A Novel and Efficient Hardware Implementation of Scalar Point Multiplier

M. Masoumi* and H. Mahdizadeh*

Abstract: A new and highly efficient architecture for elliptic curve scalar point multiplication which is optimized for the binary field recommended by NIST is presented. To achieve the maximum architectural and timing improvements we have reorganized and reordered the critical path of the Lopez-Dahab scalar point multiplier carefully such that sequentially executed operations are separated into parallel operations and operations in the critical path are diverted to noncritical paths. With *G=41*, the proposed design is capable of performing a field multiplication over the extension field with degree 163 in 11.92 μs with the maximum achievable frequency of 251 MHz on Xilinx Virtex-4 (XC4VLX200) while 22% of the chip area is occupied, where *G* is the digit size of the underlying digit-serial finite field multiplier.

Keywords: FPGA Implementation, Lopez-Dahab Algorithm, Scalar Point Multiplication.

1 Introduction

Elliptic curve cryptography (ECC) is a public key cryptography system superior to the well-known RSA cryptography: for the same key size, it gives a higher security level than RSA [1, 2]. Intuitively, there are numerous advantages of using field-programmable gate-array (FPGA) technology to implement in hardware the computationally intensive operations needed for ECC. These advantages are comprehensively studied and listed by Wollinger, et. al. in [3]. In particular, performance, cost efficiency, and the ability to easily update the cryptographic algorithm in fielded devices are very attractive for hardware implementations [4-6]. Several recent FPGA-based hardware implementations of ECC have achieved high-performance throughput and efficiency. In this work we present a new architecture as well as an efficient ECC FPGA implementation over $GF(2^{163})$ that has considerable advantages compared to other implementations as regards to speed and area. The proposed architecture is based on a modified Lopez-Dahab elliptic curve point multiplication algorithm [7] in which we have reorganized and reordered the data path carefully to achieve maximum performance and efficiency. As we know, the efficiency of an algorithm is measured by the scarce resources it consumes. Typically the measure used is time, but sometimes other measures such as space and number of processors are also considered. Our basic strategy for architectural timing improvement is to reorganize the critical path such that logic structures are implemented in parallel. Usually, this technique is used whenever a function that currently evaluates through a serial string of logic can be broken up and evaluated in parallel. By using a modified field multiplier and two squarer modules for separating the paths in which squaring is repeated several times we have designed an efficient architecture for the Itoh-Tsujii Multiplicative Inverse Algorithm (ITMIA) [8]. In the design of the ECC processor, we have separated sequentially executed operations into parallel operations and have carefully reordered paths to divert operations in the critical path to noncritical paths in order to minimize the combinatorial delay of the critical path. The architecture of the ECC processor has been designed in such a way that the calculations of point addition are separated and are performed independent of the key which in turn considerably reduces the processing delay. The results we obtained show that by using the mentioned optimization techniques and by implementing a modified *G*-bit digit serial finite-field multiplier, with $G = 41$ our proposed design is able to compute $GF(2^{163})$ elliptic curve scalar point multiplication operations in 11.92 μs with the maximum achievable frequency of 251 MHz on Xilinx Virtex-4 (XC4VLX200) while 19606 slices or 22% of the chip area is occupied which makes the design suitable for high speed applications. The organization of the article is as follows: In Section 2, a brief introduction of the mathematical background of ECC is presented. In

Iranian Journal of Electrical & Electronic Engineering, 2012.
Paper first received 4 May 2012 and in revised form 10 Nov. 2012.
* The Authors are with Islamshahr Azad University, P.O. Box: 33135-369, Sayad Shirazi Ave., Namaz Sqr., Islamshahr, Tehran, Iran.
E-mails: m_masoumi@eetd.kntu.ac.ir, h.mahdizadeh@yahoo.com.

Section 3, some previous works are reviewed. In Section 4, the algorithm optimization decomposition in parallel and resource occupation for implementation of the modular arithmetic logic unit and the finite field arithmetic units in hardware are detailed. In Section 5, the proposed architecture for ECC processor is illustrated. In Section 6, implementation results and performance obtained are compared with those in other published works. Finally, in the conclusions we summarize the results of our discussions.

2 Mathematical Background

2.1 Mathematical Background for Elliptic Curves

A finite field $GF(2^m)$ consists of 2^m elements, together with addition and multiplication operations that can be defined over polynomials. For elliptic curves over $GF(2^m)$ we use a cubic equation in which the variables and coefficients all take values in $GF(2^m)$ [1, 2, 9, 10]. It has been turned out that the form of cubic equation appropriate for cryptographic applications for elliptic curves which has been recommended by NIST is

$$y^2 + xy = x^3 + ax^2 + b \left(mod\ P(x)\right) \quad (1)$$

where it is understood that the variables x and y and the coefficients a and b are elements of $GF(2^m)$ and calculations are performed in $GF(2^m)$. Let us consider the finite field $GF(2^{163})$ generated using the irreducible polynomial $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$ which is the NIST recommended field for ECC applications. An elliptic curve group over $GF(2^m)$ consists of the points on the corresponding elliptic curve, together with a point at infinity, $\mathcal{O}$. The set of points that satisfy the Eq. (1) together with the element $\mathcal{O}$ forms an addition Abelian group with respect to the elliptic point addition operation. $\mathcal{O}$ serves as the additive identity. Thus, $\mathcal{O} = -\mathcal{O}$ and for any point P on the elliptic curve, $P + \mathcal{O} = P$ *and* $P + (-P) = \mathcal{O}$. It can be shown that a finite Abelian group can be defined based on the set $E_{2^m}(a,b)$, provided that $b \neq 0$. The rules for addition can be stated as follows. For all points $P, Q \in E_{2^m}(a, b)$:

