

A Generalized Method for Constructing Subquadratic Complexity $GF(2^k)$ Multipliers

Berk Sunar, *Member, IEEE Computer Society*

Abstract—We introduce a generalized method for constructing subquadratic complexity multipliers for even characteristic field extensions. The construction is obtained by recursively extending short convolution algorithms and nesting them. To obtain the short convolution algorithms, the Winograd short convolution algorithm is reintroduced and analyzed in the context of polynomial multiplication. We present a recursive construction technique that extends any d point multiplier into an $n = d^k$ point multiplier with area that is subquadratic and delay that is logarithmic in the bit-length n . We present a thorough analysis that establishes the exact space and time complexities of these multipliers. Using the recursive construction method, we obtain six new constructions, among which one turns out to be identical to the Karatsuba multiplier. All six algorithms have subquadratic space complexities and two of the algorithms have significantly better time complexities than the Karatsuba algorithm.

Index Terms—Bit-parallel multipliers, finite fields, Winograd convolution.



1 INTRODUCTION

FINITE fields have numerous applications in digital signal processing [1], [2], coding theory [3], [4], [5], and cryptography [6], [7], [8]. The efficient implementation of finite field operations, i.e., addition, multiplication, and inversion, is therefore critical for many applications. The efficiency of field operations is intimately related to the particular representation of the field elements (and, hence, the operands). It is customary to view a finite field as a vector space which facilitates the use of various bases for the representation of the field elements. Among the many proposed bases, the canonical basis [4], the normal basis [8], and the dual basis [9], [10], [11] are the most commonly used ones.

In this paper, we focus our attention on canonical basis bit-parallel multiplier architectures for even characteristic field extensions, i.e., $GF(2^k)$. In the canonical basis, field elements are expressed as polynomials with coefficients from a base field (e.g., $GF(2)$) and field multiplication is carried out as polynomial multiplication modulo an irreducible polynomial. Modular multiplication is usually carried out in two steps: polynomial multiplication and reduction of the polynomial product with the irreducible polynomial. Using the “paper and pencil” method, the multiplication step may be performed in parallel using n^2 two input AND gates and $(n - 1)^2$ two input XOR gates. This is followed by the reduction step, which requires only a number of additions depending on the number of nonzero coefficients of the irreducible modulus polynomial. Using low-Hamming weight polynomials (e.g., trinomials, pentanomials, etc.) the additions may be implemented with only $2n$ XOR gates. The total gate consumption becomes $2n^2 - 1$. On the other hand, when binary adder trees are utilized to

accumulate the partial products, the total delay of the bit-parallel multiplier is found as $T_{\otimes} + (\log_2 n)T_{\oplus}$, where T_{\otimes} and T_{\oplus} denote the delays of a two input AND gate and a two input XOR gate, respectively. A variation to this theme which combines the multiplication and reduction steps was proposed by Mastrovito in [12]. Although considerable research went into the design and optimization on this theme and its variations, the space and time complexity figures have not improved much further than the quadratic space and logarithmic time complexities cited above [13], [14], [15], [16], [17], [18].

An alternative approach for the initial multiplication step was developed by using the polynomial version of the Karatsuba-Ofman algorithm [19]. The recursive splitting of polynomials and the special reassembly of the partial products drastically reduces the number of AND gates required for the multiplication operation: $n^{\log_2 3}$. Although the reduction in the number of multiplications is accomplished by sacrificing extra additions in each step, since the number of steps is reduced logarithmically, the number of XOR gates also enjoys a much improved asymptotic complexity of $6n^{\log_2 3} - 8n + 2$ [15]. While the number of AND gates improves for all $n \geq 2$, the number of XOR gates improves only after $n \geq 64$ when compared to the paper and pencil multiplication method. The main disadvantages of the Karatsuba multiplier are its time complexity, which is about three times that of traditional multipliers, $T_{\otimes} + 3(\log_2 n)T_{\oplus}$, and the restriction it places on the bit-length by requiring it to be a power of two.

In this paper, we pose and address the following question: Besides the Karatsuba algorithm, are there any other bit-parallel multipliers with subquadratic space complexity and perhaps with better time complexity? To answer this question, we first reintroduce the Winograd short convolution algorithm, an old and well-known digital signal processing technique, in the context of polynomial multiplication. We then introduce a generalized recursive construction technique and present its complexity analysis.

• The author is with Worcester Polytechnic Institute, Atwater Kent Room 302, 100 Institute Rd. Worcester, MA 01609. E-mail: sunar@wpi.edu.

Manuscript received 12 Aug. 2003; revised 7 Jan. 2004; accepted 29 Jan. 2004. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0130-0803.

We summarize the complexities for several constructions. Finally, we discuss the generalization of this technique to support arbitrary polynomial lengths and present its complexity analysis.

2 WINOGRAD SHORT CONVOLUTION

In this section, we briefly describe the short convolution algorithm introduced by Winograd [20], [21]. Let $a(x)$ and $b(x)$ represent $d-1$ degree polynomials $a(x) = \sum_{i=0}^{d-1} a_i x^i$ and $b(x) = \sum_{i=0}^{d-1} b_i x^i$ defined over an arbitrary field. The product of the polynomials is computed as

$$c(x) = a(x)b(x) = \sum_{k=0}^{2d-2} c_k x^k,$$

where

$$c_k = \sum_{i+j=k} a_i b_j, \quad \text{for } 0 \leq k \leq 2d-2. \quad (1)$$

Alternatively, one can associate polynomials with sequences and view their coefficients as sequence elements:

$$\begin{aligned} a(x) &\rightarrow a = \{a_0, a_1, a_2, \dots, a_{d-1}\} \\ b(x) &\rightarrow b = \{b_0, b_1, b_2, \dots, b_{d-1}\}. \end{aligned}$$

This representation enables us to define the elements of the linear convolution $\bar{c} = a * b$ of the two sequences as follows:

$$\bar{c}_k = \sum_{i+j=k} a_i b_j, \quad \text{for } 0 \leq k \leq 2d-2. \quad (2)$$

It is easily seen that the polynomial multiplication in (1) and the linear convolution in (2) are equivalent operations since $c_k = \bar{c}_k$ for $0 \leq k \leq 2d-2$. The Winograd short convolution algorithm is based on this association, along with the following key observation: As introduced earlier, $a(x)$ and $b(x)$ represent polynomials of maximum degree $d-1$. Their product $c(x) = a(x)b(x)$ will have maximum degree $2d-2$. Hence, when a *made-up* modulus $m(x)$ with $\deg(m(x)) > 2d-2$ is introduced, then their product is not affected:

$$\begin{aligned} c'(x) &= a(x)b(x) \bmod m(x) \\ c(x) &= a(x)b(x) \\ c(x) &= c'(x), \quad \text{if } \deg(m(x)) > 2d-2. \end{aligned}$$

