A Fast Parallel Implementation of Elliptic Curve Point Multiplication over $GF(2^m)$

F. Rodríguez-Henríquez[a*], N. A. Saqib[a] and A. Díaz-Pérez[a].

[a]Computer Science Section, Electrical Engineering Department
Centro de Investigación y de Estudios Avanzados del IPN
Av. Instituto Politécnico Nacional No. 2508, México D.F.

A fast parallel architecture for the implementation of elliptic curve scalar multiplication over binary fields is presented. The proposed architecture is implemented on a single-chip FPGA device using parallel strategies that trades area requirements for timing performance. The results achieved show that our proposed design is able to compute $GF(2^{191})$ elliptic curve scalar multiplication operations in 63 μ Secs.

keywords: Finite Field Arithmetic, Elliptic Curve Cryptography, Karatsuba-Ofman Multipliers

1. Introduction

The security of currently used public-key cryptosystems is based on the computational complexity of an underlying mathematical problem, such as factoring giant numbers or computing discrete logarithms for large integers. These problems, without complete certainty, are believed to be very hard to solve. In practice, only a small number of mathematical structures could so far be applied to build public-key mechanisms. The majority of those structures are based on number theory, in particular on the multiplicative group of integers modulo a large number.

The theory of elliptic curves has been profusely studied in number theory and algebraic geometry for over 150 years. Initially pursued mainly for purely aesthetic reasons, elliptic curves have been lately utilized in primality proving, and public-key cryptography. Elliptic curve cryptosystems were first proposed in 1985 independently by N. Koblitz [1] and V. Miller [2]. Since then, an enormous amount of literature on this subject has been accumulated [3]. Although elliptic curves can be defined over real numbers, complex numbers, and any other field, from the cryptographic point of view, we are only concerned with elliptic curves defined over finite fields. Let $F_q = GF(2^m)$ be a finite field of characteristic two. A non-supersingular elliptic curve $E(F_q)$ is defined to be the set of

*Corresponding author. Tel. +(52 55) 5061-3800 Ext. 6570 Fax. +(52 55) 5061-3757 E-mail addresses: francisco@cs.cinvestav.mx (F. Rodríguez-Henríquez), nabbas@computacion.cs.cinvestav.mx (N. A. Saqib), adiaz@cs.cinvestav.mx (A. Díaz-Pérez)

points $(x, y) \in GF(2^m) \times GF(2^m)$ that satisfy the equation,

$$y^2 + xy = x^3 + ax^2 + b \tag{1}$$

where a and $b \in F_q, b \neq 0$, together with the point at infinity denoted by *0*.

Let $P = (x, y)$ be a point satisfying the above equation and let us define its inverse as $-P = (x, x + y)$. Then, the point $R = P + Q$ is defined as the point with the property that P, Q and $-R$ lie on a common line. The point at infinity plays the role of the *neutral element* for the addition. Thus,

$$\begin{aligned} P + O &= P\ , \\ P + (-P) &= O. \end{aligned} \tag{2}$$

For the special case when $P = Q$, the addition operation $2P = P + P$ is referred as *point doubling operation*. Since a quite long time, it has been known that the set of points satisfying the equation's curve (1) together with the just outlined point addition operation form an Abelian group [1].

Elliptic curve points can be added but not multiplied. It is, however, possible to perform *scalar multiplication*, which is another name for repeated addition of the same point. If n is a positive integer and P a point on an elliptic curve, the scalar multiple $Q = nP$ is the point resulting of adding $n - 1$ copies of P to itself. Scalar multiplication is by far the most important operation of elliptic curve cryptosystems.

Given two points Q and P that belong to the curve, the problem to find a positive scalar such that the equation $Q = kP$ holds, is referred as the discrete logarithm problem in this group. Solving the discrete logarithm problem over elliptic curves is believed to be an extremely hard mathematical problem, much harder than its analogous one defined over finite fields of the same size.

Due to the high difficulty to compute the discrete logarithm problem in elliptic curves over finite fields, one can obtain the same security provided by the other existing public-key cryptosystems, but at the price of much smaller fields, which automatically implies shorter key lengths. Having shorter key lengths means smaller bandwidth and memory requirements. This feature is especially valuable for some applications such as smart cards, where both memory and processing power are limited. For instance, nowadays it is widely accepted that a 160-bit key length elliptic curve cryptosystem provides about the same security that the one proportionate by a 1024-bit key length RSA cryptosystem.

Although elliptic curves can also be defined over fields of integers modulo a large prime number, $GF(P)$, for hardware implementations it is usually more advantageous to use finite fields of characteristic two, $GF(2^m)$. This is due largely to the carry-free binary nature exhibited by this type of fields, which is an especially important characteristic for hardware systems, yielding both higher performance and less area consumption.

Software implementations are not well suited for applications where time factor is vital like mobile communication, mobile electronic banking, etc. VLSI implementations tend to be fast but the associated cost of VLSI design and fabrication often proves to be too high for several applications. On the other hand, FPGA's implementations constitute an attractive option for designers, mainly because of their excellent cost/benefit tradeoff, high flexibility, possibility to optimize/reconfigure designs, etc.

To-date, fast implementations of elliptic curve scalar multiplication over $GF(2^m)$ have been reported both, in VLSI [4,5] and reconfigurable hardware [6–10] platforms.

The overall performance of any elliptic curve operation heavily depends on the efficiency of the underlying finite field arithmetic implementation. This is due to the fact that finite field arithmetic constitutes the main building block needed to obtain elliptic curve scalar multiplication as is shown in three-layer model shown in Figure 1.

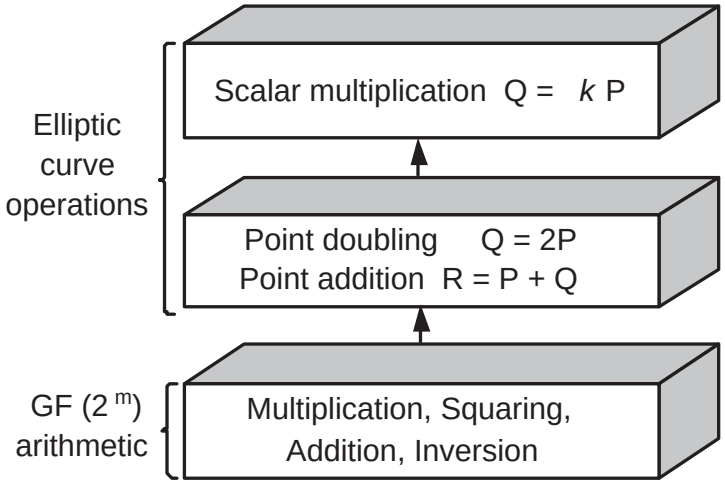

Figure 1. Three-Layer Model for Elliptic Curve Scalar Multiplication

In this paper we address the problem of how to implement efficiently finite field arithmetic in order to obtain a high performance computation of the elliptic curve scalar multiplication operation. We designed a parallel architecture able to compute $GF(2^{191})$ elliptic curve scalar multiplications. Our design was implemented on a Xilinx VirtexE 2600 device and the total computation time reported by the design was of only 63 μSecs.