1) $P + \mathcal{O} = P$.
2) If $P = (x_P, y_P)$, then $-P + (x_P, y_P) = \mathcal{O}$. The point $(x_P, x_P + y_P)$ is the negative of P, denoted as $-P$.
3) If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ with $P \neq Q$ and $P \neq -Q$, then $R = P + Q = (x_R, y_R)$ is determined by Eq. (2).

$$x_R = \lambda^2 + \lambda + x_P + x_Q + a \quad (2)$$
$$y_R = \lambda(x_P + x_R) + x_R + y_P$$

where $\lambda = \frac{y_Q + y_P}{x_Q + x_P}$

4) If $P = (x_P, y_P)$ then $R = 2P = (x_R, y_R)$ is determined by Eq. (3).

$$x_R = \lambda^2 + \lambda + a \quad (3)$$
$$y_R = x_P^2 + (\lambda + 1)x_R$$

where $\lambda = x_P + \frac{y_P}{x_P}$

2.2 Elliptic Curve Cryptography

The addition operation in ECC is the counterpart of modular multiplication in RSA, and multiple additions is the counterpart of modular exponentiation. To multiply a point by a constant, the points must be added continuously with attention to the rule mentioned in section 2.1 for $R = 2P$. If k is a positive integer and P a point on an elliptic curve, the scalar multiple $Q = kP$ is the point resulting of adding k copies of P to itself. Scalar multiplication is by far the most important operation of elliptic curve cryptosystems. The hierarchy of arithmetic for an Elliptic Curve point multiplication is depicted in Fig. 1 [9].

In order to generate an Abelian group over elliptic curves, it is necessary to define an elliptic curve group law. More specifically, we defined the point addition and point doubling primitives of Eqs. (2, 3). However, the computational cost of those equations involves the calculation of a costly field inverse operation plus several field multiplications. Hence, there is a strong motivation for finding alternative point representations that allow the trading of the costly field inversions by less expensive field multiplications.

It has been shown that the points on an elliptic curve can be represented using either two or three coordinates [2]. In affine-coordinate representation, a finite point on $E(GF(2^m))$ is specified by two coordinates $x, y \in GF(2^m)$ satisfying Eq. (2). The point at infinity has no affine coordinates. We can make use of the concept of a projective plane over the field $GF(2^m)$. In this way, one can represent a point using three rather than two coordinates. Then, given a point P with affine-coordinate representation x, y there exists a corresponding projective-coordinate representation X, Y and Z such that, $P(x;y) = P(X;Y;Z)$.

As a means of avoiding the expensive field inversion operation, it is more convenient to work with Lopez-Dahab (LD) projective coordinates which is highly attractive for hardware implementation.

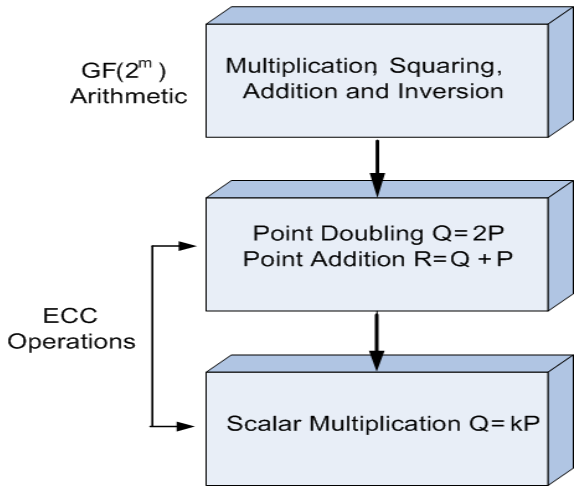

Fig. 1 Three-layer model for elliptic curve scalar multiplication.

By using the Lopez-Dahab algorithm the projective group law can be implemented without utilizing field inversions at the price of increasing the total number of field multiplications. As a matter of fact, field inversions are only required when converting from projective representation to affine representation, which becomes valuable in situations where we are planning to perform many point additions and doublings in a successive manner, such as in elliptic curve scalar multiplication. The Lopez-Dahab algorithm is shown in Fig. 2. This algorithm could be divided into three parts. In the first part, coordinates of the input point is converted to their corresponding projective coordinates. In the second part, main operations of the algorithm, i.e., point doubling and addition are performed based on the key bits and in the third part, coordinates of the output point, $Q=kP$, is converted again to their corresponding affine coordinates. 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.

INPUT: $k = (k_{t-1}, \ldots, k_1, k_0)_2$ with $k_{t-1} = 1$, $P = (x_P, y_P) \in E(F_2^m)$.

OUTPUT: kP.

1. $X_1 \leftarrow x_P$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x_P^4 + b$, $Z_2 \leftarrow x_P^2$. {Compute $(P, 2P)$}

2. For i from $t-2$ downto 0 do

2.1 If $k_i = 1$ then

$T \leftarrow Z_1$, $Z_1 \leftarrow (X_1 Z_2 + X_2 Z_1)^2$, $X_1 \leftarrow x_P Z_1 + X_1 X_2 T Z_2$.

$T \leftarrow X_2$, $X_2 \leftarrow X_2^4 + b Z_2^4$, $Z_2 \leftarrow T^2 Z_2^2$.

2.2 Else

$T \leftarrow Z_2$, $Z_2 \leftarrow (X_1 Z_2 + X_2 Z_1)^2$, $X_2 \leftarrow x_P Z_2 + X_1 X_2 Z_1 T$.

$T \leftarrow X_1$, $X_1 \leftarrow X_1^4 + b Z_1^4$, $Z_1 \leftarrow T^2 Z_1^2$.

3. $x_3 \leftarrow X_1 / Z_1$.

4. $y_3 \leftarrow (x_P + X_1/Z_1)[(X_1 + x_P Z_1)(X_2 + x_P Z_2) + (x_P^2 + y)(Z_1 Z_2)](x_P Z_1 Z_2)^{-1} + y_P$.

