Elliptic Curve Scalar Multiplication using Point Halving on Reconfigurable Hardware Platforms

Sabel Mercurio Hernández Rodríguez, Francisco Rodríguez-Henríquez

***Abstract*— In this paper, a FPGA arithmetic logic unit architecture for computing elliptic curve scalar multiplication over the binary extension field $GF(2^{163})$ is presented. Proposed architecture implements a parallel version of the mixed-coordinate point addition and point doubling formulae. This way, our design can perform elliptic curve point addition, point doubling and point halving efficiently in terms of area resources and timing performance. In fact, our experimental results show that our proposed design can perform an elliptic curve scalar multiplication in about $25\mu S$.**

***Index Terms*— elliptic curve, scalar multiplication, point halving, reconfigurable hardware.**

I. Introduction

A digital signature is a method for authenticating digital information which is analogous to a hand writing signatures cryptography. A digital signature method generally defines two complementary algorithms, one for signing and the other for verification, and the output of the signing process is also called a digital signature.

There are three common reasons for applying a digital signature to communications:

- *Authenticity*, in others words allows the recipient of a message to be confident that the sender is indeed who s/he claims to be. This is especialy important in a financial context.
- *Integrity*, this is that both parties (recipent, sender) will always wish to be confident that a message has not been altered during transmission. The encryption makes it difficult for a third party to read a message, but that third party may still be able to alter it in a useful way.
- *Non-repudiation*, because in a cryptographic context, the word repudiation refers to the act of denying association with a message (ie claiming it was sent by a third party). The recipient of a message may insist that the sender attach a signature in order to prevent any later repudiation, since the recipient may show the message to a third party to prove its origin.

Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analogue of the Digital Signature Algorithm (DSA). Unlike the discrete logarithm problem and the integer factorization problem, no sub exponential-time algorithm is known for the elliptic curve discrete logarithm problem. For this reason, the main benefit of ECC is that under certain situations it uses smaller keys than other methods (such as RSA) while providing an equivalent or higher level of security.

Although elliptic curve cryptosystems can be defined over prime fields, for hardware and reconfigurable hardware platform implementations, binary extension finite fields are preferred. This is largely due to the carry-free binary nature exhibit by this type of fields, which is a valuable characteristic for hardware systems leading to both, higher performance and lesser area consumption.

Since ECC proposal in 1985 [1], [2], a vast amount of work and investigation has been done reporting efficient and/or versatile elliptic curve scalar multiplication over divers platforms as can be: software implementations [3], [4], [5]; VLSI implementations [6] and reconfigurable hardware implementations [7], [8], [9] have been targeted.

In 1999 Schroeppel [10] and Knudsen [11] independently proposed a method to speedup scalar multiplication on elliptic curves defined over binary extension fields. Their method is based on a novel elliptic curve primitive called *point halving*, which can be defined as follows. Given a point Q of odd order, compute P such that $Q = 2P$. The point P is denoted as $\frac{1}{2}Q$. Since theoretically, point halving is up to three times as fast as point doubling, it is possible to improve the performance of scalar multiplication computation $Q = nP$ by replacing the double-and-add algorithm with a halve-and-add method based on an expansion of the scalar n in terms of negative powers of 2.

Cryptographic systems based on elliptic curves depend on arithmetic involving the points of the curve. For this reason is necesary implement efficient curve operations in order to obtain high performances. However curve operations relays on finite field arithmetic. In this contribution we investigate how to efficiently implement the later using parallel strategies. We present an architecture that employs a parallelized version of the half-and-add method and its associated building blocks. We describe all the implementation details of an architecture able to compute $GF(2^{163})$ elliptic curve scalar multiplication in approximately $25\mu s$ plus one field inversion.

II. Mathematical background

Let $P(x)$ be a degree-m pentanomial of the form $P(x) = x^m + x^{k1} + x^{k2} + x^{k3} + 1$, irreducible over $GF(2)$. Then $P(x)$ generates the finite field $F_q = GF(2^m)$ of characteristic two. Field elements can be represented in canonical basis as binary polynomials of degree at most $m-1$ with coefficients 0 or 1. A non-super singular 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 defined below.