The Winograd short convolution algorithm makes use of this fact. Choosing the factors of $m(x)$ as relatively prime polynomials enables efficient residue arithmetic with shorter operands. The final product is obtained from the residues via the Chinese Remainder Theorem (CRT). The steps of the Winograd convolution algorithm are summarized below:

Setup: Choose artificial modulus $m(x)$ with pairwise relatively prime factors as

$$m(x) = m^{(1)}(x)m^{(2)}(x)\dots m^{(k)}(x).$$

Compute the polynomials $M^{(i)}(x)$ given as follows:

$$M^{(i)}(x) = \left(\frac{m(x)}{m^{(i)}(x)} \right) \left(\left(\frac{m(x)}{m^{(i)}(x)} \right)^{-1} \bmod m^{(i)}(x) \right) \quad (3)$$

for $i = 1, \dots, k$.

Step 1 Residue Computations: Find the residue representations of $a(x)$ and $b(x)$ with respect to each factor of the modulus

$$a^{(i)}(x) = a(x) \bmod m^{(i)}(x) \quad \text{and} \quad (4)$$

$$b^{(i)}(x) = b(x) \bmod m^{(i)}(x) \quad (5)$$

for $i = 1, \dots, k$.

Step 2 Residue Products: Compute k parallel modular polynomial products

$$w^{(i)}(x) = a^{(i)}(x)b^{(i)}(x) \bmod m^{(i)}(x) \quad (6)$$

for $i = 1, \dots, k$.

Step 3 Inversion: Compute the CRT-inverse using

$$c'(x) = \sum_{i=1}^k w^{(i)}(x)M^{(i)}(x) \bmod m(x). \quad (7)$$

If $\deg(m(x)) > 2d-2$, then $c(x) = a(x)b(x) = c'(x)$ and the algorithm will terminate with this step. However, due to difficulties in selecting relatively prime factors, the case $\deg(m(x)) = 2d-2$ is also useful. The algorithm proceeds as described earlier. The computed product $c'(x)$ may not match the actual product $c(x)$ since, in the product $a(x)b(x)$, the term $a_{d-1}b_{d-1}x^{2d-2}$ causes a reduction unless $a_{d-1}b_{d-1} = 0$. Hence, $c(x)$ and $c'(x)$ are related to each other as follows:

$$\begin{aligned} c'(x) &= a(x)b(x) \bmod m(x) \\ &= a(x)b(x) - a_{d-1}b_{d-1}m(x) \\ &= c(x) - a_{d-1}b_{d-1}m(x). \end{aligned}$$

Fortunately, the desired product $a(x)b(x)$ is easily recovered from $c'(x)$ by a simple correction step.

Step 4 Correction: If $\deg(m(x)) > 2d-2$, then $c(x) = c'(x)$ else if $\deg(m(x)) = 2d-2$, then compute

$$c(x) = c'(x) + a_{d-1}b_{d-1}m(x).$$

In the correction step due to the $a_{d-1}b_{d-1}$ term, an additional multiplication is required. However, the degree of $m(x)$ is reduced by one. It is customary to indicate the use of the correction technique by adding a symbolic $(x - \infty)$ factor to $m(x)$.

In the following section, we introduce a concrete example of a 4×4 Winograd short convolution algorithm. In this construction and others described later in the paper, the expressions become quite cumbersome. To alleviate this problem we first introduce the following notation:

Notation. We define a *product element* as follows:

$$p_\ell = \left(\sum_{\forall i \in \ell} a_i \right) \left(\sum_{\forall j \in \ell} b_j \right).$$

Here, p_ℓ denotes the product of two summations formed by the coefficients of $a(x)$ and $b(x)$ and ℓ denotes the list of indexes of the summed coefficients. For example,

$$p_{013} = (a_0 + a_1 + a_3)(b_0 + b_1 + b_3).$$

3 A CONSTRUCTION FOR $d = 4$

In this section, we illustrate the construction of a 4×4 Winograd short convolution algorithm over a field of characteristic 2. Our purpose is to construct an algorithm to compute the product

$$\begin{aligned} c(x) &= a(x)b(x) \\ &= (a_3x^3 + a_2x^2 + a_1x + a_0)(b_3x^3 + b_2x^2 + b_1x + b_0). \end{aligned}$$

- **Setup.** We apply the Winograd short convolution algorithm for $d=4$ by selecting the modulus polynomial as

$$m(x) = x^2(x^2 + 1)(x^2 + x + 1)(x - \infty).$$

For CRT-inversion, we compute the polynomials $M^{(i)}(x)$ as given in (3) and find $M^{(1)}(x) = x^5 + x^3 + x^2 + 1$, $M^{(2)}(x) = x^5 + x^4 + x^3$, and $M^{(3)}(x) = x^4 + x^2$.

- **Residue Computations.** We reduce $a(x)$ and $b(x)$ modulo $m^{(i)}(x)$ for $i = 1, 2, 3$ and obtain the following residues:

$$\begin{aligned} a^{(1)}(x) &= a_1x + a_0 \\ a^{(2)}(x) &= (a_3 + a_1)x + (a_2 + a_0) \\ a^{(3)}(x) &= (a_2 + a_1)x + (a_3 + a_2 + a_0) \\ b^{(1)}(x) &= b_1x + b_0 \\ b^{(2)}(x) &= (b_3 + b_1)x + (b_2 + b_0) \\ b^{(3)}(x) &= (b_2 + b_1)x + (b_3 + b_2 + b_0). \end{aligned}$$

- **Residue Products.** The residue products are computed as

$$\begin{aligned} w^{(1)}(x) &= (a_1x + a_0)(b_1x + b_0) \\ &= (a_0b_1 + a_1b_0)x + a_0b_0 \pmod{x^2} \\ &= (p_{01} - p_0 - p_1)x + p_0 \\ w^{(2)}(x) &= [(a_3 + a_1)x + (a_2 + a_0)] \\ &\cdot [(b_3 + b_1)x + (b_2 + b_0)] \pmod{x^2 + 1} \\ &= (a_3 + a_1)(b_3 + b_1)x^2 \\ &+ [(a_3 + a_1)(b_2 + b_0) + (b_3 + b_1)(a_2 + a_0)]x \\ &+ (a_2 + a_0)(b_2 + b_0) \pmod{x^2 + 1} \\ &= (p_{0123} + p_{03} + p_{13})x + (p_{02} + p_{13}) \\ w^{(3)}(x) &= [(a_2 + a_1)x + (a_3 + a_2 + a_0)] \\ &\cdot [(b_2 + b_1)x + (b_3 + b_2 + b_0)] \pmod{x^2 + x + 1} \\ &= (a_2 + a_1)(b_2 + b_1)x^2 + [(a_2 + a_1)(b_3 + b_2 + b_0) \\ &+ (a_3 + a_2 + a_0)(b_2 + b_1)]x \\ &+ (a_3 + a_2 + a_0)(b_3 + b_2 + b_0) \pmod{x^2 + x + 1} \\ &= (p_{013} + p_{023})x + (p_{023} + p_{12}). \end{aligned}$$