5. Return (x_3, y_3)

Fig. 2 The Lopez-Dahab scalar point multiplication over $GF(2^m)$ [2].

3 Previous Works

Several recent FPGA-based hardware implementations of ECC have achieved high-performance throughput. Various acceleration techniques have been used, usually based on parallelism or precomputation.

The work introduced by Orlando and Paar [11] is based on the Montgomery method for computing KP developed by Lopez and Dahab and operates over a single field. A point multiplication over $GF(2^{167})$ is performed in 210ms, using a Galois field multiplier with an eleven-cycle latency. An ECC processor capable of operating over multiple Galois fields was presented by Gura, et al. [12], which performs a point multiplication over in 143 μs, using a Galois field multiplier with three-cycle latency. Gura, et al. stated two important conclusions: the efficient implementation of inversion has a significant impact on overall performance and, as the latency of multiplication is improved, system tasks such as reading and writing have a significant impact on performance. The work introduced by Jarvinen, et al. [13] uses two bit-parallel multipliers to perform multiplications concurrently. The multipliers have several registers in the critical path in order to operate at high clock frequencies, but the operations are not pipelined which results in a large design and high latency. Rodriguez, et al. [14] introduced an FPGA implementation that performs DBL and ADD in parallel, containing multiple instances of circuits to perform the arithmetic functions. It would appear that the inversion required for coordinate conversion at the end of the point multiplication is not performed, and that the quoted point multiplication time does not include the coordinate conversion, but the stated point multiplication time is one of the fastest in the literature. However, the complex structure of the multiplier has a long critical path, and as a result the overall performance is let down by quite a low clock frequency (46.5 MHz). Cheung, et al. in [15] presented a hardware design that uses a normal basis representation. The customizable hardware offers a trade between cost and performance by varying the level of parallelism through the number of multipliers and level of pipelining. This implementation completes the scalar point multiplication in approximately 55μs, although once again the low clock frequency (43 MHz) limits the potential performance. In [16], point multiplication is compared for supersingular and non supersingular curves. The result of the implementation on Virtex II Pro 30 is 280 μs for point multiplication at 100 MHz frequency with 8450 occupied slices. In [17] Chelton and Benaissa presents a pipelined Application-Specific Instruction set Processor (ASIP) for ECC using FPGAs which achieves a point multiplication time of 33.05 μs at 91 MHz on a Xilinx Virtex-E FPGA and 19.55 μs at 159.3 MHZ on the same platform that we used in this work, Xilinx Virtex-4 (XC4VLX200) with 16209 occupied slices. Ansari and Hasan in [18] propose an architecture for elliptic curve scalar multiplication based on the Montgomery ladder method over finite field $GF(2^m)$. The authors propose a pseudopipelined word-serial finite field multiplier, with word size w, suitable for the scalar multiplication. They implement their design on Xilinx XC2V2000. They are able to compute $GF(2^{163})$ elliptic curve scalar multiplication operations in 46.5 μs with the maximum achievable frequency of 100 MHz on Xilinx Virtex-4 (XC4VLX200). The most notable feature of their design is that it is very compact as utilizes only 7559 LUTs. Yong et al. in [19] concentrate on the high-speed hardware implementation

of ECC over $GF(2^{163})$ in FPGA. They optimize the algorithm in parallel and design a new architecture for ECC point multiplication. Their design reach to 93.3 MHz speed on Xilinx XC2V6000 and is able to perform random elliptic curve scalar point multiplication over $GF(2^{163})$ in 34.11 μs. In [20] Kim et. al. present a high speed implementation based on Gaussian normal bases in which their proposed design is able to complete a scalar point multiplication in 10 μs but the design is large as it occupies 24,363 slices on Xilinx XC4VLX80.

4 Hardware Architectures for Finite Field Operations Over $GF(2^m)$

In this section we will briefly describe some algorithms and techniques that we used for efficient implementation of finite field and modular arithmetic operations over $GF(2^m)$. They will be used in realization of the scalar point multiplier architecture. Field addition and subtraction in $GF(2^m)$ are not investigated since they are defined as polynomial addition and can be implemented simply as the XOR addition of the two m-bit operands [2, 21].

4.1 Finite Field Reduction

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ and let $A(x)$ and $B(x) \in GF(2^m)$. Assuming that we have already computed the product polynomial $D(x)= A(x)B(x)$ and we want to obtain the modular product of $C(x)$ such that

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

Recall that the polynomial product D and the modular product C; have $2m-1$ and m; coordinates, respectively, i.e.,

$$D = [d_{2m-2}, d_{2m-3}, \dots, d_{m+1}, d_m, \dots, d_1, d_0];$$
$$C = [c_{m-1}, c_{m-2}, \dots, c_1, c_0]; \tag{5}$$

Once the generating polynomial $P(x)$ has been selected, the reduction step that obtains C from D can be computed by using XOR and shift operations only. The reduction modulo $P(x)$ can be viewed as a linear mapping of the $2m-1$ coefficients of $D(x)$ into the m coefficients of $C(x)$. This mapping can be represented in matrix notation as follows:

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 & r_{0,0} & \dots & r_{0,m-2} \\ 0 & 1 & \dots & 0 & r_{1,0} & \dots & r_{1,m-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & & 1 & r_{m-1,0} & & r_{m-1,m-2} \end{bmatrix} \begin{bmatrix} d_0 \\ \vdots \\ d_{m-1} \\ d_m \\ \vdots \\ d_{2m-2} \end{bmatrix} \tag{6}$$

where

$$r_{j,i} = \begin{cases} P_j \,;\ j = 0, \cdots, m-1; i = 0 \\ r_{j-1,i-1} + r_{m-1,i-1} r_{j,0} \,;\ j = 0, \cdots, m-1; i = 1, \cdots, m-2 \end{cases} \tag{7}$$

Implementation of the above matrix for different values of m will increase the required logic gates and area. One of the most efficient approaches for hardware implementation of finite field reduction is reduction using fast reduction algorithm corresponding to field polynomial. Fig. 3 represents the implementation of the reduction modulo $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$. It has been assumed that the maximum degree of $D(x)$ is equal to $162+G$ in which the sentences with degree $163 \le i \le 162+G$ are mapped to the sentences with degree $i < 163$.

