binary representation of all the 2^g indices have $g2^{g-1}$ ones; thus the total number of 1's in the indices of all regular entries is $g(2^{g-1}-1)$. Each index of the regular entries has two or more 1's, and an index with j ones requires $j-1$ polynomial additions. Since the total number of regular entries is $2^g-(g+1)$, the total number of polynomial additions needed to compute all the regular entries is

$$g(2^{g-1}-1) - (2^g - (g+1)) = 2^{g-1}(g-2)+1 \tag{14}$$

■

B. Entry Computation in Window Sequence

We now consider the case of $s \geq 2^g$ where each table entry is expected to be accessed at least once. In this case, one can create all the table entries. In this regard, the $2^g - g - 1$ regular entries are partitioned into $g-1$ windows, namely, $W_1, W_2, \cdots, W_{g-1}$, where window W_i, $1 \leq i \leq g-1$, consists of the following $2^i - 1$ entries $T[2^i+1]$, $T[2^i+2]$, $\cdots$, $T[2^i+2^i-1]$.

Lemma 2: For window W_i,

$$T[2^i + j] = T[2^i] + T[j], \quad 1 \leq j \leq 2^i - 1. \quad \square \tag{15}$$

Proof: In (15), j is an integer in the range $[1, 2^i-1]$. Thus j can be represented in the binary form with i bits, i.e., $j = \sum_{l=0}^{i-1} j_l 2^l$. Thus,

$$\begin{aligned} T[j] &= A(x)\left(j_{i-1}x^{i-1} + j_{i-2}x^{i-2} + \right. \\ &\quad \left. \cdots + j_0\right) \bmod F(x) \\ T[2^i] + T[j] &= A(x)\left(x^i + j_{i-1}x^{i-1} + j_{i-2}x^{i-2} + \right. \\ &\quad \left. \cdots + j_0\right) \bmod F(x) \\ &= T[2^i + j] \quad \text{Q. E. D.} \end{aligned}$$

Given the entries of windows W_j, $1 \leq j \leq i-1$, the base entries $T[2^j]$, $0 \leq j \leq i$, one can compute an entry of W_i using only one polynomial addition. The entries are computed in the window sequence, *i.e.*, the entries of W_i are computed only after the entries of W_{i-1} have been computed. The entries within W_i can however be computed in

[3] This is because a multiplication by x in (13) may result in a polynomial of degree n that needs to be reduced modulo $F(x)$.

any order. A systemic way to obtain all the regular entries using the window-by-window updating scheme is given below.

Algorithm 5: (Computation of Regular Entries in Window Sequence)
Input: Base entries $T[2^i], i = 0,\ 1,\ \cdots,\ g-1$
Output: All regular entries of T

```
For i = 1 to g − 1   {
    For j = 1 to 2^{i−1} − 1   {
        T[2^i + j] := T[2^i] + T[j]
    }
}  □
```

In Algorithm 5, the loop with index i corresponds to the window computing. Since there are 2^i-1 entries in window W_i, the cost for generating entries of all the $g-1$ windows is

$$(2^1-1) + (2^2-2) + \cdots + (2^{g-1}-1) = 2(2^{g-1}-1)-g+1 \quad (16)$$

polynomial additions. The ratio of (14) to (16) gives us an estimate of the speed-up that one can expect in computing the regular entries of table T. This ratio is a function of g and is equal to 1, 1.55, 2.26 and 3.11 for $g = 2,\ 4,\ 6,$ and 8, respectively.

V. An Example

In this section we apply the proposed algorithms to an example multiplication operation in the field GF(2^8). In this regard, let $F(x)$ be $x^8+x^4+x^3+x+1$ and w be 6. Thus, two words are needed to represent each field element in multi-precision machine representation. Let us assume that $g = 3$. Then the M table[4] will have eight entries as shown in Table III.

TABLE III
The M and T tables with $F(x) = x^8+x^4+x^3+x+1$, $A(x) = x^7+x^4+1$ and $g = 3$. Table contents are given in the binary form.