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve 1. We define the addition inverse of P as $-P = (x_1, x_1 + y_1)$. Furthermore, if $Q \neq -P$, then the point $P + Q = (x_3, y_3)$, can be computed as,

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

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

Using above group law, one can define elliptic curve scalar multiplication as follows. let E be an elliptic curve over a binary finite field $GF(2^m)$, let $Q, P \in E$ be two arbitrary elliptic points satisfying the curve equation, and let us define k as an arbitrary positive integer. Then we can define elliptic curve scalar multiplication $Q = kP$, as, $kP = Q = \underbrace{P + \cdots + P}_{\text{k times}}$

III. Elliptic Curve Cryptography

In order to design an architecture that performs elliptic curve scalar multiplication it is necessary to design first an architecture for computing finite field arithmetic efficiently. By accomplishing this arithmetic, we can design efficient elliptic curve arithmetic. And finally we can implement the final circuit that performs elliptic curve scalar multiplication.

A. Efficient Finite Field Arithmetic

We need to consider six basic arithmetic operations, i.e., Field multiplication, half-trace computation, square-root extraction, field squaring, trace and addition.

In the rest of this Section we describe how we implemented all six arithmetic operators mentioned above.

1) Addition: Addition of field elements is performed bitwise using only XOR gates. This is by far the easiest arithmetic operation.

2) Field Multiplication: In order to compute a field multiplication over $GF(2^{163})$ we use a Karatsuba-Ofman multiplier.

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 $GF(2^m)$, represented in polynomial basis as $A(x) = \sum_{i=0}^{m-1} a_i x^i$ and $B(x) = \sum_{i=0}^{m-1} b_i x^i$, 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 $C(x) = A(x)B(x)$ and $C'(x) = C(x) \bmod P(x)$. This operation can be achieve by performing two steps: polynomial multiplication (depicted in fig. 1) followed by modular reduction (shown in fig. 2).

In this work we use a variation of the classic Karatsuba-Ofman multiplier called binary Karatsuba multiplier that was first propose in [12].

Let be $A, B \in GF(2^m)$, so A, b can be expressed as $A = x^{m/2}A_H + A_L$ and $B = x^(m/2)B_H + B_L$, then we can express C as:

$$\begin{aligned} C &= AB \\ &= A_H B_H x^m + A_L B_H x^{m/2} + A_H B_L x^{m/2} + A_L B_L \\ &= A_H B_H x^m + (A_H B_H + A_L B_L + (A_H + A_L)(B_H + B_L))x^{m/2} + A_L B_L \end{aligned} \quad (4)$$

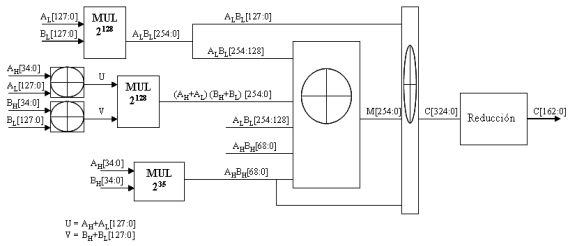

Fig. 1. Binary Karatsuba-Ofman multiplier.

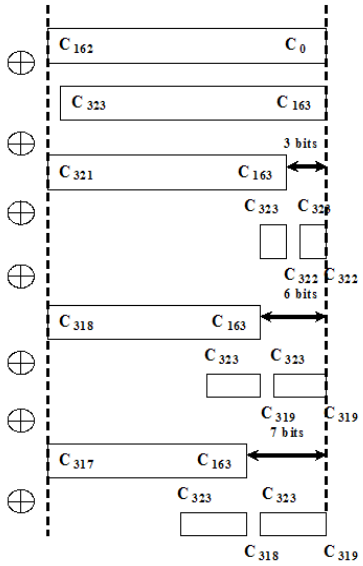

Fig. 2. Pentanomial reduction.

3) Squaring: Once again, let A be an arbitrary element of the field $GF(2^m)$, represented in the canonical basis as an $m-1$ degree polynomial, namely, $A(x) = \sum_{i=0}^{m-1} a_i x^i$. Then the element $C \in GF(2^m)$, $C = A^2 \bmod P(x)$, can be obtained by computing first the polynomial product of A by itself, followed by a reduction step modulo $P(x)$. In fact, for the first step, the polynomial square of A can be written as,

$$\begin{aligned} A^2(x) &= \left(\sum_{i=0}^{m-1} a_i x^i\right) \cdot \left(\sum_{i=0}^{m-1} a_i x^i\right) \\ &= \sum_{i=0}^{m-1} a_i x^{2i} \end{aligned} \quad (5)$$

Thereafter, as we did for multiplication, we must apply the reduction scheme of Figure 2.