The rest of this paper is organized as follows. In §2 we discuss several fully parallel designs for computing $GF(2^m)$ finite field arithmetic on reconfigurable hardware. In §3 we describe parallel strategies for the implementations of the second and third layers of Figure 1. In §4 FPGA implementation details of the main building blocks needed to compute elliptic curve point multiplication are given. In §5 we discuss the results obtained comparing them against other published works. Finally, in §6 some conclusions remarks as well as future work are given.

2. $GF(2^m)$ Finite Field Arithmetic

Arithmetic over $GF(2^m)$ has many important applications, in particular in the theory of error control coding and in cryptography [11–13]. Finite field's arithmetic operations include addition, subtraction, multiplication, and division. Addition and subtraction are equivalent operations in $GF(2^m)$. Addition in binary finite fields is defined as polynomial

addition and can be implemented simply as the XOR addition of the two m-bit operands.

Let $A(x), B(x)$ and $C'(x) \in GF(2^m)$ and $P(x)$ be the irreducible polynomial generating $GF(2^m)$. Multiplication in $GF(2^m)$ is defined as polynomial multiplication modulo the irreducible polynomial $P(x)$, $C'(x) = A(x)B(x) \bmod P(x)$. In order to obtain $C'(x)$, we can first obtain the product polynomial $C(x)$ of degree at most $2m-2$, as

$$C(x) = A(x)B(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} b_i x^i) \tag{3}$$

In a second step the reduction operation needs to be performed in order to obtain the $m-1$ degree polynomial $C'(x)$, which is defined as

$$C'(x) = C(x) \bmod P(x) \tag{4}$$

Notice that once that the irreducible polynomial $P(x)$ has been selected, the reduction step can be accomplished by using XOR gates only.

Hardware implementation efficiency of finite field arithmetic is measured in terms of the associated space and time complexities. Space complexity is defined as the total amount of hardware resources needed to implement the circuit, whereas time complexity is the total delay of the circuit.

In the rest of this section different implementation aspects and several efficient methods to compute $GF(2^m)$ finite field arithmetic are discussed. In § 2.1 and § 2.2 we study the problem of how to compute Eq. (3) efficiently, considering two separate cases. First, in section § 2.1 a variation of the classical Karatsuba-Ofman algorithm is analyzed as one of the most efficient techniques to find the polynomial product of Eq. (3). In § 2.2 we describe an efficient method to compute polynomial squaring in hardware, at a complexity cost of only $O(1)$. Finally, in § 2.3 we describe in detail a highly efficient hardware implementation that carries on the reduction step of Eq. (4).

2.1. Binary Karatsuba-Ofman Multipliers

Several architectures have been reported for multiplication in $GF(2^m)$. For example, efficient bit-parallel multipliers for both canonical and normal basis representation have been proposed in [14–17,10,18,19]. All these algorithms exhibit a space complexity $O(m^2)$. However, there are some asymptotically faster methods for finite field multiplications, such as the Karatsuba-Ofman algorithm [20]. Discovered in 1962, it was the first algorithm to accomplish polynomial multiplication in under $O(m^2)$ operations [21]. Karatsuba-Ofman multipliers may result in fewer bit operations at the expense of some design restrictions, particularly in the selection of the degree of the generating irreducible polynomial m.

In [22] was presented a Karatsuba multiplier based on composite fields of the type $GF((2^n)^s)$ with $m = sn$, $s = 2^t$, t an integer. However, for certain applications, quite particularly, elliptic curve cryptosystems, it is important to consider finite fields $GF(2^m)$ where m is not necessarily a power of two. In fact, for this specific application some sources [23] suggest that, for security purposes, it is strongly recommended to choose degrees m primes for finite fields in the range $[160, 512]$.

In fact, very few Karatsuba-Ofman multiplier variants have been reported so far for arbitrary field sizes [24,9]. To the best of our knowledge, none of them have provided theoretical complexities and total gate delays in terms of TAND and TXOR in a generic

way (i.e. for arbitrary field sizes m, especially for m a prime). References [24,9] are special purpose architectures addressing the specific cases GF(2233) and GF(2113), respectively, and they were optimized specifically for that field.

In the rest of this subsection we will briefly describe a variation of the classic Karatsuba-Ofman Multiplier called *binary Karatsuba-Ofman multipliers* that was first presented in [25]. Binary Karatsuba-Ofman multipliers can be utilized arbitrarily, regardless the form of the required degree m.

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ of degree $m = rn$, with $r = 2^k$, k an integer. Let A, B be two elements in $GF(2^m)$. Both elements can be represented in the polynomial basis as,

$$\begin{aligned} A &= \sum_{i=0}^{m-1} a_i x^i = \sum_{i=\frac{m}{2}}^{m-1} a_i x^i + \sum_{i=0}^{\frac{m}{2}-1} a_i x^i \\ &= x^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} a_{i+\frac{m}{2}} x^i + \sum_{i=0}^{\frac{m}{2}-1} a_i x^i = x^{\frac{m}{2}} A^H + A^L \end{aligned}$$

and

$$\begin{aligned} B &= \sum_{i=0}^{m-1} b_i x^i = \sum_{i=\frac{m}{2}}^{m-1} b_i x^i + \sum_{i=0}^{\frac{m}{2}-1} b_i x^i \\ &= x^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} b_{i+\frac{m}{2}} x^i + \sum_{i=0}^{\frac{m}{2}-1} b_i x^i = x^{\frac{m}{2}} B^H + B^L. \end{aligned}$$

Then, using last two equations, the polynomial product is given as

$$C = x^m A^H B^H + (A^H B^L + A^L B^H) x^{\frac{m}{2}} + A^L B^L. \tag{5}$$

Karatsuba-Ofman algorithm is based on the idea that the product of last equation can be equivalently written as,

$$\begin{aligned} C &= x^m A^H B^H + A^L B^L + (A^H B^H + A^L B^L + (A^H + A^L)(B^L + B^H)) x^{\frac{m}{2}} \\ &= x^m C^H + C^L. \end{aligned} \tag{6}$$

Let us define

$$\begin{aligned} M_A &:= A^H + A^L; \\ M_B &:= B^L + B^H; \\ M &:= M_A M_B. \end{aligned} \tag{7}$$