- **Inversion.** Using (7) we derive the CRT-inverse $c'(x) = \sum_{i=0}^5 c'_i x^i$ as

$$\begin{aligned} c'_0 &= p_0 \\ c'_1 &= p_{01} + p_0 + p_1 \\ c'_2 &= p_{0123} + p_{023} + p_{12} + p_{01} + p_{02} + p_{13} + p_1 \\ c'_3 &= p_{0123} + p_{023} + p_{013} + p_0 \\ c'_4 &= p_{0123} + p_{023} + p_{12} + p_{01} + p_0 + p_1 \\ c'_5 &= p_{023} + p_{013} + p_{02} + p_{01} + p_{13} + p_1. \end{aligned}$$

- **Correction.** We implement the correction step $c(x) = c'(x) + p_3m(x)$ and obtain the coefficients of the desired product as

$$\begin{aligned} c_0 &= p_0 \\ c_1 &= p_{01} + p_0 + p_1 \\ c_2 &= p_{0123} + p_{023} + p_{12} + p_{01} + p_{02} + p_{13} + p_1 + p_3 \\ c_3 &= p_{0123} + p_{023} + p_{013} + p_0 + p_3 \\ c_4 &= p_{0123} + p_{023} + p_{12} + p_{01} + p_0 + p_1 \\ c_5 &= p_{023} + p_{013} + p_{02} + p_{01} + p_{13} + p_1 + p_3 \\ c_6 &= p_3. \end{aligned}$$

The final algorithm requires $\mathcal{S}_{\otimes} = 10$ multiplications:

$$p_{0123}, p_{023}, p_{013}, p_{01}, p_{12}, p_{02}, p_{13}, p_0, p_1, p_3.$$

This is a significant improvement over the 16 multiplications required by the paper and pencil multiplication method. However, this improvement comes at the price of an increased number of additions. First, we need to compute the summation elements that are multiplied together to obtain the product terms. This is achieved by computing

$$a_0 + a_1, a_1 + a_2, a_0 + a_2, a_1 + a_3$$

and then $(a_0 + a_1) + a_3$, $(a_0 + a_2) + a_3$, and $(a_0 + a_2) + (a_1 + a_3)$ in order. Note that we are reusing the previously computed summation terms. This computation requires seven additions. Likewise, another seven additions are spent for preparing the coefficients of $b(x)$. Hence, the number of additions required to compute the 10 product terms is $\mathcal{S}_{\oplus}^{pre} = 14$. We call these additions *preadditions* to differentiate from additions performed after the multiplications, which we call *postadditions*. The postadditions may be realized with the following algorithm.

$$\begin{aligned} c_0 &= p_0 \\ c_1 &= (p_{01} + p_1) + p_0 \\ c_2 &= [(p_{0123} + p_{023}) + (p_3 + p_{12})] + [(p_{01} + p_1) + (p_{02} + p_{13})] \\ c_3 &= (p_{0123} + p_{023}) + (p_{013} + p_3) + p_0 \\ c_4 &= [(p_{0123} + p_{023}) + p_{12}] + [(p_{01} + p_1) + p_0] \\ c_5 &= (p_{013} + p_3) + (p_{01} + p_1) + (p_{02} + p_{13}) + p_{023} \\ c_6 &= p_3. \end{aligned}$$

The parentheses and brackets indicate shared subexpressions and hint at the order of additions. Note that the groupings of preadditions and postadditions are achieved by inspection. The ad hoc nature of this step makes it much more difficult to realize for longer convolution algorithms. In this example, by reusing common subexpressions, the number of postadditions is reduced to $\mathcal{S}_{\oplus}^{post} = 16$.

The delay associated with this algorithm may be analyzed in three parts: preadditions, multiplications, and postadditions. Assuming two input XOR gates, the preadditions may be performed using binary XOR trees. The longest delay path will be caused by the term p_ℓ with the longest list ℓ . In both multipliers, the maximum delay is found for computing p_{0123} with length 4 and delay $2T_\oplus$. Assuming all multiplications are performed in parallel, the delay is found as T_\otimes . For the postadditions, the longest delay path is found in coefficient c_2 with eight terms and three addition delays. The total delay is found as $T_\otimes + 5T_\oplus$.

4 COMPLEXITY OF WINOGRAD'S ALGORITHM

In this section, we analyze the complexity of the Winograd short convolution algorithm and establish conditions to achieve subquadratic complexity. We start by assuming that all residue products in Step 2 of Winograd's algorithm are computed using the paper and pencil method and, therefore, require a number of multiplications quadratic in the length. The entire convolution algorithm will have subquadratic multiplicative complexity if $S_\otimes < d^2$ and, hence, for our purposes, the following condition needs to be satisfied:

$$S_\otimes = \sum_{i=1}^k d_i^2 < d^2.$$

Here, $d_i = \deg(m^{(i)}(x))$ and $d = \deg(m(x))$. Also note that the Winograd algorithm establishes a lower bound as follows:

$$\sum_{i=1}^k d_i \geq 2d - 1.$$

Since $\sum_{i=1}^k d_i^2 \geq \sum_{i=1}^k d_i$, we can combine the two conditions to bound S_\otimes from both sides:

$$\begin{aligned} 2d - 1 &\leq \sum_{i=1}^k d_i \leq \sum_{i=1}^k d_i^2 < d^2 \\ 2d - 1 &\leq S_\otimes < d^2. \end{aligned}$$

In setting up the Winograd algorithm, we usually pick the equality case $\sum_{i=1}^k d_i = 2d - 1$ to eliminate redundant multiplications. To minimize S_\otimes , one has to maximize the number of factors of $m(x)$. Ideally, if all factors are chosen as first degree polynomials, then $d_i = 1$, $k = 2d - 1$, and

$$S_\otimes = 2d - 1.$$

This represents the minimum number multiplications required to multiply two polynomials of length d . As attractive as it may seem, for longer convolutions, finding $m(x)$ with linear factors becomes impossible and the performance suffers degradation. Still, there are sufficiently many relatively prime polynomials that can be used for the construction of Winograd short convolution algorithms with subquadratic multiplication complexity.