4) Square root Extraction: A straightforward but rather expensive approach for computing $\sqrt{A}$ is based on Fermat's little theorem which establishes that for any nonzero element $A \in F$, the identity $A^{2^m} = A$ holds. Therefore, $\sqrt{A}$ may be computed as $D = \sqrt{A} = A^{2^{m-1}}$, with a computational cost of $m-1$ field squarings [13]. A more efficient algorithm based on a refining of the Fermat's little theorem method just outlined was proposed in [5]. That method is based on the observation that $\sqrt{A}$ can be expressed in terms of the square root of the variable x. In the case that the generating polynomial is an irreducible trinomial, authors in [5] showed that square root can be computed with some shift-left operations and one modular reduction. In this contribution we describe an alternative method for computing $\sqrt{A}$, which is also based on the linearity property exhibited by the field squaring operation defined over binary extension fields.

Since field squaring over binary extension fields is a linear operation, we can rephrase the field squaring computation in terms of an $m \times m$ transformation matrix as, $C = MA = A^2$. Where M is a spare square matrix if the field generating polynomial is trinomial or pentanomial. On the other hand, let us recall that extracting the square root of an arbitrary field element A means finding a field element D such that $D^2 = A$ holds. Hence, if we define D as the field element $D = M^{-1}A$. Then, it follows that D as defined above is indeed the square root of A since, $D^2 = MD = M(M^{-1}A) = A$. Thus concluding that $D = \sqrt{A}$.

5) Trace function: Given $C \in GF(2^m)$, the trace function can be defined as:

$$Tr(C) = C + C^2 + C^{2^2} + \ldots + C^{2^{m-1}} \tag{6}$$

Due to its linearity, the trace function can be implemented such that the execution time is $O(1)$:

$$Tr(C) = Tr(\sum_{i=0}^{m-1} c_i x^i) = \sum_{i=0}^{m-1} c_i Tr(x^i) \tag{7}$$

As an example the field defined by $GF(2^{163})$ with the reduction polynomial $p(x) = x^{163} + x^7 + x^6 + x^3 + 1$, $Tr(x^i) = 1$ if and only if $i \in \{0, 157\}$. The implementation of the trace function in reconfigurable hardware only needs one XOR gate to add the bits *0 y 157* from the input polynomial.

6) Solving a quadratic equation over $GF(2^m)$: In order to solve a quadratic Equation (12), we may use the half-trace function. Let $C \in GF(2^m)$ be defined as $C(x) = \sum_{i=0}^{m-1} c_i x^i \in GF_{2^m}$ with $Tr(C) = 0$ and given m an odd integer, the half-trace function can be defined as:

$$H(C) = H(\sum_{i=0}^{m-1} c_i x^i) = \sum_{i=0}^{m-1} c_i H(x^i) \tag{8}$$

Therefore, by using the definition of the half trace equation 8. We can precompute the m half-traces of the field elements x^i for $i = 0, 1, \ldots, m-1$; and by arranging these Equations in a $m \times m$ matrix B, we may obtain the half-trace of an arbitrary element $C \in GF(2^m)$ by computing $H(C) = CB$.

B. Efficient Elliptic Curve Arithmetic

Using the operations defined in Section III-A, we can now define three main elliptic curve operations: point addition, point doubling and point halving.

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.

In *López-Dahab (LD)* projective coordinates [14] the projective point (X:Y:Z) with Z$\neq$ 0 corresponds to the affine coordinates $x = X/Z$ and $y = Y/Z^2$. The elliptic curve Equation (1) mapped to LD projective coordinates is given as:

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4 \tag{9}$$

1) Point Doubling: The formulae for computing point addition are given below [14]. Point doubling can be performed at a computational cost of 4 field multiplications as,

$$\begin{array}{rcl} Z_3 & = & {X_1}^2 \cdot {Z_1}^2; X_3 = {X_1}^4 + b \cdot {Z_1}^4; \\ Y_3 & = & b{Z_1}^4 Z_3 + X_3 \cdot (aZ_3 + {Y_1}^2 + b{Z_1}^4) \end{array} \tag{10}$$

TABLE I
PARALLEL LÓPEZ-DAHAB POINT DOUBLING ALGORITHM.

