A Reconfigurable Processor for High Speed Point Multiplication in Elliptic Curves

Nazar A. Saqib, Francisco Rodríguez-Henriquez, and Arturo Díaz-Pérez

***Abstract*— This paper presents a generic architecture for the computation of elliptic curve scalar multiplication over binary extension fields. In order to optimize the performance as much as we could, we designed a parallelized version of the well-known Montgomery point multiplication algorithm implemented on a reconfigurable hardware platform (Xilinx XCV3200). The proposed architecture allows the computation of the main building blocks required by the Montgomery algorithm in an efficient manner. The results achieved show that our proposed design is able to compute $GF(2^{191})$ elliptic curve scalar multiplication operations in just 22 clock cycles at a frequency of about 10MHz. Moreover, our structure is able to obtain a scalar multiplication in less than 60 μSecs.**

***Index Terms*— Elliptic Curve Cryptography, Reconfigurable Hardware, Finite Field Arithmetic.**

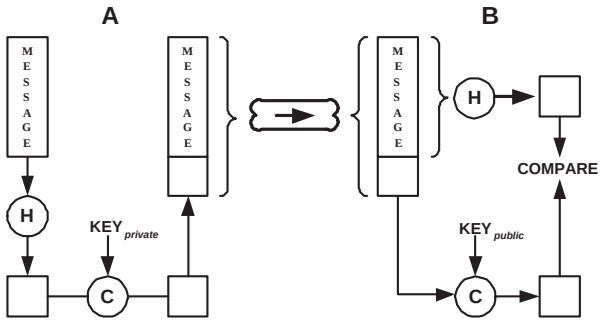

Fig. 1. A general digital signature scheme

I. INTRODUCTION

Digital signatures provide to an electronic document three security features. These features are: data integrity, authentication and non-repudiation. Successful verification of a digital signature ensures the recipient that the document received is identical to the document sent (i.e., *data integrity*) and confirms the identity of the sender (i.e., *authentication*). It also prevents any subsequent denial by the sender that the document was not originated from his/her side (i.e., *non-repudiation*).

In cryptography, digital signature schemes relay on public key encryption. A general digital signature scheme is depicted in Fig. 1. The document to be signed is first processed by a hash function in order to produce a unique message digest. Afterwards, that digest is signed using sender's private key. The signed digest along with the original document are then sent. At the receiver's side, the signature legitimacy can be verified by comparing the digest of the received message with the one obtained by decrypting the received signed hash using sender's public key.

Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analogue of the Digital Signature Algorithm (DSA). In 1999 it was named an ANSI standard, and in 2000 it was accepted as IEEE and NIST standards. Unlike the ordinary discrete logarithm problem and the integer factorization problem, no subexponential-time algorithm is known for the elliptic curve discrete logarithm problem. For this reason, the strength-per-key-bit is substantially greater in elliptic curve cryptographic schemes.

The authors are with the Computer Science Section, Electrical Engineering Department, Centro de Investigación y de Estudios Avanzados del IPN, Av. Instituto Politécnico Nacional No. 2508, México D.F.

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. These characteristics are especially important in some applications such as smart cards, where both memory and processing power are limited or in applications were extremely high levels of security are required.

Furthermore, and in deep contrast with most of the previous public-key cryptosystems which are inspired in the application of a set of number-theory problems, the elliptic curve cryptosystem is the first major cryptographic scheme that incorporates and takes advantage of the concepts of the Galois field algebra, by using elliptic curves defined over binary extension finite fields.

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

Elliptic curve primitives can also be utilized for running security protocols such as the well-known Diffie-Hellman key exchange protocol shown in Fig. I. Let A and B agree on an elliptic curve E over a large finite field F and a point P on that curve. Then the steps to exchange a secret key by using elliptic curve discrete logarithmic algorithm are as follows,

- A and B each privately choose large random integers, denoted r_1 and r_2.

- Using elliptic curve point-addition, A computes r_1P on E and sends it to B. B computes r_2P on E and sends it to A.
- Both A and B can now compute the point r_1r_2P, A by multiplying the received value of n_2P by his/her secret number n_1, and vice-versa for B.

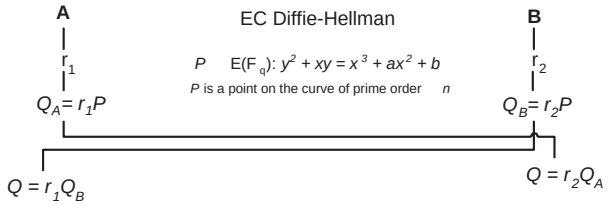

Fig. 2. Elliptic Curve Diffie-Hellman (ECDH)

The protocol of Fig. I is considered secure against passive attacks.

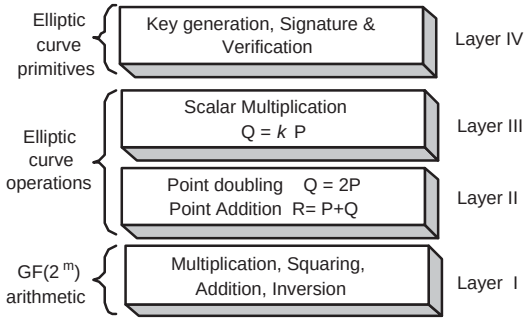

Fig. 3. Four-Layer Model for Elliptic Curve Scalar Multiplication

The most important operation for elliptic curve cryptosystems is then the so-called *Scalar multiplication* operation that can be unformally defined as follows. Let n be a positive integer and P a point on an elliptic curve. Then the scalar multiple $Q = nP$ is the point resulting of adding $n-1$ copies of P to itself. The security of elliptic curve systems is based on the intractability of the elliptic curve discrete logarithm problem (ECDLP) that can be formulated as follows. Given an elliptic curve E defined over a finite field F_q and two points Q and P that belong to the curve, where P has order r, find a positive scalar $n \in [0, r-1]$ such that the equation $Q = nP$ holds.

ECDSA can be implemented using the four-layer hierarchical strategy depicted in Fig. I. Each one of the four layers included in Fig. I are suitable for being implemented using parallel or semi-parallel approaches. Ultimately, high performance implementations of elliptic curve cryptography depend heavily on the efficiency in the computation of the underlying finite field arithmetic operations needed for the elliptic curve operations.

Since ECC proposal in 1985, a vast amount of investigation has been carried out in order to obtain efficient ECC implementations [1]. Numerous fast implementations have been reported in all of them, software [2], [3], [4], VLSI [5], [6] and reconfigurable hardware [7], [8], [9], [10], [11] platforms with variable time and space performance results.

This paper is the extened version of our article in [12]. The main focus of this paper is the study and analysis of efficient reconfigurable hardware algorithms suitable for the implementation of finite field arithmetic. This focus is crucial for a number of security and efficiency aspects of cryptosystems based on finite field algebra, and especially relevant for elliptic curve cryptosystems.

The rest of this paper is organized as follows. In §II we describe parallel strategies for the implementations of the second and third layers of Fig. I, i.e., point addition and point doubling operations based on the algorithm devised by [2]. §III describes an architectural design for the implementation of elliptic curve scalar multiplication and parallel techniques for implementing finite field arithmetic in $GF(2^{191})$. FPGA's implementation of elliptic curve scalar multiplication and implementation results are also presented in this section. Performance comparison is made in §VI. Finally, in §VII some conclusions remarks as well as future work are drawn. References are presented at the end.