Using Eq. (6), and taking into account that the polynomial product C has at most $2m-1$ coordinates, we can classify its coordinates as,

$$\begin{aligned} C^H &= [c_{2m-2}, c_{2m-3}, \ldots, c_{m+1}, c_m]; \\ C^L &= [c_{m-1}, c_{m-2}, \ldots, c_1, c_0]. \end{aligned} \tag{8}$$

Although (6) seems to be more complicated than (5), it is easy to see that Eq. (6) can be used to compute the product at a cost of four polynomial additions and three polynomial multiplications. In contrast, when using Eq. (5), one needs to compute four polynomial

Input: Two elements $A, B \in GF(2^m)$ with $m = rn = 2^k n$, and
where A, B can be expressed as,
$A = x^{\frac{m}{2}} A^H + A^L, B = x^{\frac{m}{2}} B^H + B^L$.
Output: A polynomial $C = AB$ with up to $2m - 1$ coordinates,
where $C = x^m C^H + C^L$.
Procedure Kmul2^k(C, A, B)
0. begin
1. if $(r == 1)$ then
2. $C = mul\_n(A, B)$;
3. return;
4. for i from 0 to $\frac{r}{2} - 1$ do
5. $M_{Ai} = A_i^L + A_i^H$;
6. $M_{Bi} = B_i^L + B_i^H$;
7. end
8. $mul2^k(C^L, A^L, B^L)$;
9. $mul2^k(M, M_A, M_B)$;
10. $mul2^k(C^H, A^H, B^H)$;
11. for i from 0 to $r - 1$ do
12. $M_i = M_i + C_i^L + C_i^H$;
13. end
14. for i from 0 to $r - 1$ do
15. $C_{\frac{r}{2}+i} = C_{\frac{r}{2}+i} + M_i$;
16. end
17. end

Figure 2. $m = 2^k n$-bit Karatsuba-Ofman multiplier.

multiplications and three polynomial additions. Due to the fact that polynomial multiplications are in general much more expensive operations than polynomial additions, it is valid to conclude that (6) is computationally simpler than the classic algorithm. Karatsuba-Ofman's algorithm can be applied recursively to the three polynomial multiplications in (6). Hence, we can postpone the computations of the polynomial products $A^H B^H, A^L B^L$ and M, and instead we can split again each one of these three factors into three polynomial products. By applying this strategy recursively, in each iteration each degree polynomial multiplication is transformed into three polynomial multiplications with their degrees reduced to about half of its previous value.

Eventually, after no more than $\lceil \log_2(m) \rceil$ iterations, all the polynomial operands collapse into single coefficients. In the last iteration, the resulting bit multiplications can be directly computed. Although it is possible to implement the Karatsuba-Ofman algorithm until the $\lceil \log_2 m \rceil$ iteration, it is usually more practical to truncate the algorithm earlier. If the Karatsuba-Ofman algorithm is truncated at a certain point, the remaining multiplications can be computed by using alternative techniques (classic algorithm or other

techniques).

The algorithm presented in Figure 2 implements the Karatsuba-Ofman strategy for polynomial multiplication. By Combining the Karatsuba-Ofman algorithm with the classic algorithm, it can be shown [25] that the space and time complexities of the hybrid m-bit Karatsuba-Ofman multiplier truncated at the n-bit multiplicand level are upper bounded by

$$\begin{aligned} \#\text{ XORs} \leq &\quad \left(\tfrac{m}{n}\right)^{\log_2 3}(n^2+6n-1)-8m+2\ ; \\ \#\text{ ANDs} \leq &\quad 3^{\log_2 r} M_{and2^n} \;=\; \left(\tfrac{m}{n}\right)^{\log_2 3} n^2; \\ \text{Delay} \leq &\quad T_{AND}+T_X(\log_2 n+4\log_2 r)\ . \end{aligned} \tag{9}$$

Where T_X and T_{AND} correspond to an XOR gate delay and an AND gate delay, respectively. Table 1 shows the space and time complexities of Karatsuba-Ofman multipliers for the optimal case when m is a power of two. The values of m presented in Table 1 are the first eight powers of two. Various factors including target devices and design techniques influence the space complexity in terms of number of CLBs. The space complexity provided in the last column of Table 1 is based on our experimental results optimized for VirtexE devices only.

Table 1
Space and time complexities for several $m = 2^k$-bit hybrid Karatsuba-Ofman multipliers.

m	r	n	AND gates	XOR gates	Time delay	Area (in CLBs)
1	1	1	1	0	T_A	–
2	1	2	4	1	$T_X + T_A$	–
4	1	4	16	9	$2T_X + T_A$	8
8	2	4	48	55	$6T_X + T_A$	32
16	4	4	144	225	$10T_X + T_A$	115
32	8	4	432	799	$14T_X + T_A$	368
64	16	4	1296	2649	$18T_X + T_A$	1171
128	32	4	3888	8455	$22T_X + T_A$	3379

To generalize the Karatsuba-Ofman algorithm of Figure 2 for arbitrary degrees m, particularly m primes, let us consider the multiplication of two polynomials $A, B \in GF(2^m)$, such that their degree is less or equal to $m-1$, where $m = 2^k + d$. As a very first approach, we could pretend that both operands have 2^{k+1} coordinates each, where their respective $2^{k+1} - d$ most significant bits are all equal to zero. Figure 3 shows how the subpolynomials A^H and A^L will be redefined according with this approach. If we partition the operands A and B as shown in Figure 3, then, in order to compute their polynomial multiplication, we can use the regular Karatsuba-Ofman algorithm using $m = 2^{k+1}$. Although this approach is a valid one, it clearly implies the waste of several arithmetic operations, as some of the most significant bits of the operands are zeroes. However, if we were able to identify the extra arithmetic operations and remove them from the computation, we would then be able to find a quasi-optimal solution for arbitrary degrees of m. To see how this can be

$$A = [\overbrace{\underbrace{0,\ldots,0,0}_{2^{k+1}-d},a_{2^k+d-1},\ldots,a_{2^k+1},a_{2^k}}^{A^H},\overbrace{a_{2^k-1},a_{2^k-2},\ldots,a_2,a_1,a_0}^{A^L}];$$

$$A^H = [0,\ldots,0,0,a_{2^k+d-1},\ldots,a_{2^k+1},a_{2^k}];$$
$$A^L = [a_{2^k-1},a_{2^k-2},\ldots,a_2,a_1,a_0];$$

Figure 3. Binary Karatsuba-Ofman strategy

Input: Two elements $A, B \in GF(2^m)$ with m an arbitrary number, and where A, B can be expressed as $A = x^{\frac{m}{2}}A^H + A^L, B = x^{\frac{m}{2}}B^H + B^L$.
Output: A polynomial $C = AB$ with up to $2m-1$ coordinates, where $C = x^m C^H + C^L$.
Procedure mulgen_m(C, A, B)
0. begin
1. $k = \lfloor \log_2 m \rfloor$;
2. $d = m - 2^k$;
3. if $(d == 0)$ then
4. $C = Kmul2^k(A, B)$;
5. return;
6. for i from 0 to $d-1$ do
7. $M_{Ai} = A_i^L + A_i^H$;
8. $M_{Bi} = B_i^L + B_i^H$;
9. end
10. $mul2^k(C^L, A^L, B^L)$;
11. $mul2^k(M, M_A, M_B)$;
12. $mulgen\_d(C^H, A^H, B^H)$;
13. for i from 0 to $2^k - 2$ do
14. $M_i = M_i + C_i^L + C_i^H$;
15. end
16. for i from 0 to $2^k - 2$ do
17. $C_{k+i} = C_{k+i} + M_i$;
18. end
19. end

Figure 4. m-bit binary Karatsuba-Ofman multiplier.

done, consider the algorithm shown in Figure 4, which has been adapted from the one presented in previous Figure 2.