As our goal is to use shorter Winograd convolution algorithms to build longer ones, we develop a simple but useful parameterization. This will enable us in later sections to formulate the complexity of longer convolution algorithms in terms of the parameters of the shorter convolution

algorithms used in the construction. As already used above, S_\otimes denotes the number of multiplications incurred in Step 2 of the Winograd algorithm. There are two types of additions concentrated in Steps 1 and 3 of the algorithm. These additions take place before and after the multiplications are computed. Hence, as described earlier in Section 3, we call these additions *preadditions* and *postadditions* and denote their complexities by S_\oplus^{pre} and S_\oplus^{post} , respectively. The space complexity of a short convolution algorithm is identified by an ensemble of three parameters: $(S_\oplus^{pre}, S_\otimes, S_\oplus^{post})$.

The delay of the multiplier circuit is found in three parts. We use the parameter D^{pre} to denote the delay associated with computing the preadditions. In a direct implementation, the length of the longest list ℓ of all product terms p_ℓ appearing in the preadditions will determine D^{pre} . When binary XOR trees are utilized to implement the preadditions, then the delay is found as $D^{pre} = \lceil \log_2 \ell \rceil$. In the computation of the residue products, all multiplications are performed in parallel and thus require only one coefficient multiplication delay. The maximum delay in the postadditions is determined by the coefficient that causes the longest delay. We associate the parameter D^{post} with this delay. In a direct implementation, this coefficient will be the one with the maximum number of product terms. As in the preadditions, if the postadditions are carried out using binary XOR trees, the delay is determined as $D^{post} = \lceil \log_2 l \rceil$. Here, l denotes the maximum number of product terms in any coefficient of the product.

The complexity analysis presented in this section applies to the direct implementation of the Winograd algorithm. However, there are many ad hoc techniques which may be utilized to obtain improvements. One of these is as follows:

Heuristic 1 (convolutional symmetry). The convolution operation is symmetric. More clearly, when two sequences are reversed, their convolution is also reversed. Therefore, whenever a convolution sequence is not symmetric, it may be possible to simplify the more complex sequence elements by deriving them from their symmetric counterparts by applying the two sequences in reverse.

In a more mathematical form, the heuristic is restated as follows: Consider the convolution of two sequences $\{c\} = \{a\} \star \{b\}$ or the product computation of the polynomial product $c(x) = a(x)b(x)$. We treat the elements of the convolution sequence as functions of the two sequences:

$$c_j = f_j(a_0, a_1, \dots, a_{d-1}; b_0, b_1, \dots, b_{d-1})$$

for $j = 0, \dots, d - 1$. Since the convolution operation is symmetric, it follows that

$$c_{2d-2-j} = f_j(a_{d-1}, a_{d-2}, \dots, a_0; b_{d-1}, b_{d-2}, \dots, b_0)$$

In other words, c_{2d-2-j} may be obtained by replacing all a_i with a_{d-1-i} and all b_i with b_{d-1-i} in the equation obtained for computing c_j for any $0 \leq j \leq d - 1$. For a sample application of this heuristic, see Case $d = 4$ in the Appendix (which can be found on the Computer Society Digital Library at <http://computer.org/tc/archives.htm>).

5 THE KARATSUBA-OFMAN ALGORITHM

In this section, we show that a certain Winograd short convolution algorithm is essentially identical to the Karatsuba-Ofman algorithm [19]. We state this formally as follows:

Theorem 1. *The 2×2 Winograd short convolution algorithm constructed using the modulus $m(x) = x(x-1)(x-\infty)$ is identical to the Karatsuba-Ofman algorithm.*

Proof. We aim to develop a multiplier to realize the product

$$c(x) = a(x)b(x) = (a_0 + a_1x)(b_0 + b_1x).$$

Using the given modulus, the residues are computed as

$$\begin{aligned} a^{(1)}(x) &= a_0 & a^{(2)}(x) &= a_0 + a_1 \\ b^{(1)}(x) &= b_0 & b^{(2)}(x) &= b_0 + b_1. \end{aligned}$$

The residue products are formed as

$$\begin{aligned} w^{(1)}(x) &= a_0b_0 \\ w^{(2)}(x) &= (a_0 + a_1)(b_0 + b_1). \end{aligned}$$

The inverse polynomials are computed as

$$\begin{aligned} M^{(1)}(x) &= (x-1)((x-1)^{-1} = x-1 \pmod{x}) \\ M^{(2)}(x) &= x(x^{-1}) = x \pmod{(x-1)}. \end{aligned}$$

By using the CRT inversion formula shown in (7), we assemble the product as follows:

$$c'(x) = a_0b_0(x-1) + (a_0 + a_1)(b_0 + b_1)x.$$

Due to the $(x-\infty)$ factor, a correction step is needed. We add $a_1b_1x(x-1)$ to obtain the final product as

$$c(x) = a_1b_1x^2 + [(a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1]x + a_0b_0. \quad \square$$

6 RECURSIVE SPLITTING

Using Winograd's technique one may easily build a convolution algorithm for a desired length, as shown in the previous sections. However, as the size of the polynomials grows, it becomes difficult to pick a modulus with proper factors. The problem gets worse if the domain of the polynomial coefficients is a small field such as $GF(2)$. Another serious problem is the fast growth in the number of addition operations required to realize the reductions and the CRT inverse computation. Both difficulties may be overcome by recursively nesting such short convolution algorithms to achieve longer convolutions. The same principle was successfully applied in the multilevel realization of the Karatsuba algorithm [15]. A k -level recursive construction takes a $d \times d$ convolution (or multiplication) algorithm and extends it to an $n \times n$ convolution algorithm, where $n = d^k$.