II. 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 affine 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 O. The elliptic curve group law in affine coordinates is given by:

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve 1 then $-P = (x_1, x_1+y_1)$. For all P on the curve $P+O = O+P = P$. If $Q \neq -P$, then $P+Q = (x_3, y_3)$, where

$$x_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a & P \neq Q \\ x_1^2 + \frac{b}{x_1^2} & \text{P = Q} \end{cases} \tag{2}$$

$$y_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})(x_1 + x_3) + x_3 + y_1 & P \neq Q \\ x_1^2 + (x_1 + \frac{y_1}{x_1})x_3 + x_3 & \text{P = Q} \end{cases} \tag{3}$$

It can be seen from Eqns. 2 and 3 that for both of them, point addition (when $P \neq -Q$) and point doubling (when $P = Q$), the computations for (x_3, y_3) require one field inversion and two field multiplications neglecting the costs of field additions and squarings. An important observation first made by Montgomery is that the x-coordinate of $2P$ does not involve the y-coordinate of P.

Projective Coordinates: Compared with field multiplication in affine coordinates, inversion is by far the most expensive basic arithmetic operation in $GF(2^m)$. Inversion can be avoided by means of projective coordinate representation. A point P in projective coordinates is represented using three coordinates X, Y, and Z. This representation greatly helps to reduce internal computational operations. It is customary to

convert the point P back from projective to affine coordinates in the final step. This is due to the fact that affine coordinate representation involves the usage of only two coordinates and therefore is more useful for external communication saving some valuable bandwidth.

In *standard* projective coordinates the projective point (X:Y:Z) with Z$\neq$ 0 corresponds to the affine coordinates $x = X/Z$ and $y = Y/Z$. The projective equation of the elliptic curve is given as:

$$Y^2Z + XYZ = X^3 + aX^2Z + bZ^3 \tag{4}$$

A. Montgomery Group Law

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve of Equation 1. 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 [3],

$$x_3 = x_4 + \frac{x_1}{x_1 + x_2} + \left(\frac{x_1}{x_1 + x_2}\right)^2; \tag{5}$$

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 point $2P = (X_{2P}, Y_{2P}, Z_{2P})$ is converted to projective coordinate representation, it becomes [2],

$$\begin{aligned} x_{2P} &= X^4 + b \cdot Z^4; \\ z_{2P} &= X^2 \cdot Z^2; \end{aligned} \tag{6}$$

The computation of Eq. 6 requires one general multiplication, one multiplication by the constant b, five squarings and one addition. Fig. 4 is the sequence of instructions needed to compute a single point doubling operation $Mdouble(X_1, Z_1)$ effciently.

Input: $P = (X_1, -, Z_1) \in E(F_{2^m}), c$ such that $c^2 = b$
Output: $P = 2 \cdot P$
Procedure: Mdouble(X_1, Z_1)

1. $T = X_1^2$
2. $M = c \cdot Z_1^2$
3. $Z_1 = T \cdot Z_1^2$
4. $M = M^2$
5. $T = T^2$
6. $X_1 = T + M$

Fig. 4. Montgomery point doubling

In a similar way, the coordinates of $P + Q$ in projective coordinates can be computed as the fraction X_3/Z_3 and are given as:

$$\begin{aligned} Z_3 &= (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2; \\ X_3 &= x \cdot Z_3 + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1); \end{aligned} \tag{7}$$

The required field operations for point addition of Eq. 7 are three general multiplications, one multiplication by x, one squaring and two additions. This operation can be efficiently implemented as shown in Fig. 5.

Input: $P = (X_1, -, Z_1), Q = (X_2, -, Z_2) \in E(F_{2^m})$
Output: $P = P + Q$
Procedure: Madd(X_1, Z_1, X_2, Z_2)

1. $M = (X_1 \cdot Z_2) + (Z_1 \cdot X_2)$
2. $Z_1 = M^2$
3. $N = (X_1 \cdot Z_2) \cdot (Z_1 \cdot X_2)$
4. $M = x \cdot Z_3$
5. $X_1 = M + N$

Fig. 5. Montgomery point addition

B. Montgomery Point Multiplication

A method based on the formulas for doubling (from Eq. 6) and for addition (from Eq. 7) is shown in Fig. 6 [2]. Notice that steps 2.2 and 2.3 are formulae for point doubling (Mdouble) and point addition (Madd) from Figs. 6 and 7 respectively. In fact both Mdouble and Madd operations are executed in each iteration of the algorithm. If the test bit k_i is '1', the manipulations are made for Madd(X_1, Z_1, X_2, Z_2) and Mdouble(X_2, Z_2) (step 2.2) else Madd(X_2, Z_2, X_1, Z_1) and Mdouble(X_1, Z_1) i-e Mdouble and Madd with reversed arguments (step 2.3).

The approximate running time of the algorithm shown in Fig. 6 is $6m$M + ($1I$ + 10M) where M represents a field multiplication operation, m stands for the number of bits and I corresponds to inversion. It is to be noted that the factor ($1I$ + 10M) represents time needed to convert from standard projective to affine coordinates. In the next subsection we discuss how to obtain an efficient parallel implementation of the above algorithm quite especially for step 2.

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: $Q = kP$

Procedure: MontPointMult(P, k)
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
   2.1 if $(k_i = 1)$ then
   $Madd(X_1, Z_1, X_2, Z_2)$;
   $Mdouble(X_2, Z_2)$;
   2.2 else
   $Madd(X_2, Z_2, X_1, Z_1)$;
   $Mdouble(X_1, Z_1)$;
3. $x_3 \leftarrow X_1/Z_1$
4. $y_3 \leftarrow (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) +$
   $(x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y$
5. Return (x_3, y_3)

Fig. 6. Montgomery point multiplication

C. Conversion from Standard Projective (SP) to Affine Coordinates

Both, point addition and point doubling algorithms are presented in standard projective coordinates. A conversion

process is therefore needed from SP to affine coordinates. Referring to the algorithm of Fig. 6, the corresponding affine x-coordinate is obtained in step 3: $x_3 = X_1/Z_1$. Whereas the affine representation for the y-coordinate is computed by step 4: $y_3 \leftarrow (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y$. Notice also that both expressions for x_3 and y_3 in affine coordinates include one inversion operation. Although this conversion procedure must be performed only once in the final step, still it would be useful to minimize the number of inversion operations as much as possible. Fortunately it is possible to reduce one inversion operation by using the common operations from the conversion formulae for both x and y-coordinates. A possible sequence of the instructions from SP to affine coordinates is given by the algorithm in Fig. 7.