A Parallel approach of point doubling, LD-affine coordinates.		
Input: $P = (X_1 : Y_1 : Z_1)$ in LD coordinates on $E/K : y^2 + xy = x^3 + ax^2 + b$.		
Output: $2P = (X_3 : Y_3 : Z_3)$ in LD coordinates		
# cycle	C_0	C_1
1. cycle:	$Z_3 = X_1^2 \cdot Z_1^2$	$T_1 = Z_1^4$
2. cycle:	$T_2 = (X_1^4 + T_1) \cdot (Z_3 + Y_1^2 + T_1)$	$X_3 = X_1^4 + T_1$
3: cycle:	$Y_3 = T_1 \cdot Z_3 + T_2$	

In a software implementation, computing these Equations would require fifteen [3] clock cycles due to capabilities limitations. However in a hardware platform its is possible to compute those Equations in three clock cycles.

Therefore we can parallelize certain operations in a way that we can perform two operations at time, and by arranging the set of Equations 10 we only need three clock cycles in order to compute an elliptic curve point doubling as shown in Table I. We point out that compared to a field multiplication field squaring and field additions are less time cosuming.

2) Point Addition: The formulae for computing point addition are given below [14]. Point addition of $Q = (X_1, Y_1, Z_1)$ and $P = (X_2, Y_2, 1)$ can be performed at a computational cost of 8 field multiplications as,

$$\begin{array}{rclrcl} A & = & Y_2 \cdot Z_1^2 + Y_1; & B & = & X_2 \cdot Z_1 + X_1; \\ C & = & Z_1 \cdot B; & D & = & B^2 \cdot (C + aZ_1^2); \\ Z_3 & = & C^2; & E & = & A \cdot C; \\ X_3 & = & A^2 + D + E; & F & = & X_3 + X_2 \cdot Z_3; \\ G & = & (X_2 + Y_2) \cdot Z_3^2; & Y_3 & = & (E + Z_3) \cdot F + G \end{array} \tag{11}$$

In a software implementation, computing these Equations would require twenty six [3] clock cycles due to capabilities limitations. However in a Hardware platform its is possible to compute these Equations in eight clock cycles. Table II show us how these operations can be performed.

TABLE II
PARALLEL LÓPEZ-DAHAB POINT ADDITION ALGORITHM.

A parallel approach of point addition, LD-affine coordinates.
Input: $P = (X_1 : Y_1 : Z_1)$ in LD coordinates,
$Q = (x_2, y_2)$ in affine coordinates
on $E/K : y^2 + xy = x^3 + ax^2 + b$.
Output: $P + Q = (X_3 : Y_3 : Z_3)$ in LD coordinates

# cycle	C_0	C_1
1. cycle:	$Y_3 = y_2 \cdot Z_1^2 + Y_1$	
2. cycle:	$X_3 = x_2 \cdot Z_1 + X_1$	
3. cycle:	$T_1 = X_3 \cdot Z_1$	
4. cycle:	$X_3 = X_3^2 \cdot (Z_1^2 + T_1)$	$Z_3 = T_1^2$
5. cycle:	$X_3 = Y_3 \cdot T_1 + X_3 + Y_3^2$	$T_1 = Y_3 \cdot T1$
6. cycle:	$T_1 = x_2 \cdot Z_3 + X_3$	
7. cycle:	$Y_3 = (x_2 + y_2) \cdot Z_3^2$	$T_2 = T_3$
8. cycle:	$Y_3 = (T_2 + Z_3) \cdot T_1 + Y_3$	

Once again, we point out that field multiplication is by far the most time consuming arithmetic operation. Field addition can be time neglected in a hardware implementation.

Therefore we can parallelize some operations in such a way that we can perform two operations at a time. Hence, by rearranging the set of Equations 11 we only need eight clock cycles in order to compute an elliptic curve point addition as shown in Table II.

3) Point Halving: Point halving can be seen as the reverse operation of point doubling [4]. We can define the elliptic curve point halving as: let $P = (x_1, y_1), Q = (x_2, y_2)$ a point that belong to the curve 1, then we need to compute P such that $Q = 2P$ by solving the Equations:

$$\lambda^2 + \lambda = x_2 + a \tag{12}$$

$$x_1 = \sqrt{y_2 + x_2(\lambda + 1)} \tag{13}$$

$$y_1 = \lambda x_1 + x_1^2 \tag{14}$$

TABLE III
POINT HALVING ALGORITHM.