In lines 1-2 the values of the constants k, d such that $m = 2^k + d$ are computed. If $d = 0$, i.e, if m is a power of two, then the binary Karatsuba-Ofman algorithm of Figure 4 reverts to the specialized algorithm in Figure 2 presented in the previous section. If that is not the case, the algorithm of Figure 4 uses the constants k and d to prevent us to compute unnecessary arithmetic operations. In lines 6-9, the d least significant bits of M_A and M_B of Eq. (7) are computed using the d non-zero coordinates of A^H and B^H. The remaining $k-d$ most significant bits of M_A and M_B are directly obtained from A^L and B^L, respectively. Notice that the operands, A^L, B^L, M_A and M_B are all 2^k-bit polynomials. Because of that, our algorithm invokes the multiplier of Figure 2 in lines 10 and 11. On the other hand, both operands A^H and B^H are d-bit polynomials, where d, in general, is not a power of two. Consequently, in line 12, the algorithm calls itself in a recursive manner. This recursive call is invoked using the operand's degree reduced to d. In each iteration the degree of the operands gets reduced, and eventually, after a total of h iterations (where h is the hamming weight of the binary representation of the original degree m), the algorithm ends. As a design example, consider the binary Karatsuba-Ofman multiplier

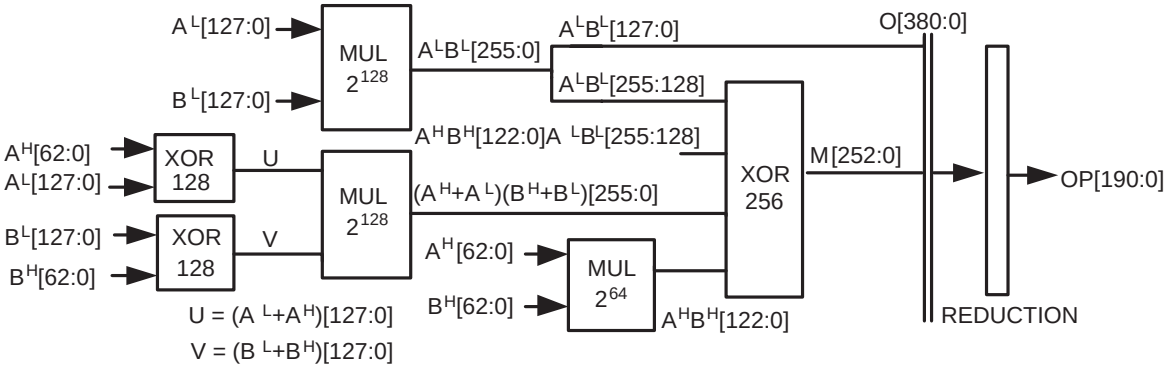

Figure 5. Karatsuba-Ofman Multiplier

shown in Figure 5. That circuit computes the polynomial multiplication of the elements A and $B \in GF(2^{191})$. Notice that for this case $m = 191 = 2^k + d = 2^7 + 63$. Since $(191)_2 = 10111111$, the Hamming weight h of the binary representation of m is $h = 7$. This implies that we would need a total of seven iterations in order to compute the multiplication using the generalized m-bit binary Karatsuba-Ofman multiplier. However we can do much better by assuming that the d $= 63$ most significant leftover bits are 64 (implying $m = (192)_2 = 11000000$). Hence, algorithm 4 can finish the computation in only two iterations, as shown in Figure 5. By using the complexity figures listed in Table 1, we can estimate the space and time complexities of the generalized 191-bit binary

Karatsuba-Ofman multiplier as,

$$\begin{array}{lcl} \text{Number of CLBs} & = & 2MUL_X(128) + MUL_X(64) + C \\ & = & 2 \cdot 3379 + 1171 + C \\ & = & 7929 + C. \\ \text{Delay} & = & MUL_{delay}(2^{\lfloor \log_2 m \rfloor}) + O \\ & = & MUL_{delay}(2^{\lfloor \log_2 191 \rfloor}) + O \\ & = & MUL_{delay}(2^7) + O \end{array} \tag{10}$$

Where C and O represent the overhead in space and time, respectively, associated with the extra circuitry shown in Figure 5. The generalized 191-bit binary Karatsuba-Ofman multiplier was implemented using Xilinx Foundation Series F4.1i software on Xilinx Virtex-E FPGA device XCV2600e-8bg560. The design is coded using VHDL, using library components and also by using Xilinx Coregenerator for design entry. The implementation occupied a total of 8721 slices and 576 I/O Blocks. We obtained a total path delay of 43 ηSec.

2.2. Squaring

In this section we investigate some efficient methods to compute polynomial squaring, which is a special case of polynomial multiplication. Let us assume that we have an element A given as $A = \sum_{i=0}^{m-1} a_i x^i$. Then the square of A is given as

$$C(x) \;=\; A(x)A(x) \;=\; A^2(x) \;=\; (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} a_i x^i) \;=\; \sum_{i=0}^{m-1} a_i x^{2i}. \tag{11}$$

The main implication of the above equation is that the first $k < m$ bits of A completely determine the first $2k$ bits of A^2. Notice also that half the bits of A^2 (the even ones) are zeroes. Taking advantage of this feature, the hardware implementation shown in Figure 6 simply interleaves a zero value between each one of the original bits of A yielding the required squaring computation. The implementation shown in Figure 6 has a computational complexity O(1), and hence its cost can be basically neglected.