Input: $P = (X_1, Z_1)$, $Q = (X_2, Z_2) \in E(F_{2^m})$
$P(x, y) \in E(F_{2^m})$
Output:(x_3, y_3) affine coordinates
Procedure: $SPtoAffine(X_1, Z_1, X_2, Z_2)$

1. $\lambda_1 = Z_1 \times Z_2$
2. $\lambda_2 = Z_1 \times x$
3. $\lambda_3 = \lambda_2 + X_1$
4. $\lambda_4 = Z_2 \times x$
5. $\lambda_5 = \lambda_4 + X_1$
6. $\lambda_6 = \lambda_4 + X_2$
7. $\lambda_7 = \lambda_3 \times \lambda_6$
8. $\lambda_8 = x^2 + y$
9. $\lambda_9 = \lambda_1 \times \lambda_8$
11. $\lambda_{10} = \lambda_7 + \lambda_9$
12. $\lambda_{11} = x \times \lambda_1$
13. $\lambda_{12} = inverse(\lambda_{11})$
13. $\lambda_{13} = \lambda_{12} \times \lambda_{10}$
14. $x_3 = \lambda_{14} = \lambda_5 \times \lambda_{12}$
15. $\lambda_{15} = \lambda_{14} + x$
16. $\lambda_{16} = \lambda_{15} \times \lambda_{13}$
17. $y_3 = \lambda_{16} + y$

Fig. 7. Standard Projective to affine coordinate

The coordinate conversion process makes use of 10 multiplications and only 1 inversion ignoring addition and squaring operations.

The algorithm in Fig. 7 includes one inversion operation which can be performed using Extended Euclidean Algorithm or Fermat's Little Theorem (FLT).

D. Montgomery point multiplication: A parallel approach

As it was mentioned earlier in the introduction section, parallel implementations of the three-layer architecture depicted in Fig. I constitutes the main interest of this paper. We will briefly discuss how to do so in the case of the first layer in §IV. However, hardware resource limitations restrict us from attempting a fully parallel implementation of second and third layers. Thus, a compromising strategy must be adopted to exploit parallelism at second and third layers. Several options to do so are shown in Table I.

TABLE I
$GF(2^m)$ ELLIPTIC CURVE POINT MULTIPLICATION COMPUTATIONAL COSTS

Strategy		Required No. of Field Multipliers	EC Operation Cost		Total Number of Field Multiplications
2nd Layer	3rd Layer		Doubling	Addition	
S	S	1	$2M$	$4M$	6mM
S	P	2	$2M$	$4M$	4mM
P	S	2	M	$2M$	3mM
P	P	4	M	$2M$	2mM

Table I presents four different options that we can possibly follow in order to parallelize the algorithm of Fig. 6. As is customary, the computational costs shown in table I 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 field multipliers. Hence, we have no option but to parallelize either the second or the third layer but not both, meaning that the first and fourth options are therefore out of the scope of this work. Both second and third options seem to be feasible; however, third option proves to be more attractive as it demonstrates better timing performance at the same area cost as compared to the second option. As it is indicated in the third row of Table I, the estimated computational cost of our elliptic curve Point multiplication implementation will be of only $3m$ field multiplications.

In next section we discuss how this approach can be carried out on hardware platforms.

III. ARCHITECTURAL DESIGN

In this section, a generic parallel architecture for computing elliptic curve scalar multiplication on hardware platforms is described. The proposed architecture is based on a parallel-sequential approach of the Montgomery algorithm of Fig. 6, discussed in the previous section. That approach corresponds to the one outlined in the third row of Table I.

Fig. 8 represents a generic architecture proposed for elliptic curve scalar multiplication by implementing Eqs. 4 and 5 in just three clock cycles.

The parallel architecture shown in Fig. 8 utilizes two multipliers in $GF(2^m)$ together with two memory blocks to retrieve and store operands. A two-port RAM is the suggested configuration for the memory blocks. This configuration makes possible to have two independent data accesses simultaneously. Some combinational logic units comprising squarer units, XOR operators and multiplexers are located before or after the two multipliers to perform pre or post computations.

Referring to the algorithm of Fig. 6, each bit of vector k is tested from left to right (in descending order). Both Madd and Mdouble operations are executed irrespective of the test-bit to be zero or one. However, order of the arguments is reversed: if the test bit is '1', Mdouble(X_2, Z_2), Madd(X_1, Z_1, X_2, Z_2) else Mdouble(X_1, Z_1), Madd(X_2, Z_2, X_1, Z_1) are computed. The computations of the algorithms for Madd and Mdouble are made as described in Figs. 4 and 5.

Table II demonstrates the algorithm flow to complete both point addition and point doubling operations in three normal

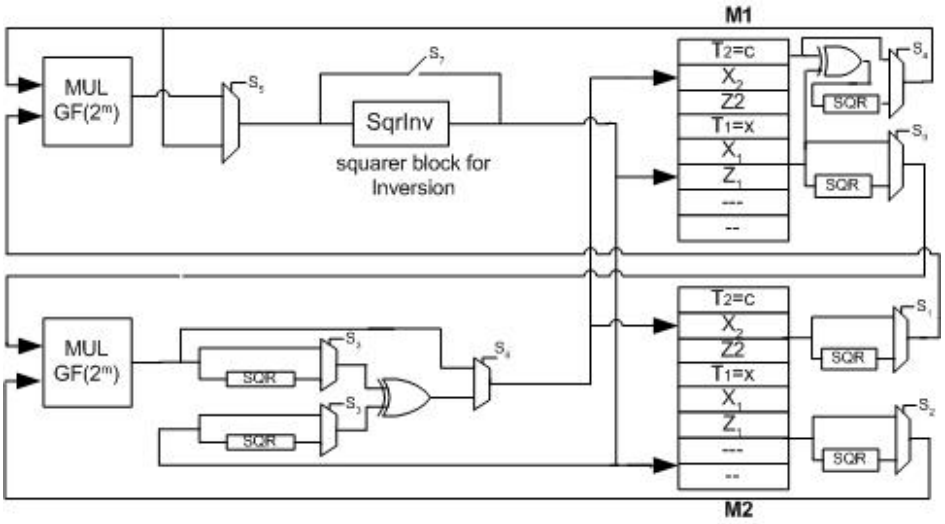

Fig. 8. kP scalar multiplication

clock cycles for the case when test-bit is one. M1 and M2 represent two memory blocks each one with two input ports $PT1$ and $PT2$. The notations used in Table II and III are the same that were used in Figs. 4 and 5. It must be noted that the modular products computed by the multipliers are not always directly stored in the RAMs as some arithmetic operations (squaring etc.) need to be performed for the next steps. Similarly, during reading cycle arithmetic operations are executed t some inputs to the multipliers.

TABLE II
kP COMPUTATION, IF TEST-BIT IS '1'

Cycle	Read				Write	
	M1		M2		M1/M2	
	$PT1$	$PT2$	$PT1$	$PT2$	$PT1$	$PT2$
1	X_1	Z_2	Z_1	X_2	P	Q
2	X_2	Z_2	Z_2	T_1	Z_2=Z_3	X_2=X_3
3	P	Q	Q	T_2	X_1=X'	Z_1=Z'

Table III details all computational steps needed to execute multiplications in three normal cycles for the case when test-bit is zero. Only the order of the arguments is reversed. The resultant vectors X_1,Z_1,X_2,Z_2, are updated at the completion of point addition and doubling operations after each 3 clock cycles for each test-bit. Total time for whole 191-bit test vector is therefore $191 \times 3 \times T$, where T represents the period of the maximum clock frequency allowed.