Point halving.
Input: $2P = (u, v)$
Output: $P = (x, y)$

1. Solve $\hat{\lambda}^2 + \hat{\lambda} = u + a$ for $\hat{\lambda}$
2. Find $T = v + u(\hat{\lambda} + 1)$
3. If $Tr(T) = 1$ then $\lambda = \hat{\lambda}, x = \sqrt{T}$ to t do
else $\lambda = \hat{\lambda} + 1, x = \sqrt{T + u}$
4. Find $y = \lambda x + x^2$
5. Return (x, y)

It is convenient to define the λ-*representation* of a point as: given $Q = (x, y) \in GF(2^m)$ let us define (x, λ_Q), where

$$\lambda_Q = x + \frac{y}{x} \tag{15}$$

Given the λ-representation of Q as the input to point halving, we may compute a point halving without converting to affine coordinates. In point multiplication, repeated halvings can be performed directly on λ-representation, with conversion to affine coordinates only when a point addition is required.

In [4] the algorithm shown in Table III is proposed for computing an elliptic point halving.

C. Scalar Multiplication

In Sections III-A and III-B, it was discuss how to perform efficient finite field arithmetic, and elliptic curve arithmetic. Now by using those operations we can perform elliptic curve scalar multiplication using the half-and-add method by instrumenting the algorithm shown in Table IV.

TABLE IV
HALF-AND-ADD ALGORITHM.

Half-and-add w-NAF (right-to-left) point multiplication.
Input: Window width w, $NAF_w = (2^{t-1} k mod n) = \sum_{i=0}^{t} k'_i 2^i$,
$P \in GF(2^m)$
Output: kP. (Note: $k = k'_0/2^{t-1} + \cdots + k'_{t-1} + 2k'_t mod n$)

1. Set $Q_i \leftarrow 0$ for $i \in I = 1, 3, \cdots, 2^{w-1} - 1$
2. If $k'_t = 1$ then $Q_1 = 2P$
3. For i from $t - 1$ downto 0 do:
   3.1 If $k'_i > 0$ then $Q_{k'_i} \leftarrow Q_{k'_i} + P$
   3.2 If $k'_i < 0$ then $Q_{-k'_i} \leftarrow Q_{-k'_i} - P$
   3.3 $P \leftarrow P/2$
4. $Q \leftarrow \sum_{i \in I} iQ_i$
5. Return(Q)

Summarizing, one can compute elliptic curve scalar multiplication operation by the half-and-add method, employing a new elliptic curve primitive: point halving

IV. IMPLEMENTATION

Proposed architecture for achieving elliptic curve scalar multiplication is shown in Figure 3. This architecture consists of two main blocks an ALU unit (that performs field arithmetic and elliptic curve arithmetic), and a control unit(that manage and controls the whole circuit).

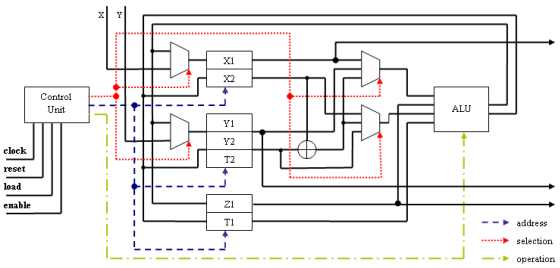

Fig. 3. Scalar multiplication circuit.

A. ALU unit

In order to write the operations previously mentioned (elliptic curve arithmetic) we have employed a two-port RAM for the memory block (Figure 3). BRAMs allows us to perform two write operations in a clock cycle.

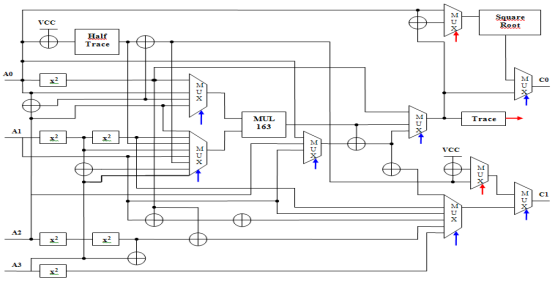

Fig. 4. ALU circuit.

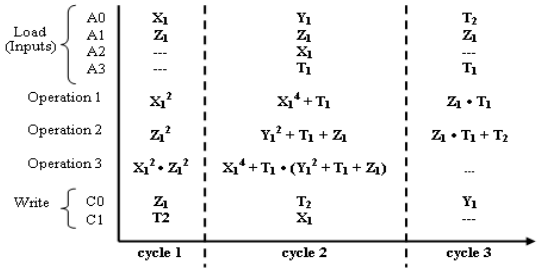

Fig. 6. Execution Point Doubling.