2.3. Reduction

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ and let $A(x), B(x) \in GF(2^m)$. Assuming that we already have computed the product polynomial $C(x)$ of Eq. (3), by using any one of the methods described in the previous two subsections, we want to obtain the modular product C' of Eq. (4). Recall that the polynomial product C and the modular product C', have $2m-1$ and m, coordinates, respectively, i.e.,

$$\begin{array}{lcl} C & = & [c_{2m-2}, c_{2m-3}, \ldots, c_{m+1}, c_m, \ldots, c_1, c_0]; \\ C' & = & [c'_{m-1}, c'_{m-2}, \ldots, c'_1, c'_0]. \end{array} \tag{12}$$

Once the generating polynomial $P(x)$ has been selected, the reduction step that obtains C' from C can be computed by using XOR and shift operations only. Let the field $GF(2^m)$ be constructed using the irreducible trinomial $P(x) = x^m + x^n + 1$ with a root α and $1 < n < \frac{m}{2}$. Let also $A(x), B(x)$ be elements in $GF(2^m)$. In order to obtain the modular

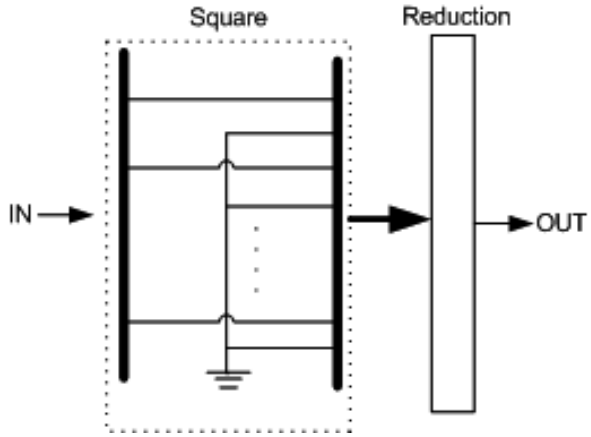

Figure 6. Squaring Circuit

product $C'(x)$ of (3), we use the property $P(\alpha) = 0$, and write

$$\begin{array}{rcl} \alpha^m & = & 1 + \alpha^n \ ; \\ \alpha^{m+1} & = & \alpha + \alpha^{n+1} \ ; \\ & \vdots & \\ \alpha^{2m-3} & = & \alpha^{m-3} + \alpha^{m+n-3} \ ; \\ \alpha^{2m-2} & = & \alpha^{m-2} + \alpha^{m+n-2} \ . \end{array} \tag{13}$$

The above $m-1$ set of identities suggests a method to obtain the m-coordinates of the modular product C' of Eq. (4). We can partially reduce the $2m-1$ coordinates of C by reducing its most significant $m-1$ bits into its first $m+n-1$ bits, as indicated by (13). For instance, in order to obtain the first partially reduced coordinate c'_0 we just need to add the regular product coordinate c_m to the c_0 coordinate, yielding c'_0 as $c'_0 \ = \ c_0 + c_m$.

Similarly the whole set of $m+n-2$ partially reduced coordinates can be found as,

$$\begin{array}{rcllllll} c'_0 & = & c_0 & + & c_m \ ; & & & \\ c'_1 & = & c_1 & + & c_{m+1} \ ; & & & \\ & \vdots & & & & & & \\ c'_{n-1} & = & c_{n-1} & + & c_{m+n-1} \ ; & & & \\ c'_n & = & c_n & + & c_{m+n} & + & c_m \ ; & \\ c'_{n+1} & = & c_{n+1} & + & c_{m+n+1} & + & c_{m+1} \ ; & \\ & \vdots & & & & & & \\ c'_{m-2} & = & c_{m-2} & + & c_{2m-2} & + & c_{2m-n-2} \ ; & \\ c'_{m-1} & = & c_{m-1} & & & + & c_{2m-n-1} \ ; & \\ c'_m & = & c_m & & & + & c_{2m-n} \ ; & \\ & \vdots & & & & & & \\ c'_{m+n-3} & = & c_{m+n-3} & & & + & c_{2m-3} \ ; & \\ c'_{m+n-2} & = & c_{m+n-2} & & & + & c_{2m-2} \ . & \end{array} \tag{14}$$

Notice that in the reduction process of (14), the constant coefficient of the irreducible generating trinomial $P(x)$ reflects its influence in the first $m-1$ partially reduced bits. The middle term of $P(x)$, on the other hand, affects the partially reduced bits of (14) in the range $[c'_n, c'_{m+n-2}]$.

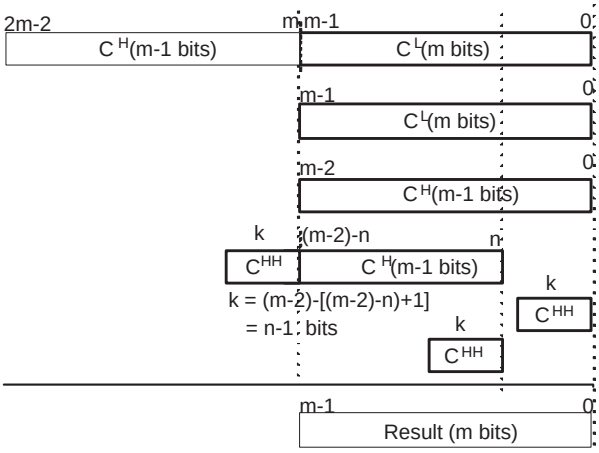

Figure 7. Reduction Diagram

Notice also that there is an overlap in the range $[c'_n, c'_{m-2}]$, where both the constant and the middle coefficients of $P(x)$ affect the partially reduced coordinates.

We say that the coefficients in (14) have been partially reduced because in general, if $n > 1$, we still need to reduce the $n-2$ most significant reduced coordinates of (14). However, this same idea can be used repeatedly until the $m-1$ modular coordinates of (12) are obtained. Each time that this strategy is applied we reduce $m-n$ coordinates.

Figure 7 shows how to implement on reconfigurable hardware the reduction strategy just outlined. As it was mentioned before, the reduction step involves XOR and overlap operations only.

Although the strategy shown in Figure 7 works for arbitrary irreducible trinomials, for the purposes of this research work we utilized a fixed irreducible generating trinomial, namely, $P(x) = x^{191} + x^9 + 1$.

3. Montgomery Point Multiplication

In this section we briefly discuss the algorithm used to compute point addition and point doubling and ultimately elliptic curve point multiplication. We follow the notation given in [26].

3.1. The Montgomery Algorithm

Let $P(x)$ be a degree-m polynomial, irreducible over $GF(2)$. Then $P(x)$ generates the finite field $F_q = GF(2^m)$ of characteristic two. A non-supersingular elliptic curve $E(F_q)$

is defined to be the set of points $(x, y) \in GF(2^m) \times GF(2^m)$ that satisfy the equation,

$$y^2 + xy \;=\; x^3 + ax^2 + b, \tag{15}$$

where a and $b \in F_q, b \neq 0$, together with the point at infinity denoted by *0*.