TABLE III
kP COMPUTATION, IF TEST-BIT IS '0'

Cycle	Read				Write	
	M1		M2		M1/M2	
	$PT1$	$PT2$	$PT1$	$PT2$	$PT1$	$PT2$
1	X_2	Z_1	Z_2	X_1	P	Q
2	X_1	Z_1	Z_1	T_1	Z_1=Z_3	X_1=X_3
3	P	Q	Q	T_2	X_2=X'	Z_2=Z'

An inversion operation is performed once at the end for the conversion of SP to affine coordinates. That operation uses building blocks for multiplication and squaring in $GF(2^m)$. One of the multiplier block MUL $GF(2^m)$ is used for the multiplication. A squarer block 'SqrInv' is especially added for the inversion only. As it is earlier explained that $m-1$ squarings are performed to complete one inversion in $GF(2^m)$. It takes $m-1$ clock cycles by using a single squarer block. However several squarer blocks can be cascaded to perform more than one squaring operation in the same clock cycle. That would be a useful approach for performing all squaring operations using a few clock cycles.

The generic architecture of Fig. 8 can be extended for implementations of elliptic curve scalar multiplication on hardware platforms like FPGAs, VLSI, with minor modifications. In the rest of this section, it is explained how the proposed architecture can be optimized for FPGA devices to achieve high timing performances.

IV. Finite Field Arithmetic

In this subsection we address the problem of how to implement efficiently finite field operations in reconfigurable hardware. In particular, we briefly describe how to implement two of the blocks shown in Fig. 8: field multiplication squaring and inversion. This section is mostly based on the work in [13]

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$, of degree m, and let A, B be two elements in $GF(2^m)$ given in the polynomial basis as $A = \sum_{i=0}^{m-1} a_i x^i$ and $B = \sum_{i=0}^{m-1} b_i x^i$, respectively, with $a_i, b_i \in GF(2)$. By definition, the field product $C' \in GF(2^m)$ of the elements $A, B \in GF(2^m)$ is given as

$$C'(x) = A(x)B(x) \bmod P(x) \tag{8}$$

In this work, Eq. 8 is computed into two steps: polynomial multiplication followed by modular reduction.

Let $A(x), B(x), C'(x) \in GF(2^m)$ and $P(x)$ be the irreducible polynomial generating $GF(2^m)$. In order to compute (8) we first obtain the product polynomial $C(x)$ of degree at most $2m-2$, as

$$C(x) = A(x)B(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} b_i x^i) \quad (9)$$

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

$$C'(x) = C(x) \mod P(x) \quad (10)$$

In the rest of this subsection we show how to compute Eq. 9 efficiently, considering two separate cases. First, we describe an efficient method to compute polynomial squaring, which is a particular case of polynomial multiplication. Then, a practical reconfigurable hardware implementation of Karatsuba-Ofman algorithm is briefly analyzed as one of the most efficient techniques to find the polynomial product of (9). Finally we discuss how to implement in a highly efficient way the reduction step of Eq. 10.

A. *Squaring over* $GF(2^m)$

Polynomial squaring over $GF(2)$ is a special case of polynomial multiplication, generally considered a costly operation in software. However in hardware platforms it is free of cost as it can be implemented occupying almost none hardware resource.

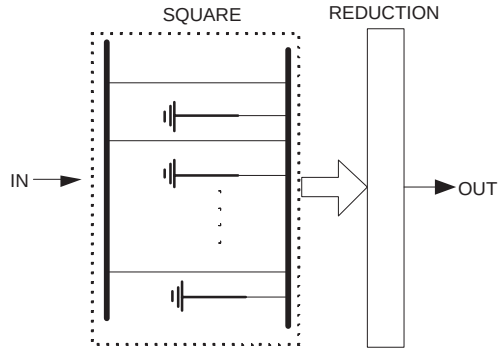

Fig. 9. Squaring Circuit

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

$$\begin{aligned} C(x) = A(x)A(x) &= A^2(x) \\ &= (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} a_i x^i) \\ &= \sum_{i=0}^{m-1} a_i x^{2i}. \end{aligned}$$

From the above equation, we immediately conclude 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 odd ones) are zeroes. This property happens to benefit hardware implementations as computation for polynomial squaring can be accomplished by just placing a zero value (connection to ground) at each alternative position of the original bits as shown in Fig. 9. The implementation has a computational complexity O(1), hence its cost can be neglected.

B. *Multiplication over* $GF(2^m)$

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], [15], [16], [17], [11]. 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. Discovered in 1962, it was the first algorithm to accomplish polynomial multiplication in under $O(m^2)$ operations [18]. 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 this research work we utilized a variation of the classic Karatsuba-Ofman Multiplier called *binary Karatsuba multipliers* that was first presented in [19]. Binary Karatsuba multipliers can be efficiently utilized regardless the form of the required degree m.

Let us consider the multiplication of two polynomials $A, B \in GF(2^m)$ with $m = rn = 2^k n$ (n is an integer), where A and B can be expressed as $A = x^{\frac{m}{2}} A^H + A^L$ and $B = x^{\frac{m}{2}} B^H + B^L$, then by using classical Karatsuba multiplier, the product of A and B can be expressed as:

$$\begin{aligned} C &= AB = A^H B^H x^m + \\ &+ (A^H B^H + A^L B^L + (A^H + A^L)(B^H + B^L)) x^{\frac{m}{2}} + \\ &+ A^L B^L \end{aligned} \quad (11)$$

Three multiplications each one of $2^{\frac{m}{2}}$ bits are therefore used to compute C.

Let us consider now the case (relevant for elliptic curve cryptography) where m is a prime, that can be expressed as $m = 2^k + d$, where d represents some *leftover* bits. One could be tempted to use direct Karatsuba algorithm by promoting m to 2^{k+1}. However this approach will clearly causes a wastage of extra arithmetic operations as all $2^k - d$ most significant bits are zeroes. Binary Karatsuba algorithm strategy suggests not to promote $2^k + d$ to 2^{k+1}, but instead perform multiplications separately for 2^k and d where only d is promoted to $2^{k'}$ where k' is an integer [19].

As a design example, consider the binary Karatsuba multiplier shown in Fig. 10. That circuit computes the polynomial multiplication of the elements A and $B \in GF(2^{191})$. Notice that for this case $m = 191 = 2^k + d = 2^7 + 63$. We can do much better by assuming that the d = 63 most significant leftover bits are 64. Notice that all the multiplier blocks of Fig. 10 are computed in parallel. Therefore the total time delay of the circuit is given as, $T_{\text{DELAY}}\text{KM}2^{128} + T_{\Delta}$ where