	The M Table
0	0000 0000
1	0001 1011
2	0011 0110
3	0010 1101
4	0110 1100
5	0111 0111
6	0101 1010
7	0100 0001

	The T Table
0	0000 0000
1	1001 0001
2	0011 1001
3	1010 1000
4	0111 0010
5	1110 0011
6	0100 1011
7	1101 1010

Suppose that $A(x) = x^7+x^4+1$ and $B(x) = x^6+x^5+x$. Thus,

$$B_0(x) = x = (010)_2,$$
$$B_1(x) = x^2 = (100)_2,$$
$$B_2(x) = 1 = (\ 01)_2,$$

where $(\cdot)_2$ indicates the binary form of the corresponding polynomial. Now we can apply Algorithm 3 to compute $A(x)B(x) \bmod F(x)$ as follows:
Step 3.1:

Using Algorithm 5, the T table is generated which is given in Table III.
Step 3.2:

$$P(x) := T[B_2(x=2)] = T[1] = (\underbrace{100}_{P_2}1\ 0001)_2$$

Step 3.3: (Below "$x << y$" indicates a left shift of x by y bits.)
$i = 1$

$$\begin{aligned}
\tau_1 &:= (1\ 0001)_2 << 3 &&= (1000\ 1000)_2\\
\tau_2 &:= M[P_2] = M[4] &&= (0110\ 1100)_2\\
\tau_3 &:= T[B_1] = T[4] &&= (0111\ 0010)_2\\
P(x) &:= \tau_1+\tau_2+\tau_3 &&= (\underbrace{100}_{P_2}1\ 0110)_2
\end{aligned}$$

$i = 0$

$$\begin{aligned}
\tau_1 &:= (1\ 0110)_2 << 3 &&= (1011\ 0000)_2\\
\tau_2 &:= M[P_2] = M[4] &&= (0110\ 1100)_2\\
\tau_3 &:= T[B_0] = T[4] &&= (0011\ 1001)_2\\
P(x) &:= \tau_1+\tau_2+\tau_3 &&= (1110\ 0101)_2
\end{aligned}$$

Thus the final product is $P(x) = x^7+x^6+x^5+x^2+1$.

VI. Comparison

In this section, we compare a number of related multipliers which were reported in the open literature in the past. A comparison based on absolute computation time is difficult since it depends on a number of parameters including the computing platform used, the level of optimization applied, efficiency of the code, etc. As a result, we restrict our basis of comparison to the number of salient instructions and look-up table accesses usually needed to implement the multipliers.

Harper *et al.* in [3] have used two look-up tables for multiplication in GF(2^n). Win *et al.* in [4] have used a similar scheme. While Harper *et al.* have reported their results using the subfield GF(2^8), Win *et al.* have done it using GF(2^{16}). Here we present both the implementations in a generalized way. They assume a composite n which is a multiple of g and do underlying computations in the subfield GF(2^g) rather than GF(2). One of the look-up tables contains logarithms of the polynomial basis representation of the elements of GF(2^g). The other table contains the corresponding anti-logarithms. The size of the two tables together is approximately[5] $2g \cdot 2^g$ bits. For a' and b' in GF(2^g), the product

$$a' \cdot b' = \text{anti-log}[\ (\log[a'] + \log[b']) \bmod (2^g-1)] \quad (17)$$

[4] Since $s < 2^g$, we really do not need to create the M and T tables in this simple example. However, most practical applications have $s \geq 2^g$. So, these tables are used here only to demonstrate the concept.

[5] log[0] is undefined.

requires three table accesses, assuming that g is less than or equal to the memory bandwidth. As both $\log[a']$ and $\log[b']$ are g bit integers and lie in the range $[0, 2^g - 2]$, the addition $\log[a'] + \log[b']$ results in an integer which is greater than or equal to $2^g - 1$ in only $(2^g - 1)(2^g - 2)/2$ out of $(2^g - 1)^2$ cases. Thus, for non-zero a' and b', the mod operation in (17) is executed with a probability of $\frac{1}{2} - \frac{1}{2(2^g-1)}$.