Let the product to be computed be $c(x) = a(x)b(x)$, where the two operands are represented as

$$a(x) = A_0^{(0)} + A_1^{(0)}x + A_2^{(0)}x^2 + \dots + A_{d^k-1}^{(0)}x^{d^k-1}$$

and

$$b(x) = B_0^{(0)} + B_1^{(0)}x + B_2^{(0)}x^2 + \dots + B_{d^k-1}^{(0)}x^{d^k-1}$$

with $A_i^{(0)}, B_i^{(0)} \in GF(2)$. The algorithm proceeds by splitting the polynomials $a(x)$ and $b(x)$ into d shorter polynomials. After the splitting, $a(x)$ and $b(x)$ have d terms, each of length $n/d = d^{k-1}$, i.e.,

$$a(x) = A_0^{(1)} + A_1^{(1)}x^{d^{k-1}} + A_2^{(1)}x^{2d^{k-1}} + \dots + A_{d-1}^{(1)}x^{(d-1)d^{k-1}}, \quad (8)$$

where each coefficient is a $(d^{k-1} - 1)$ st degree polynomial

$$\begin{aligned} A_i^{(1)} &= A_{id^{k-1}}^{(0)} + A_{id^{k-1}+1}^{(0)}x + A_{id^{k-1}+2}^{(0)}x^2 + \\ &\dots + A_{id^{k-1}+d^{k-1}-1}^{(0)}x^{d^{k-1}-1} \end{aligned}$$

for $i = 0, 1, \dots, d-1$. Since both operands now have d coefficients, to multiply the coefficients of $a(x)$ and $b(x)$, a $d \times d$ convolution algorithm may be utilized. With the application of the convolution algorithm, new products of the coefficients of $a(x)$ and $b(x)$ are formed. These are products of polynomials of length d^{k-1} . To realize these products, the splitting technique is used and another level of the $d \times d$ convolution algorithm is applied on the coefficients. In fact, the splitting and convolution operations may be applied repeatedly until the ground field $GF(2)$ is reached. The coefficients in a level are related to the ones in the previous level as follows:

$$\begin{aligned} A_i^{(j)} &= A_{id^{k-j}}^{(j-1)} + A_{id^{k-j}+1}^{(j-1)}x + A_{id^{k-j}+2}^{(j-1)}x^2 + \\ &\dots + A_{id^{k-j}+d^{k-j}-1}^{(j-1)}x^{d^{k-j}-1}. \end{aligned}$$

The $d \times d$ convolution algorithm is applied to all multiplications in each of the k levels until the ground field is reached. Once the ground field is reached, the products are computed and the polynomials are appropriately reassembled according to the short convolution algorithm.

The recursive splitting technique not only allows a simple and uniform construction, but effectively reduces the asymptotic complexity of the multiplier. In the next section, we present a detailed space and time complexity analysis which derives the overall complexity in terms of the parameters of the short convolution algorithm.

7 COMPLEXITY ANALYSIS

The analysis assumes a short convolution algorithm characterized by the parameters $(d, \mathcal{S}_{\oplus}^{pre}, \mathcal{S}_{\otimes}, \mathcal{S}_{\oplus}^{post}, \mathcal{D}^{pre}, \mathcal{D}^{post})$ as defined in Section 4. The analysis proceeds in three steps:

1. **Preadditions.** In each recursion level, the number of polynomials grows by a factor of \mathcal{S}_{\otimes} , whereas polynomial lengths shrink d times. Hence, the number of preadditions performed in the overall computation is found as:

$$\begin{aligned}\mathcal{R}_{\oplus}^{pre} &= \sum_{i=1}^{\log_d n} \mathcal{S}_{\oplus}^{i-1} \mathcal{S}_{\oplus}^{pre} \frac{n}{d^i} \\ &= \frac{\mathcal{S}_{\oplus}^{pre}}{\mathcal{S}_{\oplus} - d} (n^{\log_d \mathcal{S}_{\oplus}} - n).\end{aligned}$$

In each level of the recursion, the preadditions are performed with \mathcal{D}^{pre} delay. Since there are $\log_d n$ levels, the delay in this stage is found as:

$$T_{\oplus}^{pre} = T_{\oplus} \mathcal{D}^{pre} \log_d n.$$

2. **Multiplications.** The polynomials are split $\log_d n$ times, the number of polynomials grows by a factor of \mathcal{S}_{\otimes} . Hence, the total number of coefficient multiplications (after $\log_d n$ levels of recursion) is

$$\mathcal{R}_{\otimes} = \mathcal{S}_{\otimes}^{\log_d n} = n^{\log_d \mathcal{S}_{\otimes}}.$$

All multiplications are computed in parallel and thus require only one multiplication delay.

$$T = T_{\otimes}.$$

3. **Postadditions.** In calculating the total number of postadditions, we need to consider the additions required for the reassembly of the coefficients as well as the overlaps among the coefficients. Note that product terms may be twice as long as coefficients, thus overlaps among coefficients will occur. In the i th iteration, the number of polynomials grows to $\mathcal{S}_{\otimes}^{\log_d n - i}$. Each polynomial has length $2d^{i-1} - 1$. Since there are $\mathcal{S}_{\oplus}^{post}$ additions of such polynomials, multiplying these quantities together gives the number additions as shown in the first term in the summation below. The product terms will have $2d - 1$ coefficients, hence there are $2d - 2$ intervals with overlaps each $d^{i-1} - 1$ bits long. Factoring in these overlaps results in the following overall figure for postadditions:

$$\begin{aligned}\mathcal{R}_{\oplus}^{post} &= \sum_{i=1}^{\log_d n} \mathcal{S}_{\otimes}^{\log_d n - i} [\mathcal{S}_{\oplus}^{post} (2d^{i-1} - 1) \\ &\quad + (2d - 2)(d^{i-1} - 1)].\end{aligned}$$

The delay contributed by postadditions is found as

$$T_{\oplus}^{post} = T_{\oplus} \mathcal{D}^{post} \log_d n.$$

The overall complexities \mathcal{R}_{\otimes} and $\mathcal{R}_{\oplus} = \mathcal{R}_{\oplus}^{pre} + \mathcal{R}_{\oplus}^{post}$ are found as

$$\mathcal{R}_{\otimes} = n^{\log_d \mathcal{S}_{\otimes}}, \quad (9)$$

$$\mathcal{R}_{\oplus} = (u - v)n^{\log_d \mathcal{S}_{\otimes}} - un + v, \quad (10)$$

where

$$u = \frac{2(d-1) + 2\mathcal{S}_{\oplus}^{post} + \mathcal{S}_{\oplus}^{pre}}{\mathcal{S}_{\otimes} - d}$$

and

TABLE 1
Change in the Degree of the Optimal Asymptotic Complexity

d	2	5	10	100	1000	...	∞
$\log_d(2d-1)$	1.584	1.365	1.278	1.149	1.100	...	1

$$v = \frac{2(d-1) + \mathcal{S}_{\oplus}^{post}}{\mathcal{S}_{\otimes} - 1}.$$

The total delay is found as follows:

$$T = T_{\otimes} + T_{\oplus}(\mathcal{D}^{pre} + \mathcal{D}^{post}) \log_d n. \quad (11)$$

We observe that the polynomial degree of the space complexity is $\log_d \mathcal{S}_{\otimes}$. When an optimally constructed Winograd short convolution algorithm is used, then $\mathcal{S}_{\otimes} = 2d - 1$ and the asymptotic complexity is optimized for that particular d . For instance, the Karatsuba algorithm is optimally constructed for $d = 2$ since $\mathcal{S}_{\otimes} = 3$. However, note that, in Table 1, for higher values of d , the exponent decreases and, in the limit case, converges to 1. This means that, as d increases, the asymptotic complexity of a multiplier built by extending an optimally constructed convolution algorithm will improve. Even if no optimal construction can be found for higher d , it may still be possible to find suboptimal ($\mathcal{S}_{\otimes} < 2d - 1$) short convolution algorithms. When these algorithms are recursively extended, it may still be possible to obtain asymptotic complexities that are better than the ones obtained for multipliers built using optimal constructions for smaller d .