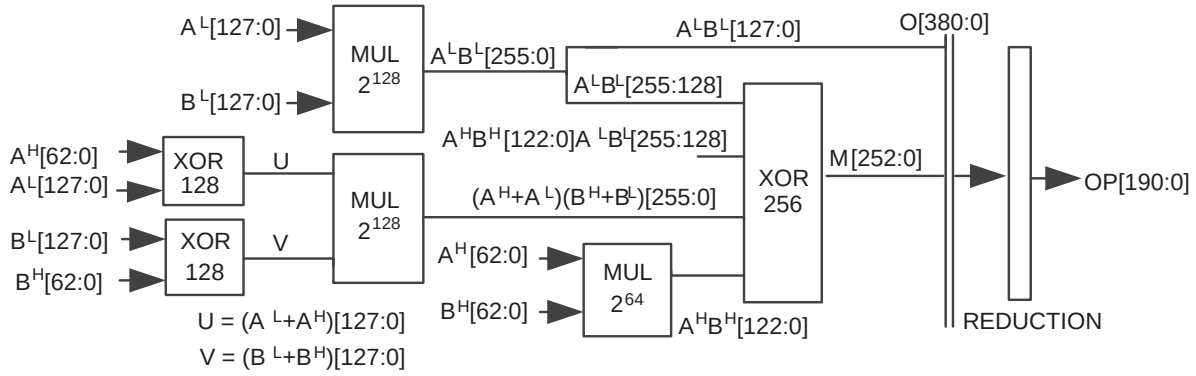

Fig. 10. Karatsuba Multiplier

$T_{\text{DELAY}}\text{KM}2^{128}$ is the time delay associated to the Karatsuba multiplier for $GF(2^{128})$ and T_Δ is the overhead associated to the other circuitry.

Once the polynomial multiplication/squaring over $GF(2^m)$ is completed, reduction must be performed as is explained in the remaining part of this subsection

C. *Reduction*

Let $A(x), B(x) \in GF(2^m)$ with irreducible polynomial $P(x)$ and we assume that the computation of polynomial product $C(x)$ has already been made by using any of the two methods described in previous two subsections, then the modular product C' can be achieved by XOR operations only. Recall that the polynomial product C and the modular product C', have $2m-1$ and m, coordinates, respectively, i.e.,

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

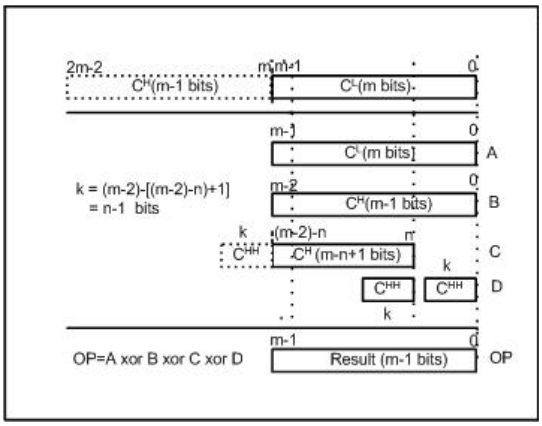

Fig. 11. Reduction Diagram

Fig. 11 shows how to implement on reconfigurable hardware the reduction strategy for a general generating polynomial $P(x) = x^m + x^n + 1$. As it was mentioned before, the reduction step involves XOR and overlap operations. Notice that if we assume that $P(x)$ is a trinomial, then reduction becomes very economical as XOR operation fits well in 4-input/1-output typical structures of FPGA devices.

Although the strategy shown in Fig. 11 is completely general, for the purposes of this research work we utilized a fixed irreducible generating trinomial, namely, $P(x) = x^{191} + x^9 + 1$.

D. *Inversion*

The Euclidean algorithm required $O(n^2)$ bit operations to perform inversion. FLT establishes that for any nonzero element $\alpha \in GF(2^m)$, the identity $\alpha^{-1} \equiv \alpha^{2^m-2}$ holds. Therefore, multiplicative inversion can be performed by computing,

$$\alpha^{2^m-2} = \alpha^{2^1} \times \alpha^{2^1} \times \cdots \alpha^{2^{m-1}} \quad (13)$$

A straightforward implementation of Eq. 13 can be carried out using the binary exponentiation method, which requires $m-1$ field squarings and $m-2$ field multiplications. The Itoh-Tsujii Multiplicative Inverse Algorithm (ITMIA) algorithm on the other hand, reduces the required number of multiplications to $k + hw(m-1) - 2$, where $k = \lfloor log_2(m-1) \rfloor$ and $hw(m-1)$ are the number of bits and the Hamming weight of the binary representation of $m-1$, respectively. This remarkably saving on the number of multiplications is based on the observation that since $2^m - 2 = (2^{m-1} - 1) \cdot 2$, then the identity from Fermat's little theorem can be rewritten as $\alpha^{-1} \equiv \alpha^{2^m-2} \equiv \alpha^{(2^{m-1}-1)^2}$. Then ITMIA computes the field element $(2^{m-1} - 1)$ using a recursive re-arrangement of the finite field operations.

Using the ITMIA algorithm one can find an addition chain for an arbitrary m. We used this algorithm to obtain an addition chain for m = 191 (our design example). In this case $e = m-1$ = 190 = $2^7 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1$. Therefore,

$$\begin{aligned} U = 1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 128 \rightarrow \\ 160 \rightarrow 176 \rightarrow 184 \rightarrow 188 \rightarrow 190 \end{aligned} \quad (14)$$

The addition chain in Eq. 14 can be accomplished using 12 multiplications and $m-1$ squarings in accordance to the algorithm. However it can be seen that various addition chains of shortest length for the same m exist. Let l be the shortest length of any valid addition chain for a given positive integer $e = m-1$. Then the problem of finding an addition chain for e with length l is an **NP**-hard problem [20]. Let us first formally define an addition chain in order to describe a methodology to achieve field multiplicative inverses.

E. Addition Chains

An *addition chain* U for a positive integer $e = m - 1$ of length t is a sequence of positive integers $U = \{u_0, u_1, \cdots, u_t\}$, and an associated sequence of r pairs $V = \{(v_1, v_2 \cdots, v_t\}$ with $v_i = (i_1, i_2), 0 \leq i_2 \leq i_1 < i$, such that:

- $u_0 = 1$ and $u_t = m - 1$;
- for each $u_i, 1 \leq i \leq t, u_i = u_{i_1} + u_{i_2}$.

Consider the same design example for m = 191, $e = m - 1 = 190 - 1$. Then, one of possible addition chain with length $t = 10$ is,

$$U = 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 10 \rightarrow 20 \rightarrow 40 \rightarrow 80 \rightarrow 160 \rightarrow 180 \rightarrow 190 \quad (15)$$

Each term of this chain u_i is obtained either by doubling the immediate preceding term $u_{i-1} + u_{i-1}$ or by adding any two previous terms $u_j + u_k$. That provides us two indices i_1 and i_2. In the former case, the same term was doubled so $i_1 = i_2 = i - 1$ while for later case two different terms u_j and u_k were added provided $i_1 = j$ and $i_2 = k$.

Input: An element $\alpha \in GF(2^m)$, an addition chain U of length t for $m - 1$ and its associated sequence V.
Output: $\alpha^{-1} \in GF(2^m)$
Procedure MultiplicativeInversion(α, U)
1. $\beta_0 = \alpha$
2. for i from 1 to t do:
3. $\beta_i = (\beta_{i_1})^{2^{u_{i_2}}} \cdot \beta_{i_2}$
4. return ($\beta_t{}^2$);

Fig. 12. An Algorithm for multiplicative inversion using addition chains

Consider the algorithm shown in Fig. 12. That algorithm iteratively computes the β_i coefficients in the exact order stipulated by the addition chain U. Indeed, starting from $\beta_0 = (\alpha)^{2^{u_0}-1} = (\alpha)^{2^{u_0}-1} = \alpha^{2^1-1}$, the algorithm computes the other t β_i coefficients. In the final iteration, after having computed the coefficient $\beta_t = (\alpha)^{2^{m-1}-1}$, the algorithm returns the required multiplicative inversion by performing a regular field squaring, namely, $\beta_t^2 = (\alpha^{2^m-2}) = \alpha^{-1}$.