Using the above look-up tables for the subfield multiplication, the remaining steps of GF(2^n) multiplication scheme of [3], [4] can be described as follows. For a, b, and p are in GF(2^n), and $a = \sum_{i=0}^{s-1} a'_i x^i$ and $b = \sum_{i=0}^{s-1} b'_i x^i$, where a'_i and b'_i's are in GF(2^g), initialize p to 0, then compute a, ax, ..., ax^{s-1}; after each stage add $b'_i(ax^i)$ mod the irreducible polynomial used to define GF(2^n) over the subfield GF(2^g). Thus, the complexity of a multiplication in GF(2^n) is approximately $2s^2$ XOR instructions, and s^2 GF(2^g) multiplications. The latter is equivalent to $3s^2$ table accesses, s^2 integer additions, and $s^2\left(\frac{1}{2} - \frac{1}{2(2^g-1)}\right)$ integer subtractions (to emulate mod $(2^g - 1)$ operations).

Another look-up table based scheme for GF(2^n) multiplication, where n is a multiple of g, has been presented in [5]. This scheme uses the same two look-up tables for logarithms and anti-logarithms as described above. However, for multiplication of polynomials over the subfield GF(2^g), it uses the Karatsuba-Ofman algorithm [13]. Assuming that s is a power of two, the overall multiplication scheme requires $6s^{\log_2 3} - 8s + 2$ XOR instructions, $s^{\log_2 3}$ ADD instructions and $3s^{\log_2 3}$ look-up table accesses.

In [6], extending the Montgomery algorithm for modular multiplication Koc and Acar have presented an algorithm for $A(x)B(x)x^{-n}$ mod $F(x)$. They have suggested a look-up table to store the product $A'(x)B'(x)$ where both $A'(x)$ and $B'(x)$ are polynomials over GF(2) of degree $g-1$ or less. The table has $2^{2g}(2g-1)$ bits and is accessed $2s^2+s$ times. The algorithm also requires $4s^2$ XOR instructions. The use of the algorithm may require certain computational overheads since the algorithm outputs $A(x)B(x)x^{-n}$ mod $F(x)$ rather than $A(x)B(x)$ mod $F(x)$.

With regard to Algorithm 3 presented in this work, if $s < 2^g$ then $T[B_i(x = 2)]$, $0 \leq i \leq s-1$, is computed using Algorithm 4 and otherwise using Algorithm 5. Referring to the analyzes of Algorithms 3, 4 and 5, one can determine the cost of a GF(2^n) multiplication operation, which is given in Table IV along with the costs of multiplication operations of [3] and [6]. In the table, d is the degree of the second leading coefficient of the field defining polynomial. In determining the number of look-up table accesses, we have assumed that the memory bandwidth is equal to the datapath width of w bits.

As it can seen from the above table, for large values of g (and hence small s) the multiplication scheme of [5] requires the least number of instructions. However, the corresponding look-up table is too large to be implemented in resource constrained cryptosystems. More importantly, the scheme of [5] as well as that of [3] cannot be applied to GF(2^n) with n being a prime, an important criteria that is sought to have in many cryptosystems employing such extension fields [12].

In order to be able to see the usefulness of the proposed algorithm in memory constrained cryptosystems where g ought to be small, one can apply the cryptographic parameters recommended by standard bodies, such as ANSI, to the above multipliers. In this regards, one can use $n = 113$ with $F(x) = x^{113} + x^9 + 1$ for smart cards, or $n = 191$ with $F(x) = x^{191} + x^9 + 1$ for applications where much higher level of security is required. Assuming 16 and 32 bit processors, Tables V and VI, respectively, compare the multiplication algorithms of [3], [6] and this work for two different values of g.

It can be seen from the above tables that in memory constrained cryptosystems where a small g (say, 4) has to be used, the proposed multiplication algorithm would provide computational advantages[6] over the similar other available algorithms. Additionally, unlike [3], [4] where a composite n is assumed, the proposed algorithm can be used for any n. Also, unlike [6] which outputs $A(x)B(x)x^{-n}$ mod $F(x)$, the new algorithm directly gives $A(x)B(x)$ mod $F(x)$ requiring no post-processing for a GF(2^n) multiplication.