4.2 Finite Field Multiplication

As mentioned before, field multiplication is by far the most costly arithmetic operation which directly affects the working frequency and speed of the ECC processor. One can make an speed-area trade-off by using a serial-parallel strategy, in which multiplication of two arbitrary field elements is accomplished by using a procedure inspired in the well-known digit-serial/parallel (LSD) finite field multipliers [21, 22]. In this work, we have designed LSD multiplier directly at digit-level.

Based on [22], LSD multiplication algorithms are classified as least significant digit (LSD) first and most significant digit (MSD) first algorithms. It has been shown that the LSD first algorithm consumes fewer gates and has shorter critical path compared with the MSD first algorithm. Various approaches have been proposed for efficient implementation of the LSD multiplier. With digit size G, the total number of digits in $GF(2^m)$ will be $n = \lceil m/G \rceil$. Assume $A = \sum_{j=0}^{m-1} a_j \alpha^j$ and $B = \sum_{j=0}^{m-1} b_j \alpha^j$ such that

$$B_i = \begin{cases} \sum_{j=0}^{G-1} b_{G*i+j} \alpha^j & 0 \le i \le n-2 \\ \sum_{j=0}^{m-1-G(n-1)} b_{G*i+j} \alpha^j & i = n-1 \end{cases} \tag{8}$$

Input: D = $[d_{162+G}, d_{161+G}, \dots, d_1, d_0]$,
$P(x) = x^{163} + x^7 + x^6 + x^3 + 1$;
Output: $C = [c_{162}, c_{161}, \dots, c_1, c_0]$;

if $G = 0$ then
    $C \leftarrow D$;
else
    $[c_{162}, \dots, c_G] \leftarrow [d_{162-G}, \dots, d_0]$;
    $[c_{G-1}, \dots, c_0] \leftarrow 0$;

    for i from 1 to G do
        $c_{i-1} \leftarrow c_{i-1}\ xor\ d_{163+i-1}$;
        $c_{3+i-1} \leftarrow c_{3+i-1}\ xor\ d_{163+i-1}$;
        $c_{6+i-1} \leftarrow c_{6+i-1}\ xor\ d_{163+i-1}$;
        $c_{7+i-1} \leftarrow c_{7+i-1}\ xor\ d_{163+i-1}$;

Return C;

Fig. 3 Reduction algorithm for $C(x) = D(x) \bmod P(x)$

$$C = A * B \bmod P(x) = \sum_{j=0}^{m-1} c_j \alpha^j$$

$$= \begin{pmatrix} B_0 A + B_1\left(A\alpha^G \bmod f(x)\right) + B_2\left(A\alpha^G \cdot \alpha^G \bmod P(x)\right) \\ + B_{n-1}\left(A\alpha^{G*(n-2)} \cdot \alpha^G \bmod P(x)\right) \end{pmatrix} \bmod P(x) \quad (9)$$

The LSD algorithm is summarized in Fig. 4. Consider the two-step classical multiplication over $GF(2^m)$ which involves in a polynomial multiplication and a reduction modulo an irreducible polynomial. The product of the polynomials $A(x)$ and $B(x)$, $D(x)=A(x)\times B(x)$, is a polynomial with maximum degree $2m-2$ and can be written as follows.

$$\begin{cases} \sum_{i=0}^{k} a_i b_{k-i} \; ; \; k = 0, \cdots, m-1 \\ \sum_{i=k}^{2m-2} a_{k-i+(m-1)} b_{i-(m-1)} \; ; \; k = m, \cdots, 2m-2 \end{cases} \quad (10)$$

We implemented the above scheme in a matrix form. Thus, we put A in a three-section multiplicand matrix. The upper part is a lower triangular submatrix. The middle part is a $(m-G+1)\times G$ submatrix. The lower part is an upper triangular submatrix.

$$\begin{bmatrix} a_0 & 0 & 0 & \cdots & 0 \\ a_1 & a_0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ a_{G-2} & a_{G-3} & \cdots & a_0 & 0 \\ a_{G-1} & a_{G-2} & \cdots & a_1 & a_0 \\ a_G & a_{G-1} & \cdots & a_2 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ a_{m-1} & a_{m-2} & \cdots & a_{m-(G-1)} & a_{m-G} \\ 0 & a_{m-1} & a_{m-2} & \cdots & a_{m-(G-1)} \\ 0 & 0 & a_{m-1} & \cdots & a_{m-(G-2)} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & a_{m-1} \end{bmatrix} * \underset{G*1}{\begin{bmatrix} b_0 \\ \vdots \\ b_{G-1} \end{bmatrix}} = \underset{(m+G-1)*1}{\begin{bmatrix} d_0 \\ \vdots \\ d_{m+G-2} \end{bmatrix}} \quad (11)$$

Input: $A,B \in GF(2^m)$
Output: $C \in GF(2^m)$, $C = AB$ over $GF(2^m)$
Set: $A^{(0)} = a$, $D^{(0)} = 0$, $n = \lceil m/G \rceil$
for i from 1 to n do
1) $A^{(i)} = A^{(i-1)}\alpha^G \bmod P(x)$,
2) $D^{(i)} = A^{(i-1)} \cdot B_{i-1} + D^{(i-1)}$
Where
$A^{(i)} = \sum_{j=0}^{m-1} A_j^{(i)} \alpha^j$
$D^{(i)} = \sum_{j=0}^{m+G-2} d_j^{(i)} \alpha^j$ and
$B_i = \begin{cases} \sum_{j=0}^{G-1} b_{G*i+j}\alpha^j & 0 \le i \le n-2 \\ \sum_{j=0}^{m-1-G*(n-1)} b_{G*i+j}\alpha^j & i = n-1 \end{cases}$
end for
3) **Return** $C = D^{(n)} \bmod P(x)$

Fig. 4 The LSD multiplication algorithm [22].