Let us assume that the binary extension field $GF(2^{191})$ has been generated with the trinomial $P(x) = x^{191} + x^9 + 1$, irreducible over $GF(2)$. Let $\alpha \in GF(2^{191})$ be an arbitrary nonzero field element. Then, using the addition chain of Example 15, the algorithm of Fig. 12 will compute the sequence of β coefficients as shown in Table IV-E. Once again, notice that after having computed the coefficient β_{10} the only remaining step is to obtain α^{-1} as, $\alpha^{-1} = \beta_{10}^2$

Let us now assess the computational complexity of the algorithm shown in Fig. 12. The algorithm performs t iterations (where t is the length of the addition chain U) and one field multiplication per iteration. Thus, we conclude that a total of t field multiplication computations are required.

On the other hand, notice that at each iteration i, a total of $2^{u_{i_2}}$ field squarings are performed. Notice also that by definition, the addition chain guarantees that for each $u_i, 1 \leq i \leq t$, the relation $u_{i_2} = u_i - u_{i_1}$ holds. Hence, one can show by induction that the total number of field squaring operations performed right after the execution of the i-esime iteration is $u_i - 1$ [21]. Therefore, at the end of the final iteration t, a total of $u_t - 1 = m - 2$ squaring operations have been performed. This, together with the final squaring operation, yield a total of $m - 1$ field squaring computations.

Summarizing, the algorithm of Fig. 12 can find the inverse of any nonzero element of the field using exactly,

$$\begin{aligned} \#Multiplications &= t; \\ \#Squarings &= m - 1. \end{aligned} \quad (16)$$

V. Design Implementation

Although the basic structure of FPGAs is the same, FPGA manufacturers provide diverse features for their devices, Those features include 1, 2 or 4 slices within a CLB, built-in additional logic or memory modules. It will be therefore useful to understand the architectural description of the target device. In the rest of this section, the architectural description for the target device (XCV3200) is provided and the implementation summary that describes device ultilisation by our design.

A. Architectural Description of the Target Device

FPGAs comprise two major configurable elements: configurable logic blocks (CLBs) and input/output blocks (IOBs). CLBs provide the functional elements for constructing Logic and IOBs provide the interface between the package pins and the CLBs. FPGAs come from different vendors but the basic CLB structure remains the same. High density FPGAs contains large number of CLBs and IOBs. Some families of FPGA devices offer additional logic such as built-in multipliers and memory modules and even some of them include power PCs. We used Virtex-E device XCV3200 for the implementation of elliptic curve scalar multiplication. A brief detail of the internal structure is presented here for the understanding of the equation for device utilization.

The basic building block of the Virtex-E CLB is the logic cell (LC). An LC includes a 4-input function generator, carry logic, and a storage element (flip-flop). The output from the function generator in each LC drives both the CLB output and the D input of the flip-flop. Each Virtex-E CLB contains four LCs, organized in two similar slices. In addition to the four basic LCs, the Virtex-E CLB contains logic that combines function generators to provide functions of five or six inputs. Consequently, when estimating the number of system gates provided by a given device, each CLB counts as 4.5 LCs. Virtex-E function generators are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT can provide a 16 x 1-bit synchronous RAM. Furthermore, the two LUTs within a slice can be combined to create a 16 x 2-bit or 32 x 1-bit synchronous RAM, or a 16 x 1-bit dual-port synchronous RAM. The Virtex-E architecture also includes dedicated block memories of 4096 bits each. The Virtex and Virtex-E family of devices contain more than 280 block SelectRAM memories. Those are fast access memories and do not occupy CLBs. This is why

TABLE IV
ALGORITHM OF FIG. 12: β_i COEFFICIENT GENERATION

i	u_i	i_1	i_2	u_{i2}	rule	$\beta_{i_1}^{2^{u_{i_2}}} \cdot \beta_{i_2}$	$\beta_i = \alpha^{2^u_i - 1}$
0	1	0	0	1	–	–	$\beta_0 = \alpha^{2^1-1}$
1	2	0	0	1	$u_0 + u_0$	$\beta_0^{2^1} \cdot \beta_0$	$\beta_1 = \alpha^{2^2-1}$
2	3	1	0	1	$u_1 + u_0$	$\beta_1^{2^1} \cdot \beta_0$	$\beta_2 = \alpha^{2^3-1}$
3	5	2	1	2	$u_2 + u_1$	$\beta_2^{2^2} \cdot \beta_1$	$\beta_3 = \alpha^{2^5-1}$
4	10	3	3	5	$u_3 + u_3$	$\beta_3^{2^5} \cdot \beta_3$	$\beta_4 = \alpha^{2^{10}-1}$
5	20	4	4	10	$u_4 + u_4$	$\beta_4^{2^{10}} \cdot \beta_4$	$\beta_5 = \alpha^{2^{20}-1}$
6	40	5	5	20	$u_5 + u_5$	$\beta_5^{2^{20}} \cdot \beta_5$	$\beta_6 = \alpha^{2^{40}-1}$
7	80	6	6	40	$u_6 + u_6$	$\beta_6^{2^{40}} \cdot \beta_6$	$\beta_7 = \alpha^{2^{80}-1}$
8	160	7	7	80	$u_7 + u_7$	$\beta_7^{2^{80}} \cdot \beta_7$	$\beta_8 = \alpha^{2^{160}-1}$
9	180	8	5	20	$u_8 + u_5$	$\beta_8^{2^{20}} \cdot \beta_5$	$\beta_9 = \alpha^{2^{180}-1}$
10	190	9	4	10	$u_9 + u_4$	$\beta_9^{2^{10}} \cdot \beta_4$	$\beta_{10} = \alpha^{2^{190}-1}$

they are mentioned separately when used by the designers. Each Virtex-E CLB contains two 3-state drivers (BUFTs) that can drive on-chip busses. Each Virtex-E BUFT has an independent 3-state control pin and an independent input pin. Using BUFTs in the design does not include in the CLBs count and are described separately. The implementation summary of an FPGA design therefore includes number of CLB slices, number of slice flip-flops, number of LUTs, number of TBUFs, number of bonded IOBs, number of BRAMs and number of global clocks (GCLKIOBs). It is to be noted that the design tool will only provide the detail about a specific unit in the design summary if it is used in the design.

B. FPGA's implementation of Elliptic Curve Scalar Multiplication

For our FPGA implementation two 191-bit field multipliers were used in combination with eight multiplexers and six field squarer blocks by using VirtexE XCV3200 Xilinx device. The Virtex and VirtexE family of devices integerate more than 280 fast access memory modules BlockRams (BRAMs). A dual port BRAM can be configured into two independent single port BRAMs. This special feature was exploited in our design in order to store and retrieve data from different memory locations irrespective of the input ports. Two similar memory blocks each of 12 BRAMs were used. Just two BRAMs are sufficient here however the combination of 12 BRAMs actually allows storing and retrieving word lengths of 191 bits. Extra benefits of using BRAMs include reduction of design complexity by saving a lot of register and multiplexer operations.