B. Control Unit

Architecture control unit has four signals entries, these signals are: clock, load, enable and reset. The clock signal synchronizes the whole circuit, while the load signal allows loading the input coordinates to the memory block in order to start the scalar multiplication operation. The enable signal is the signal that starts such operation, and finally the reset signal clear all the entries of the ALU circuit and memory blocks, i.e. initializing the architecture.

Table V, show which operations can be performed by the circuit per clock cycle. The first eight rows show the operations needed to compute an elliptic curve point addition. The next three rows show the operations needed to compute a elliptic curve point doubling. And the last three rows show the operation to compute a point halving (by λ-representation or in affine coordinates). The third column represents the word that allows the entire circuit to perform such operations. It consists of a twenty-five bit word. The first column represents the inputs given to the ALU circuit. And finally the last column shows ALU circuit output being written in memory.

The control word consists of 26 bits arrange as follows:

$$\underbrace{XX001010}_{direction}\ \underbrace{1100}_{MUX}\ \underbrace{100110010XXX1X}_{ALU}$$

as it can be see the first eight bits designate the directions to be read by the memory block, the next four bits designate which operand will be loaded to the ALU unit, and finally the last fourteen bits designate which operations will be performed by the ALU unit according to Table V.

C. Operations

As it was mentioned before, we can perform elliptic curve point addition in eight clock cycles as shown in Figure 5, and elliptic curve point doubling in three clock cycles as shown in Figure 6. In a first phase it is imperative to load the operands on the inputs of the ALU unit. Then, according to the control word the appropriate operations are selected and executed. We can classify operation execution in three main classes. First, operand's preprocessing before their fetching into the field multiplier unit. That pre-processing could be a square, add, square-add, square-square-add, square-add-add, half-trace, or half-trace-add operations.

ALU's main operation, namely, finite field multiplication is performed in combination with the storage of the corresponding results in circuit's outputs.

As an example, consider point halving computation in affine coordinates. First, it is necessary to load x_2, y_2 into inputs A_0, A_2. Then, the operations for loading $HT(A_0+1)$ and A_2 on the finite field multiplier are commanded by the control unit. Next, we multiply $A_2 \cdot HT(A_0+1)$. The result obtained by this multiplication is computed into the trace unit, in order to choose the appropriate operand for the square-root unit, and for sending to the corresponding outputs C_0, C_1. The dataflow just outlined can be seen in Figure 7.

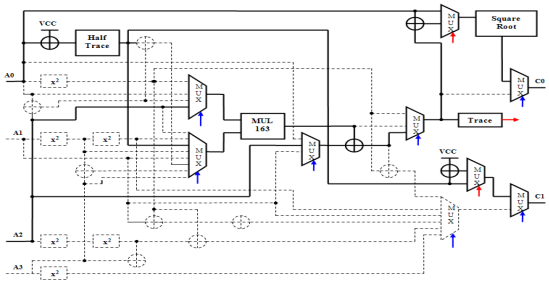

Fig. 7. Execution Point Halving.

V. Results and Comparison

As mentioned in previous Section, we can perform three main elliptic curve operations point addition, point doubling and point halving, Table VI show us how many cycles are required in order to perform such operations.

TABLE VI
CYCLES PER OPERATION

Elliptic curve operations	# cycles
Point Halving (affine coordinates)	1
Point Halving (λ-representation)	2
Point Doubling	3
Point Addition	8

A. Timings and Area Cost

All finite field arithmetic and then kP computational architectures were implemented on VirtexE XCV3200 by using Xilinx Foundation Tool F6.0i for design entry, synthesis, testing, implementation and verification of results. Table VII represents timing performances and occupied resources by the said architectures.

TABLE V
OPERATIONS ALLOWED BY THE ALU UNIT.