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve (15). Then $P + Q = (x_3, y_3)$ and $P - Q = (x_4, y_4)$, also belong to the curve and it can be shown that x_3 is given as [27],

$$x_3 = x_4 + \frac{x_1}{x_1 + x_2} + \left(\frac{x_1}{x_1 + x_2}\right)^2; \tag{16}$$

Hence we only need the x coordinates of P, Q and $P - Q$ to exactly determine the value of the x-coordinate of the point $P + Q$. Let the x coordinate of P be represented by X/Z. Then, when the points $2P = (X_{2P}, Y_{2P}, Z_{2P})$ and $P + Q = (X_3, Y_3, Z_3)$ are converted to projective coordinate representation their coordinates can be computed as [26],

$$\begin{aligned} X_{2P} &= X^4 + b \cdot Z^4; \\ Z_{2P} &= X^2 \cdot Z^2; \\ Z_3 &= (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2; \\ X_3 &= x \cdot Z_3 + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1); \end{aligned} \tag{17}$$

The algorithm shown in Figure 8 computes elliptic point multiplication over $GF(2^m)$

Input: $k = (k_{n-1}, k_{n-2}.....,k_1, k_0)_2$ with $k_{n-1} = 1$, $P(x, y) \in E(F_{2^m})$
Output: $Q = kP$

1. Set $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$
2. For i from $n - 2$ downto 0 do
3. if $(k_i = 1)$ then
4. Madd(X_1, Z_1, X_2, Z_2), Mdouble(X_2, Z_2)
5. else
6. Madd(X_2, Z_2, X_1, Z_1), Mdouble(X_1, Z_1)
7. Return$(Q = M_{xy}(X_1, Z_1, X_2, Z_2))$

Figure 8. Montgomery point multiplication

based on the formulae of Eq. (17) [27]. Its approximate running time is $6mM$ where M represents a field multiplication operation. In the next subsection we discuss how to obtain an efficient parallel implementation of the above algorithm.

3.2. Montgomery point multiplication: A parallel approach

As it was mentioned in the introduction section, we are mainly interested in parallel implementations of the three-layer architecture depicted in Figure 1. We already discussed

Table 2
$GF(2^m)$ Elliptic Curve Point Multiplication Computational Costs

Strategy		Required Number	EC Operation Cost		Total Number of
Second Layer	Third Layer	of Field Multipliers	Doubling	Addition	Field Multiplications
Sequential	Sequential	1	$2M$	$4M$	6mM
Sequential	Parallel	2	$2M$	$4M$	4mM
Parallel	Sequential	2	M	$2M$	3mM
Parallel	Parallel	4	M	$2M$	2mM

how to do so in the case of the first layer. However, in the case of the second and third layers we need to make a compromise as the required amount of hardware resources will soon become prohibited for a fully parallel implementation.

Table 2 presents the four different options that we can possibly follow in order to parallelize the algorithm of Figure 8. As is customary, the computational costs shown in Table 2 are normalized with respect to the required number of field multiplication operations.

Due to area restrictions we can afford to accommodate in our design, up to two fully-parallel Karatsuba-Ofman multipliers as the ones discussed in subsection 2.1. Hence, we have no option but to parallelize either the second or the third layer but not both. Although parallel implementation of the second layer seems to be more promising in terms of performance, for the sake of modularity we decided to go for the other design option. Hence, we devote one field multiplier for the implementation of the point addition building block and the other one for point doubling building block implementation. This way although in the second layer we are forced to have a sequential architecture, in the third layer we can save time by using the point addition and point doubling blocks in parallel.

Hence, we can estimate the expected theoretical time complexity of our computation of the elliptic curve scalar multiplication over $GF(2^{191})$ as the time required to perform a total of $4m = 4 * 191 = 764$ field multiplications. Once again, we stress that to our knowledge before this work only in reference [7] a theoretical complexity analysis for parallel approaches on elliptic curve multiplication has been reported.

In the next section we will discuss how the parallel approach outlined in the third raw of Table 2 was carried out on reconfigurable hardware.

4. FPGA Montgomery Point Multiplication Implementation

The kP computational time of Figure 8 heavily depends upon the efficient implementation of its two main building blocks: point addition (Madd) and point doubling (Mdouble). The remainder of this section is dedicated to describe the FPGA implementation of those two building blocks.

4.1. Point addition

Let $P(x, y) \in E(F_{2^m})$ be a point defined on the curve E, then the computation of point addition can be obtained from the execution of the sequence indicated in (18) that was

directly obtained from (17).

$$\begin{array}{ll} T & = x \\ p & = X_1 \times Z_2 \\ q & = Z_1 \times X_2 \\ r & = p + q \\ Z_3 & = r^2 \\ m & = p \times q \\ n & = T \times Z_3 \\ X_3 & = m + n \end{array} \tag{18}$$

Thus, the point addition computation consists of 4 multiplications, 2 additions and only one squaring. The FPGA implementation of Eq. 18 is shown in Figure 9.

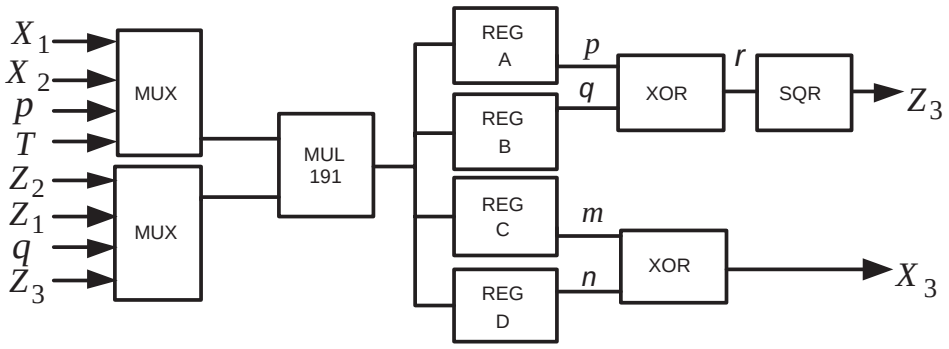

Figure 9. Point Addition

The MUL191 is a 191-bit fully-parallel multiplier based on the binary Karatsuba-Ofman strategy explained in section 2.1. Two multiplexers are used to switch the multiplier components and the resultant vectors are then saved in four registers REGA,REG,REGC and REGD. Finally an XOR operation is executed for final outputs, with additional squaring for Z component.

The architecture has been implemented on Xilinx VirtexE XCV2600 by using Xilinx Foundation Tool F4.1i. The design is coded in VHDL,using library components and also Xilinx Coregenerator Tool is used for design entry. It occupies 9389 (36%) CLB slices, and a single multiplication cycle is completed in 82.4 ηs, yielding a total timing performance of 330 ηs (four multiplication cycles) for the whole computation.