Two multiplications are computed simultaneously by using two field multipliers as shown in Fig. 8. Six multiplications (four for point addition and two for point doubling) are therefore performed in just three clock cycles. It takes two clock cycles for reading/writing to BRAMs, a rapid clock is applied to them which is further divided by two to serve as master clock for the rest of the design.

Control unit (CU) synchronizes data flow described in Table II and Table III by generating select signals (s_i) to the multiplexers and addresses to BRAMs. CU test each bit of the vector k from left to right (in descending order) and generates control signals depending the bit is 1 or 0. The bits are examined after each 3 master clock cycles by consuming $3 \times 191 = 573$ clock cylces to complete EC scalar multiplication.

A squarer block 'SqrInv' in Fig. 8 is used to perform squaring operations for inversion. Squaring in $GF(2^m)$ is a simple operation however, as it was stated in Eq. 16, it devours $m-1$ clock cycles performing $m-1$ squarings for m-bit inversion. Fortunately, the number of field multiplications needed for computing the multiplicative inversion is quite moderated, which is valuable for the design since field multiplication in $GF(2^m)$ is a costly and extensive time consuming operation.

Figure 13.a shows our strategy for implementing field squaring trying to use as few clock cycles as possible. Polynomial squaring is free of cost and is obtained by connecting each alternate even bit to zero (ground). That operation is followed by a second step where reduction is performed by using few XOR gates. Figure 13.b shows the $GF(2^{191})$ field squarer implementation used in this work. Referring to the addition chain described in Table IV-E, frequent squaring operations in $GF(2^{191})$ are ${\beta_i}^{2^1}$ (2 times), $\beta_i 2^2$ (1 time), $\beta_i 2^5$ (1 time), and $\beta_i 2^{10}$ (10 times). That is why we preferred to cascade 10 field squarer blocks back to back and then by the appropriate use of multiplexers obtain the corresponding outputs after 1, 2, 5, and 10 squarer blocks as shown in Figure 13.b. As an example, the $x^{2^{20}}$ field operation can be performed in just two clock cycles by taking the output after the last squarer block (10 squarers) in the first clock cycle and then after a second clock cycle we will get the required 20 field squarings.

C. Implementation summary

All finite field arithmetic and then kP computational architectures were implemented on VirtexE XCV3200 by using Xilinx Foundation Tool F4.1i for design entry, synthesis, testing, implementation and verification of results. Table V represents timing performances and occupied resources by the said architectures.

Elliptic curve point addition and point doubling do not participate directly as a single computational unit in this design

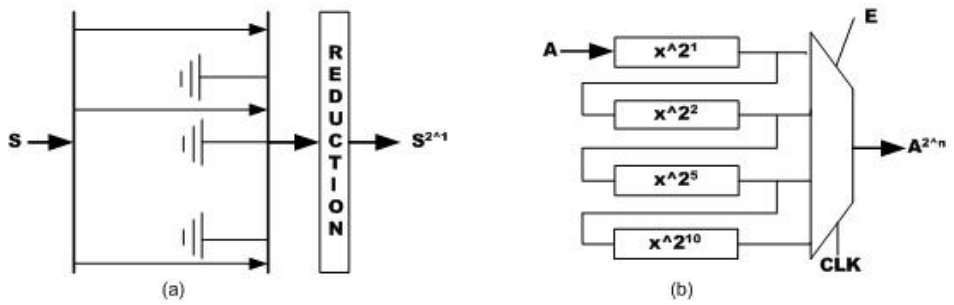

Fig. 13. Squarer $GF(2^{191})$ (a) for x^{2^1} (b) for x^{2^n}

TABLE VI
$GF(2^m)$ ELLIPTIC CURVE POINT MULTIPLICATION HARDWARE PERFORMANCE COMPARISON

Reference	Field	Platform	Max. Freq.	$kP(\mu s)$	Speedup	Multiplier Block Utilized
[22]	$GF(2^{113})$ $GF(2^{155})$ $GF(2^{173})$	Annapolis Micro Systems Wildstar board		4300 8300 11100	76 148 198	Serial multiplier
[5]	$GF(2^{160})$	0.13μ CMOS ASIC	510.2MHz	190	3.39	64-bit multiplier
[23]	$GF(2^{176})$	0.25μ CMOS	50 MHz	6950	124	1024-bit adder
[24]	$GF(2^{167})$	XCV400E	76.7 MHz	210	3.75	16-bit multiplier 167-bit squarer
[8]	$GF(2^{191})$	XCV4000XL		11820 17710	211 316	Not specified
[9]	$GF(2^{163})$ $GF(2^{193})$	XCV2000E		143 187	2.55 3.34	64-bit shift and add multiplier
[10]	$GF(2^{113})$	AT94K40	12MHz	1400	25	24-bit Karatsuba multiplier
[11]	$GF(2^{191})$	XCV1000BG	50MHz 36MHz	270 2270	4.82 40	4 2-Bit level LFSR multiplier 1 Massey-Omura multiplier
This work	$GF(2^{191})$	XCV3200E	9.99MHz	59.26		Two $GF(2^{191})$ Binary karatsuba multiplier

TABLE V
SUMMARY OF IMPLEMENTED DESIGNS

Design	Device (XCV)	CLB slices	Timings
Binary Karatsuba Multiplier $GF(2^{191})$	3200E	8721	43.1ηs
Point addition + Point doubling $GF(2^{191})$(Fig 8)	3200E	18300	300.3ηs
Inversion $GF(2^{191})$	3200E	1312	1.9μs
Point Multiplication $GF(2^{191})$	3200E 26 BRAMs	19626	59.26μs

but parallel computations for both point addition and point doubling are designed together as it was shown in Fig. 8. Both point addition and point doubling occupy 18300 (56 %) CLB slices and it takes $100.1\eta s$ (9.99 MHz) for one computational cycle. As it was mentioned earlier those three cycles are used for computing both point addition and point doubling (six multiplications), $300.3\eta s$ is the total consumed time. Inversion is performed at the end. It takes 19 clock cycles to perform one inversion in $GF(2^{191})$ occupying 1312 CLB slices. The CLB slices for inversion in fact are the FPGA resources occupied for squaring operations only and the multiplier blocks are the same used for point addition and point doubling. Finally, point multiplication is performed in $59.26\mu s$ which is m (m=191 for our case) times the computational time for point addition and point doubling. It costs 19626 (60 %) CLB slices and 24 (11%) BRAMs. The total time is the sum of the time for the scalar multiplication and the time to perform inversion for coordinate conversion i-e 57.36 + 1.9 = 59.26 μs. Our Binary Karatsuba Multiplier in $GF(2^{191})$ occupies 8721 (26.87%) CLB slices and one field multiplication is performed in 43.1ηs.