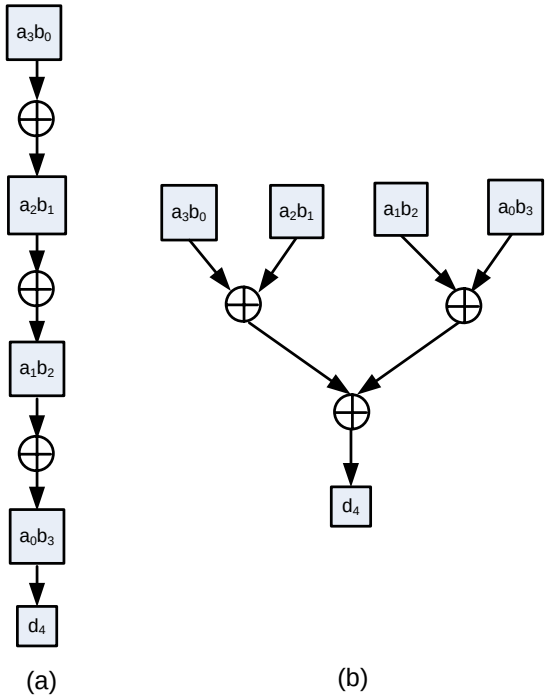

Fig. 5 XORing four bits together, (a) direct method, and (b). by using a binary graph tree.

By converting Eq. (10) into the matrix form of Eq. (11), the G^{th} term of the polynomial $D(x)$ or d_G can be expressed as Eq. (12).

$$d_G = a_{G-1}b_0 + a_{G-2}b_1 + \cdots + a_0 b_{G-1} \quad (12)$$

where G is the digit size of the underlying LSD multiplier. If each terms of Eq. (12) is considered as a bit string, d_G could be obtained by a binary XOR of the terms of this equation.

As it is seen in Fig. 5(a), using this method will lead to a critical path with length $(G\text{-}1)\Delta_{XOR}$, where Δ_{XOR} is the delay of an XOR gate. Now, if we use a binary tree graph for XORing this bit string as it is seen in Fig. 5(b), by placing the binary bits at the head branches of the tree we can reach to the result or root of the tree with path length of $(\lceil log_2(G) \rceil)\Delta_{XOR}$. We used this idea to reduce the critical path of the LSD multiplier.

Now, we describe the implementation architecture for the LSD multiplier. As it was seen, there are three steps for implementing this algorithm. Steps 1 and 2 of the LSD multiplier as is represented in Eq. (13) can be implemented in parallel.

$$\begin{aligned} A^{(i)} &= A^{(i-1)}\alpha^G \bmod P(x) \\ D^{(i)} &= A^{(i-1)} B_{i-1} + D^{(i-1)} \end{aligned} \quad (13)$$

Step 1 of the LSD algorithm reduces $m+G$ bits to m bits and step 2 shows the partial products. For obtaining the final result, $m+G\text{-}1$ bits is reduced to m bits in step 3. The implementation architecture for different stages of the LSD multiplier is depicted in Fig. 6. Step 1 is performed by the left side of Fig. 6, step 2 is performed by the multiplication function and step 3 is performed by the right side of Fig. 6.

4.3 Finite-Field Multiplicative Inversion

Based on Fermat's Little Theorem (FLT) and using an ingenious rearrangement of the required field operations, the Itoh-Tsujii Multiplicative Inverse Algorithm (ITMIA) was presented in [8]. The main advantage of ITMIA algorithm in comparison with the Extended Euclidian Algorithm is that it does not require a separate inversion module. When computing the multiplicative inverse using ITIMA algorithm, we need to iteratively perform 81 squaring in the algorithm's addition chain. Since these iterative computations are done sequentially, further parallelism is not possible [2]. This algorithm is briefly illustrated in appendix.

Now, to design an efficient multiplicative inversion block based on the ITMIA, it is necessary to think how to reduce its critical path. In other words, the critical path of the multiplier and the critical path of the inversion block should be along with each other. If we use only one squarer module in the inversion block, this module should accomplish squaring for the input of the inversion block, output of the multiplier and also its own output (for consecutive squaring) and therefore, we are forced to use a 3 to 1 multiplexer at the input of the squarer. Output of this squarer together with a number of combinational gates such as AND, OR, and NOT gates are connected to the input of the multiplier. As a result of such architecture, the critical path will place on the squarer which will create a bottleneck for reducing the clock cycle time. We can break this critical path by changing the architecture so that a 2 to 1 multiplexer is used in place of a 3 to 1 multiplexer at the cost of adding another squarer in the inverter architecture. The first squarer is used for squaring in stages 1, 3, 8 and also for the final stage squaring, while the other required squaring in Table A.1 (See Appendix for more detail) is accomplished with the second squarer [2]. Thanks to, in stages 1, 3, 8, $u_0 = 1$ and only one squaring needs to be performed while at the other stages several squaring are performed (see appendix for more details). The schematics of the designed architecture for multiplicative inversion over finite field $GF(2^{163})$ is shown in Fig. 7.