4.2. Point doubling

The computational complexity of Point doubling (Mdouble) is simpler than the one of point addition. The following equation is the sequence of instructions needed to compute

Table 3
Summary of Implemented designs

Design	Device	CLB slices	Timings
Field Multiplication $GF(2^{191})$	XCV2600E	8721	$82.4\eta s$
Point Doubling $GF(2^{191})$	XCV2600E	8241	$165\eta s$
Point Addition $GF(2^{191})$	XCV2600E	9389	$329.6\eta s$
Point Multiplication $GF(2^{191})$	XCV2600E	17630	$63\mu s$

a single point doubling operation.

$$\begin{aligned} T &= c \\ p &= X^2 \\ q &= Z^2 \\ Z_2 &= p \times q \\ l &= T \times q \\ m &= l^2 \\ n &= p^2 \\ X_2 &= m + n \end{aligned} \tag{19}$$

The algorithm consists of only 2 multiplications, 4 squarings and one addition. The FPGA implementation of Eq. 19 is shown in Figure 10.

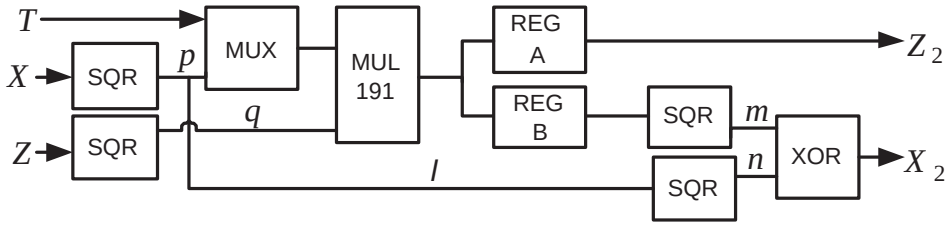

Figure 10. Point Doubling

Again a single 191-bit binary Karatsuba-Ofman multiplier is used here in combination with one multiplexer. There are four squaring blocks, however, as we discussed in subsection 2.2, squaring is a trivial operation on hardware implementations. The resultant vectors are saved in two registers REGA and REGB, which are further manipulated to get final X and Z components. The architecture is implemented using Xilinx Foundation Tool F4.1i. The design coding style remains the same, however, it occupies 8241(32%) CLB slices and one multiplication cycle is completed in 82.4 ηs, yielding a total timing performance of $165\eta s$ (two multiplication cycles) for the whole computation.

4.3. Point Multiplication

Table 3 shows the performance figures achieved for each one of the three main building blocks implemented in this paper and described in the precedent subsections. Those blocks can be combined as shown in Figure 11 to implement the Montgomery point multiplication algorithm of Figure 8.

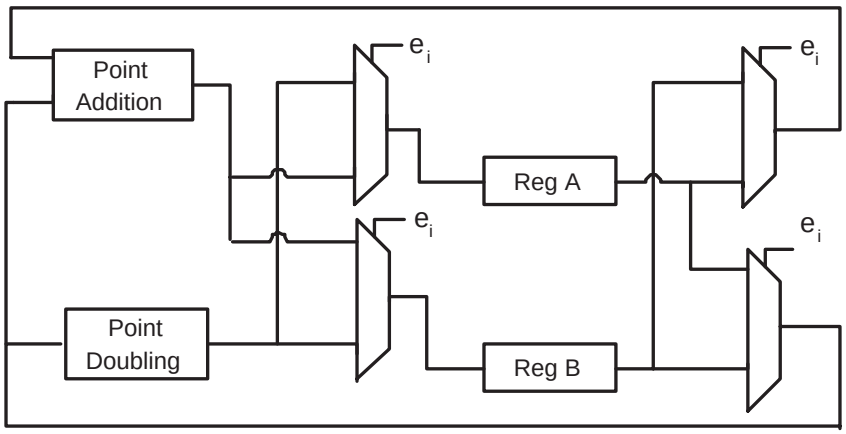

Figure 11. Montgomery Point Multiplication

5. Performance and Comparisons

Table 4 shows the hardware performance comparison for the computation of elliptic curve scalar multiplication over $GF(2^m)$. All the designs listed above have been published within the last three years and actually, most of them were published this same year. To our knowledge, the designs featured in Table 4 represent the state-of-the-art with respect to scalar multiplication performance on hardware implementations. The field sizes are all 160 bits or more. From that list, our design obtained the fastest computation time needing only 63μSecs to compute the scalar multiplication operation.

The usage of a relatively large FPGA target device is not the only reason why our design achieves a high timing performance, but we also spent quite a long time optimizing the critical path for the field multiplier presented here and the structures for point addition and point doubling computation of Figures 9 and 10, respectively. In fact, we are optimistic that the parallel approach followed in every design stage of this work can still be optimized to achieve even faster times.

Although our design has specifically targeted the field generated by the irreducible polynomial $P(x) = x^{191} + x^9 + 1$, all the machinery discussed in this paper has made no assumption about the specific field targeted and hence, can be easily adapted to accommodate other designs with different field sizes.

The design presented in [4] can handle arbitrary fields and elliptic curves without changing its hardware configuration, while those parameters have been fixed in our implemen-

Table 4
$GF(2^m)$ Elliptic Curve Point Multiplication Hardware Performance Comparison

Reference	Field	Platform	kP (μSecs)
[4]	$GF(2^{160})$	0.13μ CMOS ASIC	190
[6]	$GF(2^{167})$	Xilinx XCV400E	210
[7]	$GF(2^{191})$	Xilinx XCV4000XL	11820
[8]	$GF(2^{163})$	Xilinx XCV2000E	143
[8]	$GF(2^{193})$	Xilinx XCV2000E	187
[9]	$GF(2^{113})$	AT94K40	1400
[10]	$GF(2^{191})$	Xilinx XCV1000BG	270
This work	$GF(2^{191})$	Xilinx XCV2600E	63

tation. However the design in [4] was implemented on a traditional ASIC chip, where the flexibility for design changes is quite limited or many times even inexistent. In our approach on the other hand, taking advantage of the reconfigurability feature of the platform selected, we preferred to optimize the performance of our design for a given field while the possibility to reconfigure the design for other parameters can still be instrumented.

6. Conclusions

In this paper, the design of a fast elliptic curve point multiplication in $GF(2^{191})$ was presented. Our proposed architecture was designed on a Xilinx VirtexE XCV2600 FPGA device. To our knowledge our design showed the best timing published so far for elliptic curve scalar multiplication over binary fields. With a performance time of just $63\mu s$ our implementation is more than two times faster than any other work reported in the literature.