VII. Hardware Architecture

In order to reduce the computation time, the multiplication scheme described in Algorithm 3 can be mapped on to a special-purpose hardware. Towards this end, a multiplier structure which uses n-bit bus is shown in Fig. 1. The structure mainly consists of two look-up tables (M and T), a 3-operand mod 2 adder and two n-bit registers (B and P). Since, the M table contents depend only on the the irreducible polynomial $F(x)$ which is fixed for many applications, the table can be implemented simply with a ROM. On the other hand, the T table contents change each time a new *multiplicand* (a) is used; hence the table needs to be implemented with a RAM. Before presenting a structure to initialize the T table, an informal description of the overall operation of the multiplier structure of Fig. 1 is given below.

In each iteration of a multiplication process, the T table is addressed by a group of g bits from the *multiplier* (b) stored in register B. The latter is left-shifted g positions in each iteration so that the correct bits are used to address T. The output from T is τ_3 which enters the 3-operand mod 2 adder. The other operands are τ_2 and τ_1. Operand τ_2 is obtained from the M table which is addressed by the "most-significant" g bits of the partial product stored in register P. Operand τ_1 is derived by a g-bit left-shift of the "least-significant" $n - g$ bits of the partial product. A new partial product is mod 2 addition of τ_1, τ_2 and τ_3. The alignment of these operands that corresponds to the correct number of shifts is shown inside the 3-operand adder. After $\lceil n/g \rceil$ iterations as described above, register P contains the

[6] For the proposed multiplication scheme, a further reduction in the number of XOR instructions can be obtained by noting that when $g + d + 1 < w$, only one XOR instruction (as opposed to $\lceil \frac{n}{w} \rceil$ XOR instructions) is needed to add the term X_2 in Step 3.3 of Algorithm 3.

TABLE IV

COMPARISON OF RELATED MULTIPLICATION ALGORITHMS.

	No. of important Instructions		Look-up table	
	XOR	ADD/SUB, SHIFT/AND	Size in bits	No. of accesses
Harper *et al.* [3]	$2s^2$	$s^2\left(\frac{3}{2}-\frac{1}{2(2^g-1)}\right)$	$2g2^g$	$3s^2$
Guajardo & Paar [5]	$6s^{\log_2 3}-8s+2$	$s^{\log_2 3}$	$2g2^g$	$3s^{\log_2 3}$
Koc & Acar [6]	$4s^2$	-	$(2g-1)2^{2g}$	$2s^2+s$
This work ($s<2^g$)	$(\frac{1}{2}(s+1)(g+3)-4)\lceil\frac{n}{w}\rceil$	$(s-1)\lceil\frac{n}{w}\rceil+2(2s-1)$	$(g+d)2^g$	$(s-1)\lceil\frac{g+d}{w}\rceil$
This work ($s\geq 2^g$)	$(2^g+2s-4-\frac{g-1}{2})\lceil\frac{n}{w}\rceil$	$(g+s-2)\lceil\frac{n}{w}\rceil+2(2s-1)$	$(g+d)2^g$ (M table) + $n2^g$ (T table)	$(s-1)\lceil\frac{g+d}{w}\rceil$ (M table) + $(2^{g+1}+s-g-2)\lceil\frac{n}{w}\rceil$ (T table)

TABLE V

COMPARISON OF MULTIPLICATION ALGORITHMS WITH $n=113$, $w=16$ AND $F(x)=x^{113}+x^9+1$.

	$n=113,\ w=16$							
	$g=4$, *i.e.*, $s=29>2^g$				$g=8$, *i.e.*, $s=15<2^g$			
	Instr.		Table		Instr.		Table	
	XOR	Others	Bytes	Accesses	XOR	Others	Bytes	Accesses
Harper *et al.* [3]	Not applicable since n is prime							
Guajardo & Paar [5]	Not applicable since n is prime							
Koc & Acar [6]	3,364	-	2^8	1,711	900	-	2^{17}	465
This work	548	362	2^5 (M table) + 2^8 (T table)	28 (M table) + 220 (T table)	672	170	3×2^8 (M table)	14 (M table)

TABLE VI

COMPARISON OF MULTIPLICATION ALGORITHMS WITH $n=191$, $w=32$ AND $F(x)=x^{191}+x^9+1$.