5 Proposed Architecture for the ECC Processor

As was mentioned, the most important strategy for architectural timing improvements is to reorganize and reorder the critical path such that logic structures are implemented in parallel and to divert operations in the critical path to a noncritical path. This technique should be used whenever a function that currently evaluates through a serial string of logic can be broken up and evaluated in parallel. This assumption can dramatically speed up the implementation of a large design. For the design of architecture for ECC scalar multiplier, two different parts are considered; the first part that involves in calculations in the affine coordinate system and the other part that involves in the calculations for converting projective coordinate to affine coordinates. For projective calculations, parts 1 and 2 of the LD algorithm are considered. In the design of this part of the processor, the number of computational units is chosen in such a way that allows parallel computations to be performed. Hence, we use three field multipliers to implement the main loop of the algorithm in which point addition and doubling are carried out. So, according to Section 2.1 of the LD algorithm, at the first stage, the three multiplications X_1Z_2, X_2Z_1, TZ_2 ($T \rightarrow X_2$) are performed in parallel by using three multipliers as is shown in Fig. 6, and then, the three other multiplications $x_P Z_1$, $X_1X_2T\,Z_2$ ($T \leftarrow Z_1$), bZ_2^4 are accomplished in parallel at the second stage. Hence, the delay of each iteration is reduced from six field multiplication delay to two field multiplications. For this part of the processor (computations in the projective coordinates) we have used five squarers and two adders, as is shown in Fig. 6. Four squarers are used for computing Z_1^4, X_1^4, Z_2^4 and X_2^4 while the fifth squarer is used for $(X_1Z_2+X_2Z_1)^2$. In addition, It is essential after the first field multiplication to save the result of $(X_1Z_2+X_2Z_1)^2$ and $(X_2^4+bZ_2^4)$ in the registers t_1 and t_2 respectively for the later calculations. The most important modules in the design of the scalar point multiplier processor are field multiplication, field inversion and field squaring. The key point here is that the critical path must be placed on the longest path among these modules. Since the inverter module was designed such that its critical path is coincided with the multiplier's critical path and since the multiplier's path is larger than the squarer's path, the critical path need to be placed on the multiplier.

It is notable that if resource sharing is used in implementing the field squarer, the number of required computational elements will decrease; however, since for squaring of different values we are forced to use multiplexers at the input of this computational unit that are controlled with conditional statements, the critical path length will increase. To avoid long critical path, the architecture should be designed synchronous and by using combinational logic. In addition, in the design of the projective calculations, separate calculations have not been performed for using the initial values of part 1 of the LD algorithm, since if further computational modules are designed for these calculations, the complexity of the critical path and the amount of required area will increase. We can avoid additional or unnecessary calculations by using calculations of part 2 of the algorithm for obtaining the results for part 1. In the proposed design, calculations of part 1 need to be performed whenever the most significant bit of the key is 1. So, when $k_i = 1$, if the values of Eq. (14) are used in the calculations of part 2.1, then the required initial values of the LD algorithm are obtained in accordance with part 1 of the LD algorithm.

$$X_1 \leftarrow 1,\ Z_1 \leftarrow 0,\ X_2 \leftarrow x_P,\ Z_2 \leftarrow 1 \tag{14}$$

The results of the calculation in section 2.1 of the LD algorithm are obtained as Eq. (15) by using the values of Eq. (14).

$$X_1 \leftarrow x_P,\ Z_1 \leftarrow 1,\ X_2 \leftarrow x_P^4 + b,\ Z_2 \leftarrow x_P^2 \quad (15)$$

As it is seen in Fig. 7, whenever the key bit is equal to 1, the values of '1', '0', and x_P are entered into the multiplexers to connect to the appropriate inputs to make the terms of Eq. (15). After designing the computational units for projective coordinates, its input and output ports should be connected together based on the key bits to complete the iteration in the LD algorithm. When designing the architecture for calculations in the projective coordinate system in part 2.1 of the LD algorithm, to set up part 2.2 of the algorithm which works with zero bits of the key, it is enough to swap X_1 and Z_1 with X_2 and Z_2 respectively when the key bits change. So, we need to use a 2 to 1 multiplexer that is controlled with the key bits. Therefore, in order to avoid long critical path, another strategy should be considered. As it is seen from the architecture of Fig. 8, in order to prevent further complexity when swapping X_1 and Z_1 with X_2 and Z_2, the input-output paths of point addition and doubling have been separated from each other. The idea behind this subject is to connect the outputs of point addition and doubling to the inputs of the adder, independent of the values of the key bits .For example, if we consider the following point addition operation for k_i =1, inputs to this operation are X_1, X_2, Z_1 and Z_2 and outputs are saved in X_1 and Z_1.

$$T \leftarrow Z_1,\ Z_1 \leftarrow (X_1Z_2 + X_2Z_1)^2,\ X_1 \leftarrow x_P Z_1 + X_1X_2TZ_2 \quad (16)$$

When a key bit changes from $k_i = 1$ to $k_i = 0$, this change will lead to change in the terms $X_1Z_2+X_2Z_1$ and $X_1X_2TZ_2$. However, since whenever the value of any key bit changes only X_1 and Z_1 are swapped with X_2 and Z_2, the terms $X_1Z_2+X_2Z_1$ and $X_1X_2TZ_2$ will remain unchanged. So, the point addition operation can be repeated in the iterative part of the algorithm without involvements of the key bits and only after the end of the loop, the registers are swapped with each other. The point doubling operation for k_i = 1 is performed in accordance with Eq. (17).

$$T \leftarrow X_2,\ X_2 \leftarrow X_2^4 + bZ_2^4,\ Z_2 \leftarrow T^2Z_2^2 \quad (17)$$

This operation for $k_i = 0$ is done by swapping X_2 and Z_2 with X_1 and Z_1 respectively. Therefore, output registers are swapped in order to provide proper inputs for the point doubling operation based on the key bits in the iterative part of the LD algorithm. In order to realize that when the initial values are entered into the calculations and also to be aware of the iterations of the LD algorithm based on the key bits, it is necessary to combine the module designed in Fig. 8 with a key shift register in a new structure. The aim of this work is that the inputs and outputs of the architecture of Fig. 8 are properly connected to each other when all values of the key are scanned. The new design is shown in Fig. 9. The second part of the processor involves in calculations that convert projective coordinates to affine coordinates. It is obvious from the LD algorithm that parts 3 and 4 of this algorithm require many calculations to be implemented.