8 SEVERAL CONSTRUCTIONS

Table 2 summarizes the complexities of several multiplier constructions obtained by recursively extending the Winograd short convolution algorithms derived in this paper. The listed complexities are obtained by a straightforward application of the formulae given in (9), (10), (11) on the convolution algorithm parameters ($\mathcal{S}_{\otimes}, \mathcal{S}_{\oplus}^{pre}, \mathcal{S}_{\oplus}^{post}, \mathcal{D}^{pre}, \mathcal{D}^{post}$). The complexities are given in terms of the length of the multiplier $n = d^k$. The first algorithm is identical to the Karatsuba algorithm as derived in Section 5. The two algorithms for $d = 4$ are derived in Section 3 and in the Appendix (which can be found on the Computer Society Digital Library at <http://computer.org/tc/archives.htm>). The algorithms for $d = 3$ and $d = 5$ are included in the Appendix. The stars in the two moduli indicate the improved versions of the two algorithms as explained in the Appendix.

In Table 2, when compared to the paper and pencil method, all six algorithms exhibit superior asymptotic complexities. Albeit very close, the degree of the asymptotic complexity of the Karatsuba algorithm, i.e., $\log_2 3 \approx 1.58$, is slightly better than that of the other six algorithms ($\log_3 6 \approx 1.63$, $\log_4 10 \approx 1.66$, $\log_5 14 \approx 1.64$). Nevertheless, the constant factor of the polynomial term in the additive complexities \mathcal{R}_{\oplus} for the improved algorithms for $d = 3$ and $d = 4$ is smaller than that of the Karatsuba algorithm. This translates into only slightly worse space complexity for certain practical values of n . For example, the additions required for the improved algorithms for $d = 3$ and $d = 4$

TABLE 2
Recursive Multiplier Complexities of Several Constructions

d	$m(x)$	\mathcal{R}_{\otimes}	\mathcal{R}_{\oplus}	\mathcal{T}
2	$x(x-1)(x-\infty)$	$n^{\log_2 3}$	$6n^{\log_2 3} - 8n + 2$	$\mathcal{T}_{\otimes} + 3 \log_2 n \mathcal{T}_{\oplus}$
3	$x(x+1)(x^2+x+1)(x-\infty)$	$n^{\log_3 6}$	$\frac{97}{15}n^{\log_3 6} - \frac{26}{3}n + \frac{11}{5}$	$\mathcal{T}_{\otimes} + 5 \log_3 n \mathcal{T}_{\oplus}$
3	$x(x+1)(x^2+x+1)(x-\infty)^*$	$n^{\log_3 6}$	$\frac{16}{3}n^{\log_3 6} - \frac{22}{3}n + 2$	$\mathcal{T}_{\otimes} + 4 \log_3 n \mathcal{T}_{\oplus}$
4	$x^2(x^2+1)(x^2+x+1)(x-\infty)$	$n^{\log_4 10}$	$\frac{56}{9}n^{\log_4 10} - \frac{26}{3}n + \frac{22}{9}$	$\mathcal{T}_{\otimes} + 5 \log_4 n \mathcal{T}_{\oplus}$
4	$x^2(x^2+1)(x^2+x+1)(x-\infty)^*$	$n^{\log_4 10}$	$\frac{47}{9}n^{\log_4 10} - \frac{22}{3}n + \frac{19}{9}$	$\mathcal{T}_{\otimes} + 5 \log_4 n \mathcal{T}_{\oplus}$
5	$x(x^2+1)(x^2+x+1)(x^3+x+1)(x-\infty)$	$n^{\log_5 14}$	$\frac{982}{117}n^{\log_5 14} - \frac{106}{9}n + \frac{44}{13}$	$\mathcal{T}_{\otimes} + 7 \log_5 n \mathcal{T}_{\oplus}$

are only about 10 percent and 20 percent more than what is required for the Karatsuba algorithm for $n=100$. Also, considering that the Karatsuba algorithm requires n to be a power of two, the algorithms for $d=3$ and $d=5$ clearly have practical value.

We see a different picture in the time complexities. With increasing d , the base of the logarithms increases. When all six complexities are converted to logarithms in base 2 and rounded appropriately, the logarithmic identities

$$\begin{aligned} 5 \log_3 n &= 3.15 \log_2 n, \text{ for } d=3 \\ 4 \log_3 n &= 2.52 \log_2 n, \text{ for } d=3 \\ 5 \log_4 n &= 2.53 \log_2 n, \text{ for } d=4 \\ 7 \log_5 n &= 3.01 \log_2 n, \text{ for } d=5 \end{aligned}$$

are obtained. Hence, the time complexities of the four algorithms developed for $d=3$ and $d=4$ are significantly better than that of the Karatsuba multiplier. Note that all six multipliers have worse time complexities than the paper and pencil multiplication method, which has only $\mathcal{T}_{\otimes} + \log_2 n \mathcal{T}_{\oplus}$ delay.

We would like to emphasize that these particular constructions are only a very small fraction of the multipliers that may be built by recursively extending short convolution algorithms obtained by Winograd's algorithm or by any other means. In the application of Winograd's algorithm, depending on the choice of the artificial modulus $m(x)$, the complexity parameters will change, giving rise to new recursive multiplication algorithms. Furthermore, any short convolution algorithm obtained by any method besides the Winograd algorithm may be recursively extended. The developed complexity formulae will work as long as the same multiplier parameters can be identified.