	$n=191,\ w=32$							
	$g=4$, *i.e.*, $s=48>2^g$				$g=8$, *i.e.*, $s=24<2^g$			
	Instr.		Table		Instr.		Table	
	XOR	Others	Bytes	Accesses	XOR	Others	Bytes	Accesses
Harper *et al.* [3]	Not applicable since n is prime							
Guajardo & Paar [5]	Not applicable since n is prime							
Koc & Acar [6]	9,216	-	2^8	4,656	2,304	-	2^{17}	1,176
This work	630	490	2^5 (M table) + 24×2^4 (T table)	47 (M table) + 444 (T table)	801	232	3×2^8 (M table)	23 (M table)

product of a and b. It should be noted that the 3-operand adder consists of $n+d$ two-input XOR gates and does not have any registers inside.

A hardware structure that can be used for updating the T table is shown in Fig. 2. The LFSR of the structure is an n-stage Fibonacci type linear feedback shift register whose feedback connection is defined by the irreducible polynomial $F(x)$. If the LFSR is loaded with $A(x)$ then after i left-shifts the LFSR contains $x^iA(x)$ mod $F(x)$. The SR block is a g-bit shift register and the CNT block is a g-bit up counter. This counter is incremented by one in every two system clock cycles. Initially, CNT and location 0 of T are cleared to zero, and SR and LFSR are loaded with 1 and $A(x)$, respectively.

The updating of the T table using the structure of Fig. 2 proceeds as follows. When CNT=0, the LFSR contents are written on to T at location specified by SR, which corresponds to a base entry writing. When SR=CNT, both LFSR and SR are shifted one position left, CNT is cleared to zero and no read/write operation is performed on T. For a nonzero CNT value, which remains valid for two clock cycles, first the T table entry of the location specified by CNT is read and stored in the buffer. In the next clock cycle, the buffered value is added (mod 2) to the LFSR value and the result is written on to T at location specified by CNT + SR (mod 2). This corresponds to the updating of the non-basis entries of T. The entire updating operation is coordinated by the control block of Fig. 2 whose details are omitted here.