VI. COMPARISON

Table VI provides a quick comparison of the existing FPGA's implementations of elliptic curve scalar multiplication over $GF(2^m)$. That table sums up the last three years state of the art implementations, where most of the works featured have been published this same year. A microcoded EC processor in [22] is implemented on Annapolis Microsystems Wildstar board. For this design, EC multiplication is executed in $4300\mu s$, $8300\mu s$, and $11100\mu s$ for $GF(2^{113})$, $GF(2^{155})$, and $GF(2^{173})$ respectively. An efficient VLSI EC processor in [5] support EC scalar multiplication both in $GF(p)$ and $GF(2^n)$. Achieved results for a 160-bit EC scalar multiplication are $1210\mu s$ and $190\mu s$ for $GF(p)$ and $GF(2^n)$ respectively. A generic VLSI architecture in [23] implements cryptographic primitives over various fields. It consumes $6950\mu s$ to compute

point multiplication using a repeated double-and-add algorithm. Another reconfigurable system on chip ECC implementation is reported on a special architecture AT94K40 from Atmel that integrates various components including an AVR 8-bit RISC micro-controller, several peripheral devices and up to 36K bytes SRAM within a single chip. That design execute EC operation in just $1400\mu s$. All other designs [24], [4], [8], [11] implements EC scalar multiplication on single chip FPGA.

Table VI also includes a speedup factor that measures how much our design is faster than the others. It is obtained by dividing kP computational time of a given design over the time taken by our design. As it can be seen, our design shows an improvement ranging from 2.55 times up to 316 times of speedup.

The design presented in [5] can handle arbitrary fields and elliptic curves without changing its hardware configuration, while those parameters have been fixed in our implementation. However the design in [5] 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.

VII. Conclusions

In this work, a parallel generic design strategy for elliptic curve point multiplication was presented. The architecture was then optimized for the finite field $GF(2^{191})$, using a Xilinx VirtexE XCV3200 FPGA device, occupying 18314 CLB slices, with a performance time of just $59.26\mu s$.

Our design included the implementation of the complete set of field arithmetic operators (including field multiplication and inversion) as well as the main building blocks needed to perform the Montgomery scalar multiplication algorithm, namely point doubling and point addition. It is important to point out that those two blocks were implemented using a parallel architecture that allowed us to cut the computing time by a factor of two.

That high performance was achieved not only because of the usage of a relatively large FPGA target device, but also because of two major design improvements. First, the instrumentation of parallel structural arrangements for the computation of point addition and point doubling as shown in Fig. 8 yield us a reduction in the required number of field operations, thus reducing the total computational time. The second major reason of the high performance shown by our design is due to the usage of efficient field multipliers whose critical paths were carefully optimized to give maximum performance.

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 make no assumption about the specific field targeted and hence can be easily adapted to accommodate other designs with different field sizes.

As a whole, the architecture presents a promising approach for elliptic curve scalar multiplication providing a balance between hardware area and time.

References

[1] D. V. Chudnovsky and G. V. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factorization tests," *Advances in Applied Math.*, vol. 7, pp. 385–434, 1986.

[2] J. Lopez and 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*, vol. 1717, pp. 316–327, August 1999.

[3] D. Hankerson, J. Lopez-Hernandez, and 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*, vol. 1965, pp. 1–24, August 2000.

[4] N. Smart and E. Westwood, "Point multiplication on ordinary elliptic curves over fields of characteristic three," *Applicable Algebra in Engineering, Communication and Computing*, vol. 13, pp. 485–497, 2003.

[5] A. Satoh and K. Takano, "A Scalable Dual-Field Elliptic Curve Cryptographic Processor," *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 449–460, April 2003.

[6] R. Schroeppel, C. Beaver, R. Gonzales, R. Miller, and 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*, vol. 2523, pp. 366–380, August 2003.

[7] G. Orlando and C. Paar, "A Scalable $GF(p)$ Elliptic Curve Processor Architecture for Programmable Hardware," *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, vol. 2162, pp. 348–363, May 2001.

[8] 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*, vol. 2162, pp. 118–125, May 2001.

[9] N. Gura, S. Shantz, and 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*, vol. 2523, pp. 349–365, August 2003.

[10] M. Ernst, M. Jung, and F. M. 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*, vol. 2523, pp. 381–399, August 2003.

[11] M. Bednara, M. Daldrup, J. Shokrollahi, J. Teich, and 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., April 2002.

[12] N. A. Saqib, F. Rodríguez-Henriquez, and A. Díaz-Pérez, "A Parallel Architecture for Fast Computation of Elliptic Curve Scalar Multiplication over GF(2^m)," in *Proc. of The 11th Reconfigurable Architectures Workshop (RAW-04*. Santa Fé, New Mexico): IEEE Computer Society, April 2004, p. 144.

[13] F. Rodríguez-Henriquez, N. A. Saqib, and A. Díaz-Pérez, "A fast parallel implementation of elliptic curve point multiplication over gf2^n)," *Microprocessor and Microsystems*, vol. 28, no. 5-6, pp. 329–339, Auguest 2004, special Issue on FPGAs: Applications and Designs.

[14] M. A. Hasan, M. Z. Wang, and V. K. Bhargava, "A modified Massey-Omura parallel multiplier for a class of finite fields," *IEEE Transactions on Computers*, vol. 42(10), pp. 1278–1280, November 1993.

[15] B. Sunar and Ç. K. Koç, "Mastrovito multiplier for all trinomials," *IEEE Transactions on Computers*, vol. 48(5), pp. 522–527, May 1999.

[16] M. Morii, M. Kasahara, and D. L. Whiting, "Efficient bit-serial multiplication and the discrete-time Wiener-Hopf equation over finite fields," *IEEE Transactions on Information Theory*, vol. 35, no. 6, pp. 1177–1183, 1989.

[17] H. Wu and M. A. Hasan, "Low complexity bit-parallel multipliers for a class of finite fields," *IEEE Transactions on Computers*, vol. 47, no. 8, pp. 883–887, August 1998.

[18] E. Bach and J. Shallit, *Algorithmic number theory, Volume I: efficient algorithms*. Boston, MA: Kluwer Academic Publishers, 1996.

[19] F. Rodríguez-Henríquez and Ç. K. Koç, "On fully parallel Karatsuba Multipliers for $GF(2^m)$," in *International Conference on Computer Science and Technology (CST 2003)*, Cancun, Mexico, May 2003.
[20] A. J. Menezes, P. C. van Oorschot, and S. A.Vanstone, *Handbook of Applied Cryptography*. Boca Raton, Florida: CRC Press, 1996.
[21] J. von zur Gathen and M. Nöcker, "Computing special powers in finite fields: extended abstract," in *Proceedings of the 1999 international symposium on Symbolic and algebraic computation*. ACM Press, 1999, pp. 83–90.
[22] P. H. W. Leong and I. K. H. Leung, "A Microcoded Elliptic Curve Processor Using FPGA Technology," *IEEE Transactions on VLSI Systems*, vol. 10, no. 5, pp. 550–559, October 2002.
[23] J. Goodman and A. Chandrakasan, "An Energy-Efficient Reconfigurability Public-Key Cryptography Processor," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1808–1820, November 2001.
[24] G. Orlando and 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*, vol. 1965, pp. 41–56, August 2000.