9 GENERALIZATION TO ARBITRARY n

In many applications, the multiplier length is determined by system constraints and, hence, n may not be a prime power. For such applications, it is crucial to build an efficient multiplier for arbitrary n . This may be accomplished by recursively combining multiple level splitting algorithms of different lengths. That is, if n factorizes into prime powers as

$$n = d_1^{e_1} d_2^{e_2} \dots d_t^{e_t},$$

then, as before, the polynomial operands are recursively split in e_1 levels and multiplied in each level using a $d_1 \times d_1$ convolution algorithm. However, the coefficients obtained in the final splitting are not elements of $GF(2)$ as before, but

rather elements of $GF(2^{d_1^{e_1} \dots d_t^{e_t}})$. Thus, another e_2 levels of recursive splitting and convolutions of length $d_2 \times d_2$ are applied. This process is repeated for all prime factors in the representation of n until the ground field is reached and the entire computation is expressed in terms of $GF(2)$ operations.

The complexity formulae developed in Section 7 still apply to each level of the recursive construction of the multiplier. However, to determine the overall complexity, the complexity formulae should be nested in accordance with the application order of the splitting:

$$\begin{aligned} \mathcal{R}_{\otimes, d_i} &= (d_i^{e_i})^{\log_{d_i} \mathcal{S}_{\otimes, d_i}} \mathcal{R}_{\otimes, d_{i+1}} = \mathcal{S}_{\otimes, d_i}^{e_i} \mathcal{R}_{\otimes, d_{i+1}} \\ \mathcal{R}_{\oplus, d_i} &= \left[(u_i - v_i) (d_i^{e_i})^{\log_{d_i} \mathcal{S}_{\otimes, d_i}} - u_i d_i^{e_i} + v_i \right] \mathcal{R}_{\oplus, d_{i+1}} \\ &= \left[(u_i - v_i) \mathcal{S}_{\otimes, d_i}^{e_i} - u_i d_i^{e_i} + v_i \right] \mathcal{R}_{\oplus, d_{i+1}}, \end{aligned}$$

where

$$u_i = \frac{2(d_i - 1) + 2\mathcal{S}_{\oplus, d_i}^{post} + \mathcal{S}_{\oplus, d_i}^{pre}}{\mathcal{S}_{\otimes, d_i} - d_i}$$

and

$$v_i = \frac{2(d_i - 1) + \mathcal{S}_{\oplus, d_i}^{post}}{\mathcal{S}_{\otimes, d_i} - 1}$$

for $i = 1, 2, \dots, t$. The subscript d_i associates a particular parameter to a $d_i \times d_i$ convolution algorithm and its application in the overall recursion. Note that $\mathcal{R}_{\otimes, d_i}$ and $\mathcal{R}_{\oplus, d_i}$ give the overall complexities of the multiplier. Also note that $\mathcal{R}_{\otimes, d_{i+1}}$ and $\mathcal{R}_{\oplus, d_{i+1}}$ indicate the complexities of multiplication and addition operations in $GF(2)$, respectively. The closed form expressions are found as

$$\begin{aligned} \mathcal{R}_{\otimes} &= \prod_{i=1}^t \mathcal{S}_{\otimes, d_i}^{e_i} \\ \mathcal{R}_{\oplus} &= \prod_{i=1}^t \left[(u_i - v_i) \mathcal{S}_{\otimes, d_i}^{e_i} - u_i d_i^{e_i} + v_i \right]. \end{aligned}$$

By ignoring linear terms and constants, the overall addition complexity may be approximated as follows:

$$\mathcal{R}_{\oplus} \approx \prod_{i=1}^t (u_i - v_i) \mathcal{S}_{\otimes, d_i}^{e_i} = \mathcal{R}_{\otimes} \prod_{i=1}^t (u_i - v_i).$$

The time complexity is derived by writing the latency of multiplication in a level in terms of the latency of operations in a lower level as follows:

$$\mathcal{T}_{\otimes, d_i} = \mathcal{T}_{\otimes, d_{i+1}} + \mathcal{T}_{\oplus, d_{i+1}} (\mathcal{D}_i^{pre} + \mathcal{D}_i^{post}) \log_{d_i} d_i^{e_i}.$$

The total delay is found by substituting $\mathcal{T}_{\otimes, d_{i+1}}$ until the ground field is reached. This gives a simple summation which simplifies to the following closed form expression:

$$\mathcal{T} = \mathcal{T}_{\otimes} + \mathcal{T}_{\oplus} \sum_{i=0}^{d-1} (\mathcal{D}_i^{pre} + \mathcal{D}_i^{post}) e_i.$$

We also would like to note that, although not treated in the paper, the generalization of the recursive extension technique and the presented complexity analysis to higher characteristic field extensions is trivial. Furthermore, over higher characteristic fields, selecting relatively prime polynomial factors in the setup of the Winograd algorithm becomes easier and, therefore, the algorithm and the recursive extension become more efficient. For instance, if $p > 2n$, then any Winograd $n \times n$ convolution algorithm may be realized using only $2n - 1$ multiplications since all factors of $m(x)$ can be selected as linear polynomials.

A final word on generalization is in order. It may appear that only a multiplier with a sufficiently composite bit-length would benefit from the recursive construction technique. In certain applications, nonprime bit-lengths may be prohibited. In fact, an attack based on Weil descent was shown [22] to be effective on elliptic curve discrete logarithm problems built over certain composite extension fields. Hence, in practice, composite extensions are avoided as much as possible in elliptic curve schemes. Fortunately, by simply appending the polynomials to be multiplied with zero coefficients or by partitioning them into shorter polynomials, the recursive construction can be made to work. For instance, if one wishes to multiply two polynomials, each of length $n = 251$, then, rather than finding a 251×251 convolution algorithm, one could partition $a(x)$ and $b(x)$ as

$$a(x) = \sum_{i=0}^{249} a_i x^i + a_{250} \quad \text{and} \quad b(x) = \sum_{i=0}^{249} b_i x^i + b_{250}$$

and obtain the product in four parts as

$$a(x)b(x) = \left(\sum_{i=0}^{249} a_i \sum_{i=0}^{249} b_i x^i \right) + a_{250} \sum_{i=0}^{249} b_i x^i + b_{250} \sum_{i=0}^{249} a_i x^i + a_{250} b_{250}.$$

The implementation of the last three terms is trivial. The first term requires a 250×250 convolution algorithm which could be built by using a 2×2 algorithm and a recursively extended $5^3 \times 5^3$ algorithm. Alternatively, one could append $a(x)$ and $b(x)$ with five zero coefficients and apply a recursively extended $2^8 \times 2^8$ convolution algorithm.