operation	input	control word	output
	$a_0a_1a_2a_3$	$s_{25}\cdots s_0$	c_0c_1
$Y_1 = y_2 \cdot Z_1^2 + Y_1$	$y_2Z_1Y_1-$	1xx01000xx11010000110xxx1x	Y_1x
$X_1 = x_2 \cdot Z_1 + X_1$	$x_2Z_1X_1-$	110xxxx0xx00010010110xxx1x	X_1x
$T_1 = X_1 \cdot Z_1$	X_1Z_1--	10xxxxx0x0xx01001xx00xxx1x	T_1x
$X_1 = X_1^2 \cdot (Z_1^2 + T_1)$	$X_1Z_1-T_1$	00xxxxx010xx00100xx0000111	X_1Z_1
$X_1 = Y_1 \cdot T_1 + X_1 + Y_1^2$	$y_2Z_1Y_1-$	0xx01000xx11010000110xxx1x	T_1X_1
$T_2 = x_2 \cdot Z_1 + X_1$	$x_2Z_1X_1-$	110xxxx0xx00010010110xxx1x	T_2x
$Y_1 = (x_2 + y_2) \cdot Z_1^2$	$x_2Z_1y_2-$	01xxx010xx0111000xx00xxx1x	Y_1x
$Y_1 = (T_1 + Z_1) \cdot T_2 + Y_1$	$Y_1T_1T_2Z_1$	0xx0010x1011100110010xxx1x	Y_1x
$Z_1 = X_1^2 \cdot Z_1^2$	X_1Z_1--	00xxxxx0x0xx00000xx0000011	Z_1T_2
$X_1 = (X_1^4 + T_1) \cdot (Y_1^2 + Z_1 + T1)$	$Y_1Z_1X_1T_1$	0x010xxxx10xxxxxxxx0101011	T_2X_1
$Y_1 = Z_1 \cdot T_1 + T_2$	$T_2Z_1-T_1$	00xxx101xx01010010110xxx1x	Y_1x
Point Halving (affines)	$x_2 - y_2-$	101xxx01xx01011010110xxx00	x_2y_2
Point Halving (λ-representation)	$x_2 - y_2-$	101xxx01xx0101110xx00xxx00	x_2y_2
$y_2 = \lambda x_2 + x_2^2$	$x_2 - y_2-$	101xxx01xx01010011010xxx1x	$-y_2$

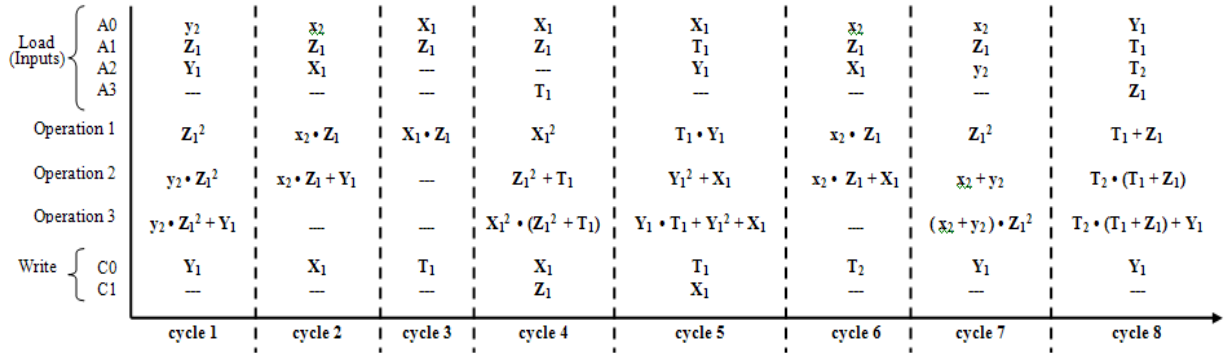

Fig. 5. Execution Point Addition.

TABLE VII
AREA COST AND TIMINGS.

Design	CLB slices	Timings
Karatsuba-Ofman Multiplier $GF(2^{163})$	6730	$21.620ns$
Half-trace function $GF(2^{163})$	1,258	14.621
Squaring unit $GF(2^{163})$	95	$7.938ns$
Square-root unit $GF(2^{163})$	328	$21.243ns$
ALU unit + control unit $GF(2^{163})$	11,063	$41.765ns$

B. Comparison

We estimate the running time of the circuit of Fig. 3 as follows. We need eight cycles for performing a point addition in mixed coordinates, however we need to convert a point from λ-representation to affine coordinates so we may perform 8 (point addition) + 3 (point halving + conversion) = 11 multiplications. Therefore, we employ a window NAF-method (in this case a NAF of two). Thus, the approximate cost would be given as,

$$\left(\frac{t}{w+1} - 2^{w-2}\right) 11M \tag{16}$$

where M is the cost of a finite field multiplication. Then the aproxematly running time of the implementation is : $0.025ms$.

Table VIII provides a comparison of several relevants FPGA's implementations of elliptic curve scalar multiplication over $GF(2^m)$. The design at [15] targeted a VLSI platform implementation. To our knowledge, that design is the only other hardware implementation of halve-and-add algorithm. All others design [7], [9], [16], [8] implements EC scalar multiplication on a single chip FPGA.