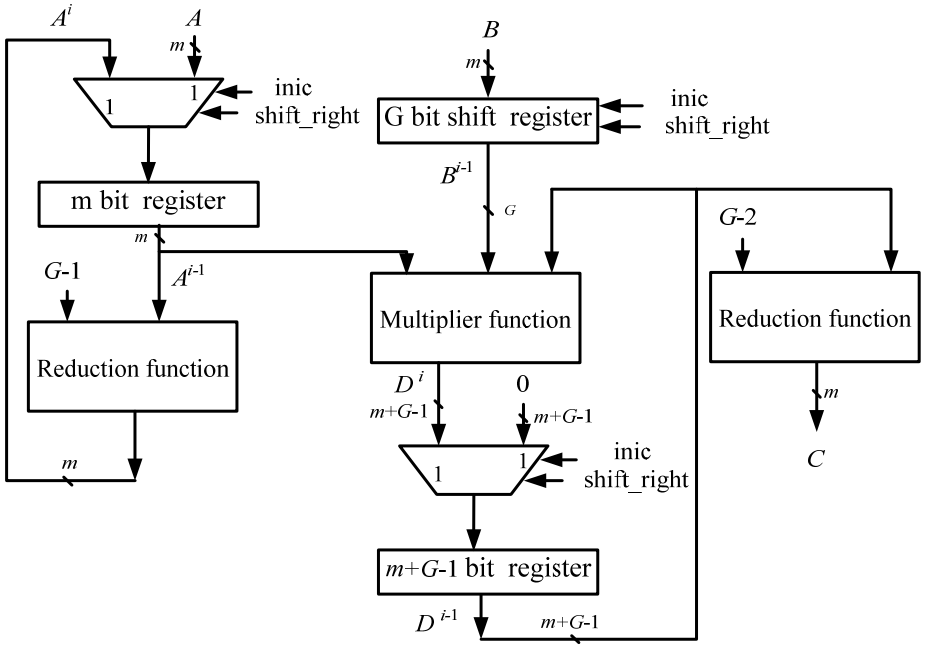

Fig.6 Block diagram of the LSD multiplier implemented in this work.

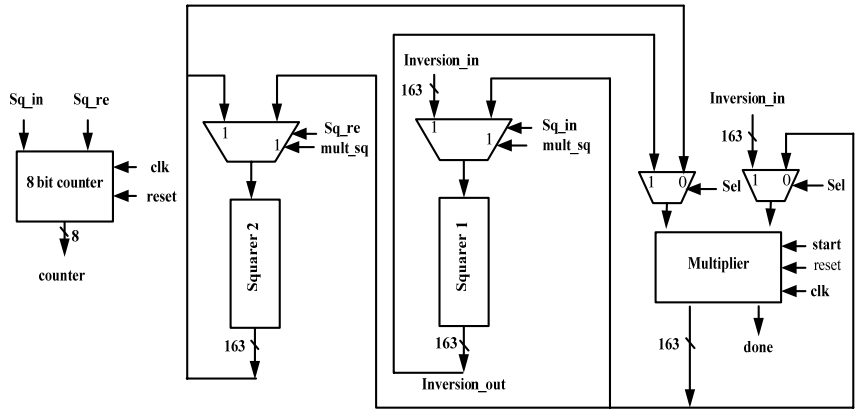

Fig. 7 Schematic of the designed architecture for finite field multiplicative inversion.

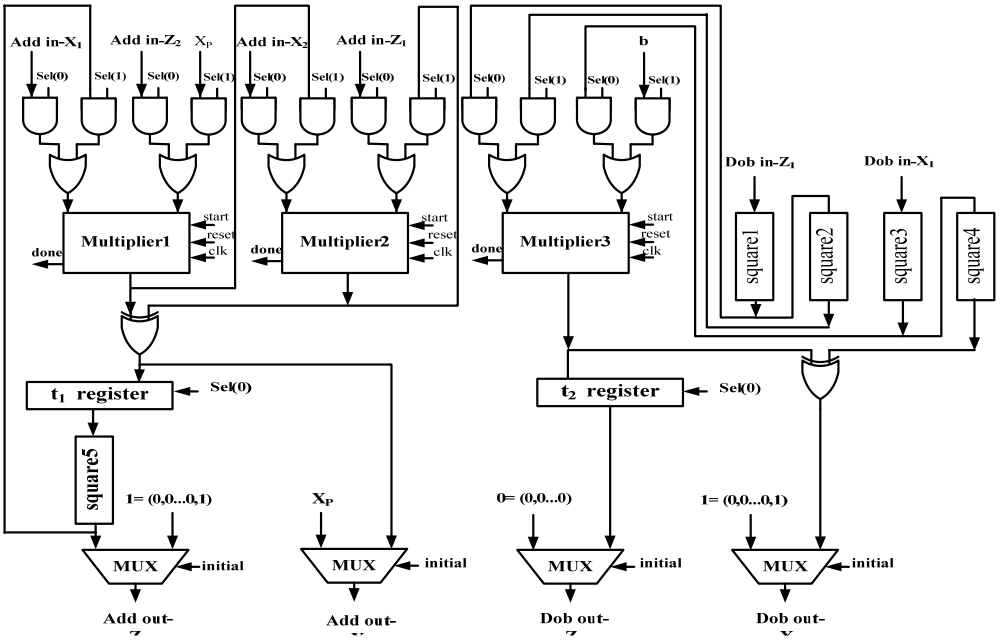

Fig. 8 The architecture designed for the computation of point addition and point doubling in projective coordinates of the LD algorithm.