10 CONCLUSION

We introduced a generalized construction method for building low-complexity bit-parallel multipliers for arbitrary bit-lengths. The construction is based on a divide and conquer strategy where, through the prime factorization of the bit-length, a multiplier of arbitrary length is built using

shorter multiplication algorithms. The Winograd algorithm provides a straightforward construction method for such short convolution algorithms. In Section 6, we have presented a recursive construction technique that extends any d point multiplier with \mathcal{S}_{\otimes} multiplications into an $n = d^k$ point multiplier with $\mathcal{O}(n^{\log_d \mathcal{S}_{\otimes}})$ area and $\mathcal{O}(\log_d n)$ delay. Note that the recursive combination technique and its complexity analysis presented in Section 7 apply to any multiplication algorithm. The generalized construction method presented in Section 9 nests such multipliers to build low-complexity bit-parallel multipliers for arbitrary bit-lengths. The exact space and time complexities of the generalized multiplier are derived in terms of the parameters of the short convolution algorithms. We provide a simple formula to compute the overall space and time complexities of the multiplier from the short convolution parameters. In Section 9, we note that certain applications restrict the bit-length to be a prime. We illustrate two simple techniques which may be used to circumvent this restriction and make the constructions presented in this paper still relevant.

In Section 8, we presented six recursive multiplier constructions, all of which exhibit subquadratic asymptotic space complexities. Note that these multipliers are only a small fraction of the multipliers that can be obtained by using the constructions introduced in this paper. Nevertheless, one of these constructions obtained by extending a particular Winograd algorithm turns out to be identical to the Karatsuba algorithm. We note that two of the other four constructions and the Karatsuba algorithm require slightly more gates for practical values of the bit-length. On one hand, the time complexities of two of the multipliers are significantly better than that of the Karatsuba multiplier. The construction for $d = 3$ and $d = 5$ complements the Karatsuba multiplier since the Karatsuba algorithm is limited and may only be used when the length of the multiplication is a power of two.

Finally, we point out a possible disadvantage of the algorithms presented in this paper. When compared to the paper and pencil method, the algorithms with subquadratic complexity including the Karatsuba algorithm appear to have less structure. This irregularity may cause additional wire delays in actual VLSI implementation and may render the comparison in delay imprecise. We define a VLSI level performance evaluation of these algorithms and comparison to the paper and pencil method as future work.

ACKNOWLEDGMENTS

This material is based upon work supported by the US National Science Foundation under Grant No. ANI-0112889.

REFERENCES

- [1] R.E. Blahut, *Fast Algorithms for Digital Signal Processing*. Reading, Mass.: Addison-Wesley, 1984.
- [2] R.E. Blahut, *Theory and Practice of Error Control Codes*. Reading, Mass.: Addison-Wesley, 1983.
- [3] *Reed-Solomon Codes and Their Applications*, S.B. Wicker and V.K. Bhargava, eds. IEEE Press, 1994.
- [4] E.R. Berlekamp, *Algebraic Coding Theory*. New York: McGraw-Hill, 1968.

- [5] W.W. Peterson and E.J. Weldon Jr., *Error-Correcting Codes*. Cambridge, Mass.: MIT Press, 1972.
- [6] S.W. Golomb, *Shift Register Sequences*. San Francisco: Holden-Day, 1967.
- [7] R.J. McEliece, *Finite Fields for Computer Scientists and Engineers*, second ed. Kluwer Academic, 1989.
- [8] I.F. Blake, X.H. Gao, R.C. Mullin, S.A. Vanstone, and T. Yaghoobin, *Applications of Finite Fields*. Kluwer Academic, 1993.
- [9] W. Geiselmann and D. Gollmann, "Self-Dual Bases in F_q ," *Designs, Codes, and Cryptography* vol. 3, pp. 333-345, 1993.
- [10] S.T.J. Fenn, M. Benaissa, and D. Taylor, " $GF(2^m)$ Multiplication and Division over the Dual Basis," *IEEE Trans. Computers*, vol. 45, no. 3, pp. 319-327, Mar. 1996.
- [11] S.T.J. Fenn, M. Benaissa, and D. Taylor, "Finite Field Inversion over the Dual Basis," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 1, pp. 134-136, Mar. 1996.
- [12] E.D. Mastrovito, "VLSI Architectures for Computation in Galois Fields," PhD thesis, Dept. of Electrical Eng., Linköping Univ., Sweden, 1991.
- [13] E.D. Mastrovito, "VLSI Design for Multiplication over Finite Fields $GF(2^m)$," *Proc. Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC-6)*, pp. 297-309, Mar. 1989.
- [14] M.A. Hasan, M. Wang, and V.K. Bhargava, "Modular Construction of Low Complexity Parallel Multipliers for a Class of Finite Fields $GF(2^m)$," *IEEE Trans. Computers*, vol. 41, no. 8, pp. 962-971, Aug. 1992.
- [15] C. Paar, "Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields," PhD thesis (English translation), Inst. for Experimental Math., Univ. of Essen, Germany, June 1994.
- [16] Ç.K. Koç 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.
- [17] B. Sunar and Ç.K. Koç, "Mastrovito Multiplier for All Trinomials," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 522-527, May 1999.
- [18] B. Sunar and Ç.K. Koç, "An Efficient Optimal Normal Basis Type II Multiplier," *IEEE Trans. Computers*, vol. 50, no. 1, pp. 83-87, Jan. 2001.
- [19] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics-Doklady (English translation)*, vol. 7, no. 7, pp. 595-596, 1963.
- [20] S. Winograd, "Some Bilinear Forms Whose Multiplicative Complexity Depends on the Field of Constants," *Math. Systems Theory*, vol. 10, pp. 169-180, 1977.
- [21] S. Winograd, *Arithmetic Complexity of Computations*. SIAM, 1980.
- [22] M. Jacobson, A.J. Menezes, and A. Stein, "Solving Elliptic Curve Discrete Logarithm Problems Using Weil Descent," CACR Technical Report CORR2001-31, Univ. of Waterloo, May 2001.



Berk Sunar received the BSc degree in electrical and electronics engineering from Middle East Technical University in 1995 and the PhD degree in electrical and computer engineering (ECE) from Oregon State University in December 1998. After briefly working as a member of the research faculty at Oregon State University, he joined Worcester Polytechnic Institute as an assistant professor, where he currently heads the Cryptography and Information Security Laboratory (CRIS). He received the US National Science Foundation CAREER award in 2002. His research interests include finite fields, elliptic curve cryptography, low-power cryptographic hardware design, and computer arithmetic. He is a member of the IEEE Computer Society, the ACM, and the International Association of Cryptologic Research (IACR) professional societies.

► For more information on this or any computing topic, please visit our Digital Library at www.computer.org/publications/dlib.