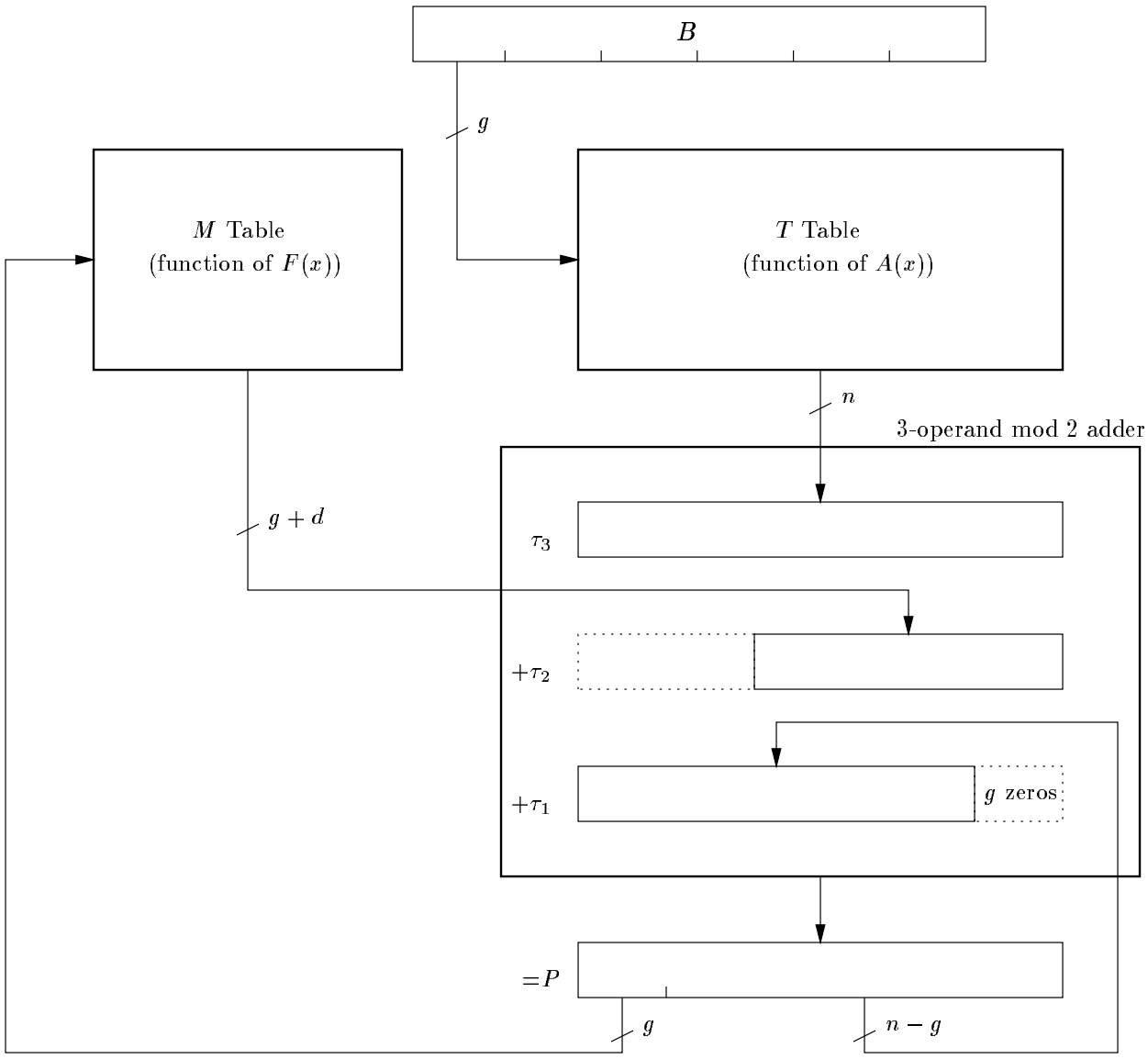

Fig. 1. Datapath for the loop-up table based multiplication in GF(2^m).

VIII. Conclusions

In this work, an algorithm for GF(2^n) multiplication has been proposed. The algorithm examines a group of bits of one operand in each iteration and uses two look-up tables. One of the tables is dependent on the irreducible polynomial used for defining the representation of the field elements. For many practical applications, this polynomial does not change frequently, hence the corresponding look-up table can be precomputed. The width of the table depends on the choice of the polynomial, and a list of good polynomials, which are of practical interest and result in look-up tables of width of 16 bits or less, has been given.

The second look-up table depends on one input operand of the multiplication operation. If the input is fixed, the table can be precomputed and optimized; otherwise it needs to be generated during the run time. In this effect, two algorithms, which are based on the entry computation on *demand* and in *window sequence*, have been presented. For today's cryptographic applications, the window sequence based algorithm is more advantageous.

A special purpose architecture for the hardware implementation of the multiplication algorithm has been presented. Also, the related structure for the generation of the look-up tables has been given. Hardware implementation can be used for achieving a high speed multiplier.

For a software implementation of the proposed GF(2^n) multiplication algorithm with the window sequence table generation scheme, the operand dependent look-up table is accessed to read s n-bit entries. Thus, on average an entry of the table is read $s/2^g$ times. Consequently, to have computational advantages of the usage of the table, we should choose g such that $2^g \leq \lceil \frac{n}{g} \rceil$. For $g = 2$, 4, 6 and 8, the corresponding cross over values of n are 8, 64, 384 and 2048, respectively.