Occupying a total of 8721 slices and 576 I/O Blocks, our fully-parallel binary Karatsuba-Ofman multiplier is able to compute field multiplication operations over $GF(2^{191})$ at a speed of up to $43\eta s$. The binary Karatsuba-Ofman methodology outlined in this paper can be used for arbitrary field sizes regardless if the bit length m is a prime number or not. The remaining field operations, namely squaring and reduction are also implemented in the most general way, so that can be easily reconfigured to be used in designs with different field sizes.

The first reason why our design achieved a high performance is due to the saving on the total number of field operations required by our architecture. That was achieved mainly by using parallel structural arrangements for the computation of point addition and point doubling blocks corresponding to the layer 2 of the three-layer model of Figure 1. The details of the parallel strategies used in this work were summarized in Table 2.

Secondly, optimized field multiplier blocks based on fully parallel bit Karatsuba-Ofman multiplication contribute to achieve high performance too. In this work a variation of the standard Karatsuba-Ofman algorithm was utilized to allow field multiplication computation for arbitrary field sizes. Careful architectural considerations have been made to cut, as much as possible, the data critical paths for the said multipliers. Referring to Figure

5 the critical path for a $GF(2^{191})$ bit multiplier is essentially the critical path associated to the main building block utilized, namely the $GF(2^{128})$ Karatsuba-Ofman multiplier block (the time delay associated to extra circuitry can be neglected for most practical purposes).

The extensive usage of parallel strategies makes our design different from other architectures previously reported in literature. To our knowledge, only in reference [7] a fully parallel approach has been addressed.

As a whole, the elliptic curve scalar multiplication design presented here shows a good balance between required area and speed. Maybe the most important moral of this work is that today's FPGA technology has achieved a level good enough to accommodate fully-parallel designs of cryptographic algorithms such as elliptic curve cryptography.

Future work to be addressed for the authors includes further improvements in the performance of the algorithms, especially by reducing the critical path in all the three layers of the model, and also exploring other parallel strategies yet to be implemented.

REFERENCES

1. N. Koblitz, Elliptic curve cryptosystems, Mathematics of Computation 48(177) (1987) 203–209.
2. V. Miller, Uses of elliptic curves in cryptography, In H. C. Williams (editor) *Advances in Cryptology — CRYPTO 85 Proceedings,* Lecture Notes in Computer Science 218 (1985) 417–426.
3. D. V. Chudnovsky, G. V. Chudnovsky, Sequences of numbers generated by addition in formal groups and new primality and factorization tests, Advances in Applied Math. 7 (1986) 385–434.
4. A. Satoh, K. Takano, A scalable dual-field elliptic curve cryptographic processor, IEEE Transactions on Computers 52 (4) (2003) 449–460.
5. R. Schroeppel, C. Beaver, R. Gonzales, R. Miller, T. Draelos, A low-power design for an elliptic curve digital signature chip, Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers 2523 (2003) 366–380.
6. G. Orlando, C. Paar, A high-performance reconfigurable elliptic curve processor for $GF(2^m)$, Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings 1965 (2000) 41–56.
7. N. Smart, The Hessian form of an elliptic curve, Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings 2162 (2001) 118–125.
8. N. Gura, S. Shantz, H. E. et. al., An end-to-end systems approach to elliptic curve cryptography, Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers 2523 (2003) 349–365.
9. M. Ernst, M. Jung, F. Madlener, et. al., A reconfigurable system on chip implementation for elliptic curve cryptography over $GF(2^n)$, Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA,

USA, August 13-15, 2002, Revised Papers 2523 (2003) 381–399.
10. M. Bednara, M. Daldrup, J. Shokrollahi, J. Teich, J. von zur Gathen, Reconfigurable implementation of elliptic curve crypto algorithms, in: Proc. of The 9th Reconfigurable Architectures Workshop (RAW-02), Fort Lauderdale, Florida, U.S.A., 2002.
11. R. J. McEliece, Finite Fields for Computer Scientists and Engineers, Kluwer Academic Publishers, Boston, MA, 1987.
12. A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullen, S. A. Vanstone, T. Yaghoobian, Applications of Finite Fields, Kluwer Academic Publishers, Boston, MA, 1993.
13. S. B. Wicker, V. K. B. (editors), Reed-Solomon Codes and Their Applications, Prentice-Hall, Englewood Cliffs, NJ, 1994.
14. M. A. Hasan, M. Z. Wang, V. K. Bhargava, A modified Massey-Omura parallel multiplier for a class of finite fields, IEEE Transactions on Computers 42(10) (1993) 1278–1280.
15. B. Sunar, Ç. K. Koç, Mastrovito multiplier for all trinomials, IEEE Transactions on Computers 48(5) (1999) 522–527.
16. M. Morii, M. Kasahara, D. L. Whiting, Efficient bit-serial multiplication and the discrete-time Wiener-Hopf equation over finite fields, IEEE Transactions on Information Theory 35 (6) (1989) 1177–1183.
17. H. Wu, M. A. Hasan, Low complexity bit-parallel multipliers for a class of finite fields, IEEE Transactions on Computers 47 (8) (1998) 883–887.
18. T. Zhang, K. Parhi, Systematic design of original and modified mastrovito multipliers for general irreducible polynomials, IEEE Transactions on Computers 50 (7) (2001) 734–749.
19. A. Halbutogullari, Ç. K. Koç, Mastrovito multiplier for general irreducible polynomials, IEEE Transactions on Computers 49 (5) (2000) 503–518.
20. C. Paar, A new architecture for a parallel finite field multiplier with low complexity based on composite fields, IEEE Transactions on Computers 45(7) (1996) 856–861.
21. E. Bach, J. Shallit, Algorithmic number theory, Volume I: efficient algorithms, Kluwer Academic Publishers, Boston, MA, 1996.
22. J. Guajardo, C. Paar, Efficient algorithms for elliptic curve cryptosystems, In B. S. Kaliski Jr. (editor) *Advances in Cryptology —CRYPTO 97* Lecture Notes in Computer Science 1294 (1997) 342–356.
23. I. P1363, Standard specifications for public-key cryptography, draft version 7 Edition, IEEE standards documents, "http://grouper.ieee.org/groups/1363/", 1998.
24. C. Grabbe, M. Bednara, et. al., FPGA designs of parallel high performance gf(2^{233}) multipliers, in: IEEE International Symposium on Circuits and Systems (ISCAS-03), Bangkok, Thailand, 2003, pp. 268–271.
25. F. Rodríguez-Henríquez, Ç. K. Koç, On fully parallel karatsuba multipliers for $GF(2^m)$, in: International Conference on Computer Science and Technology (CST 2003), Cancun, Mexico, 2003.
26. J. Lopez, R. Dahab, Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation, Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings 1717 (1999) 316–327.
27. D. Hankerson, J. Lopez-Hernandez, A. Menezes, Software implementation of elliptic

curve cryptography over binary fields, Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings 1965 (2000) 1–24.