VI. CONCLUSIONS

In this work we present an architecture to compute elliptic curve scalar multiplication using the new algorithm half-and-add on FPGA platform. We have realize some optimizations to compute point addition in eight cycles. This architecture is proposed over the binary field $GF(2^{163})$ and perform an point multiplication under $0.025ms$. The architecture targets Xilinx VirtexE XCV3200 FPGA device and occupies 11,063 CLB slices.

Although our design has targeted the field generated by the irreducible polynomial $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$, all the modules discuss in this paper can be adapted to design another implementation with different fields sizes.

Future work includes further improvements in the design performance to make an full parallel circuit, the implementation of other design strategies and comparison between them.

REFERENCES

[1] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, Janury 1987.

TABLE VIII
COMPARISON TABLE.

Reference	Field	Platform	kP	Frequency	Area	Speed/Area
[15]	$GF(2^{178})$	VLSI	$4.4ms$	227Hz	143,000 gates	1.6×10^{-3}
[7]	$GF(2^{167})$	XCV400E	$0.21ms$	4.7MHz	3,002 LUTs	1.56
[9]	$GF(2^{163})$	XCV2000E	$0.143ms$	6.9MHz	-	-
[9]	$GF(2^{193})$	XCV2000E	$0.187ms$	5.3MHz	-	-
[16]	$GF(2^{163})$	XCV2000E	$0.075ms$	13.3MHz	10,017 LUTs	1.32
[8]	$GF(2^{191})$	XCV3200E	$0.056ms$	17.8MHz	19,626 slices	0.90
This work	$GF(2^{163})$	XCV3200E	$0.025ms$	41MHz	11,616 slices	3.52

[2] V. Miller. Use of elliptic curves in cryptography. In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[3] J. López and R. Dahab. Fast multiplication on elliptic curves over gf(2m) without precomputation. In *CHES '99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, pages 316–327, London, UK, 1999. Springer-Verlag.

[4] Darrel Hankerson, Julio López Hernandez, and Alfred Menezes. Software implementation of elliptic curve cryptography over binary fields. *CHES 2000*, 1965:1–24, August 17-18 2000.

[5] K. Fong, D. Hankerson, J. López, and A. Menezes. Field inversion and point halving revisited. In *IEEE Trans. Computers*, volume 53, pages 1047–1059, 2004.

[6] I. K. H. Leung and P. H. W. Leong. A microcoded elliptic curve processor using fpga technology. *IEEE Transactions on VLSI Systems*, 10(5):550–559, 2002.

[7] G. Orlando and C. Paar. A high performance reconfigurable elliptic curve processor for $gf(2^m)$. In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 41–56, London, UK, 2000. Springer-Verlag.

[8] N. A. Saqib, F. Rodríguez-Henríquez, and A. Díaz-Pérez. A parallel achitecture for fast computation of elliptic curve scalar multiplication over $gf(2^m)$. *Elsevier Journal of Microprocessors and Microsystems*, 28:329–339, 2004.

[9] N. Gura, S. C. Shantz, H. Eberle, S. Gupta, V. Gupta, D. Finchelstein, E. Goupy, and D. Stebila. An end-to-end systems approach to elliptic curve cryptography. In *Cryptographic Hardware an Embedded Systems - CHES 2002, 4th International Workshop*, pages 349–365, 2002. Revised Papers, 2523:349-365, August 2003.

[10] R. Schroeppel. Elliptic curve point ambiguity resolution apparatus and method. International Application Number PCT/US00/31014. filed 9 November 2000.

[11] E. W. Knudsen. Elliptic scalar multiplication using point halving. In *ASIACRYPT '99: Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security*, pages 135–149, London, UK, 1999. Springer-Verlag.

[12] F. Rodríguez-Henríquez and C. K. Koc. On fully parallel karatsuba multipliers for $gf(2^m)$. In *Proceedings of International Conference on Computer Science and Technology CST*, pages 405–410, May 19-21 2003. Acta Press, Cancún México.

[13] IEEE P1363. Standard specifications for public-key cryptography, draft version d18. IEEE standards documents, available at: http://grouper.ieee.org/groups/1363/, 2004.

[14] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2003.

[15] R. Schroeppel, C. L. Beaver, R. Gonzales, R. Miller, and T. Draelos. A low-power design for an elliptic curve digital signature chip. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 366–380, London, UK, 2003. Springer-Verlag.

[16] J. Lutz. High performance elliptic curve cryptographic co-processor. Master's thesis, University of Waterloo, 2004.