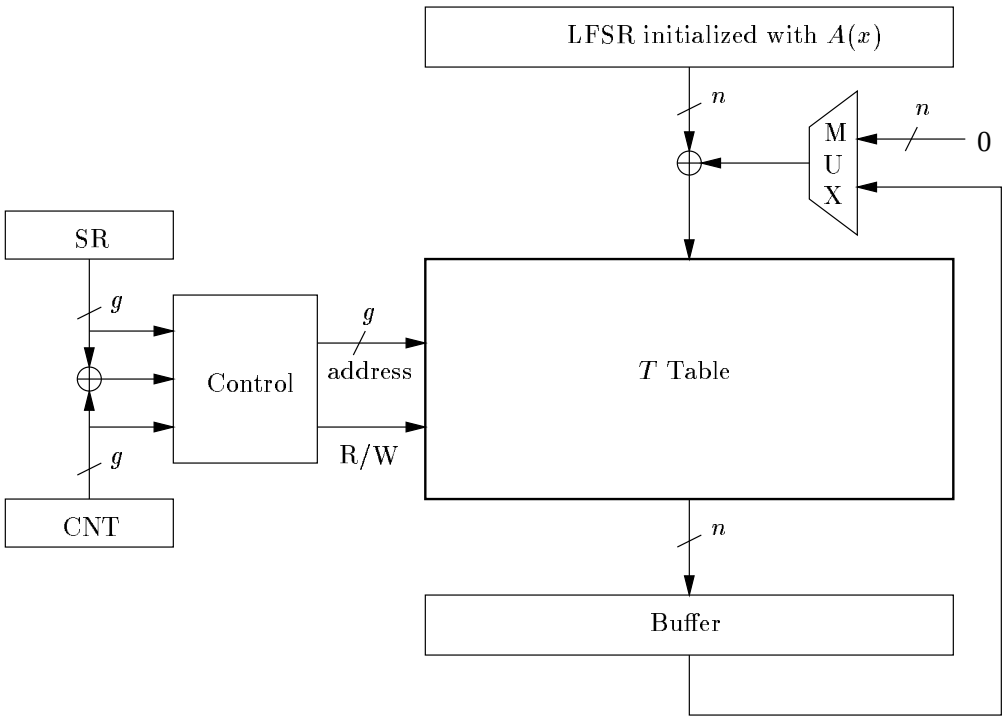

Fig. 2. Structure for the updating of the T table.

Acknowledgment

This work was done during the author's sabbatical leave with the Motorola Labs, Schaumburg, IL, USA. The author wishes to thank Larry Puhl and Ezzy Dabbish for their encouragement to pursue this work. Thanks are also due to Dean Vogler, Tom Messerges and L. Finkelstein, for their help with the various computing resources of the labs.

References

[1] M. A. Hasan, "Look-up Table Based Large Finite Field Multiplication in Memory Constrained Cryptosystems," in *Proceedings of the Seventh IMA Conf. on Cryptography and Coding, Lecture Notes in Computer Science*, pp. 213–221, Springer-Verlag, 1999.

[2] G. B. Agnew, R. C. Mullin, and S. A. Vanstone, "An Implementation of Elliptic Curve Cryptosystems over F_{2155}," *IEEE J. on Selected Areas in Communications*, vol. 11, pp. 804–813, June 1993.

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

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

[5] J. Guajardo and C. Paar, "Efficient Algorithms for Elliptic Curve Cryptosystems," in *Advances in Cryptology- CRYPTO '97, Lecture Notes in Computer Science*, pp. 342–356, Springer-Verlag, 1997.

[6] C. Koc and T. Acar, "Montgomery Multiplication in GF(2^k)," *Design, Codes and Cryptography*, vol. 14(1), pp. 57–69, Apr. 1998.

[7] L. Song and K. K. Parhi, "Low Energy Digit-Serial/Parallel Finite Field Multipliers," *Journal of VLSI Signal Processing*, vol. 19, pp. 149–166, June 1998.

[8] E. D. Mastrovito, *VLSI Architectures for Computations in Galois Fields*. PhD thesis, Dept. Elect. Eng., Linköping University, Linköping, Sweden, 1991.

[9] C. Koc and B. Sunar, "Mastrovito Multiplier for All Trinomials," *IEEE Trans. Computers*, vol. 48, pp. 522–527, May 1999.

[10] T. Itoh and S. Tsujii, "Structure of parallel multipliers for a class of fields GF(2^m)," *Inform. and Comp.*, vol. 83, pp. 21–40, 1989.

[11] M. A. Hasan, M. Z. Wang, and V. K. Bhargava, "Modular construction of low complexity parallel multipliers for a class of finite fields GF(2^m)," *IEEE Trans. Comput.*, vol. 41, pp. 962–971, Aug. 1992.

[12] Certicom Research, "GEC1: Recommended Elliptic Curve Domain Parameters," in *Standards for Efficient Cryptography Group*, http://www.secg.org, 1999.

[13] C. Paar, "A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields," *IEEE Trans. Computers*, vol. 45(7), pp. 856–861, 1996.