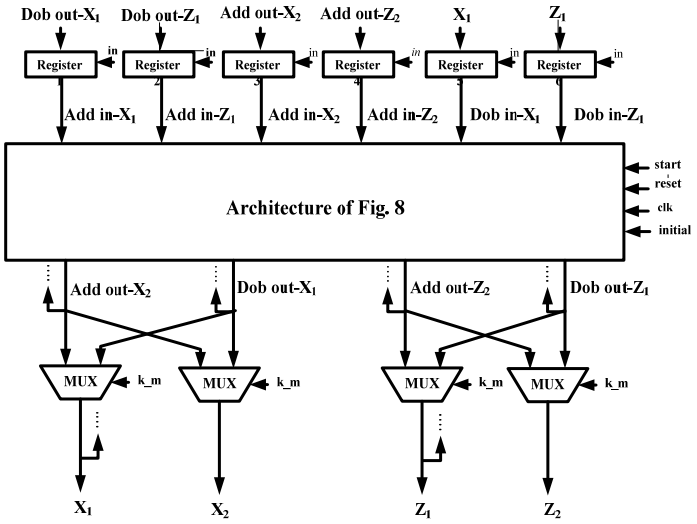

Fig. 9 Architecture of point addition and doubling iteration based on the key bits.

In addition, most of the calculations are performed in a sequential manner. A possible sequence of the instructions from standard Projective to affine coordinates is proposed in [2] in which only one inversion unit is used for converting projective coordinates to affine coordinates (see Fig. A-1 in the appendix for more detail). As it is seen form the LD algorithm, by calculating $(x_P Z_1 Z_2)^{-1}$, another inversion, X_1/Z_1, can be calculated using $(x_P Z_2 X_1)*(x_P Z_1 Z_2)^{-1}$. In this approach, the number of field inverters is reduced with the cost of increase in the number of field multipliers. However, considering the sequence of the algorithm and due to repeated referrals to these multipliers, if we use several field multipliers the length of the critical path will increase. For implementing this algorithm, ten field multiplications should be performed. In addition, for performing twelfth to seventeenth steps, we need to wait for the calculation of $(x_P Z_1 Z_2)^{-1}$ and therefore a long computational delay will be inevitable. As it is seen from the second part of the scalar multiplier processor which involves in converting projective coordinates to affine coordinates, there are many computations that should be carried out sequentially.

As was mentioned, in order to keep the critical path on the multiplier, we need to design this part of the algorithm with combinational logic as much as possible. Another approach for the implementation is based on step 4 of the LD algorithm as it is seen in Eq. (18).

$$y_3 \leftarrow (x_P+X_1/Z_1)[(X_1+xZ_1)(X_2+x_P Z_2)+(x_P^2+y_p)(Z_1 Z_2)] (x_P Z_1 Z_2)^{-1} + y_P \quad (18)$$

There are two field inversions and five field multiplications in Eq. (18). One way to implement the above function is to use two field inverter and three parallel field multiplier units. However, this causes that these multipliers to remain unused in other stages since the results of multiplications in the next steps are dependent to the results of the previous steps. This subject will cause an unbreakable delay which prohibits further speed up. Another design that leads to more efficient implementation is to enter $(Z_1 Z_2)^{-1}$ in the square brackets of Eq. (18). This will result in Eq. (19).

$$y_3 \leftarrow (x_P +X_1/Z_1)[(X_1/Z_1+ x_P)(X_2/Z_2+ x_P)+(x_P^2+y)] (x_P)^{-1} + y_P \quad (19)$$

Therefore, first we calculate Z_1^{-1}, Z_2^{-1} and x_P^{-1} using three parallel field inverters concurrently and then implement five required multiplications of Eq. (19) by using two multipliers that are implemented in parallel in three stages. Also, for this part of calculations we also need five adder units. The final value of variable x in affine coordinate system in accordance with part 2 of the LD algorithm is $x_3=X_1/Z_1$ for which we have to calculate Z_1^{-1} by using an inverter and then multiply the result by X_1. Since $X_1*Z_1^{-1}$ is used for the next multiplication, $(X_1/Z_1+x_P)*(X_2/Z_2+x_P)$, it is necessary to save the result of $X_1*Z_1^{-1}$. However, saving this value in a register and using it in next clock cycles will increase the critical path. To avoid this, this register should be eliminated. Since in the conversion of coordinates, implementation of the multipliers have been done in a parallel combinational manner (i.e., five multiplications are performed in three stages using two multipliers), in the second stage of multiplication the result of first multiplication will be lost. However, in the third stage of multiplication one of the multipliers is unused and could be used for calculating $X_1*Z_1^{-1}$. So, the multiplication $X_1*Z_1^{-1}$ is repeated in the third stage to eliminate the need for saving data in this section of the processor. Finally, one of the important steps that must be considered in the design of scalar multiplier is to select the word length (G). Due to iterative calculations in the projective coordinate system (part 2 of the LD algorithm), fast performing of calculations is very important in the design of an efficient ECC processor. So, choosing large G values for the multipliers used in the design of the first part of the processor (i.e., the multipliers in Fig. 8 or projective calculations) will be more appropriate. The word lengths that were used in this part of the processor is G_1= 41. Since calculations of the third and fourth part of the LD algorithm are used only once at the end of the algorithm and there is no iteration as part 2 of the algorithm, there is no need to select large values for G. Instead, since there are relatively a large number of computational units in this part of the processor, a relatively small value for G should be chosen to reduce the required implementation area. The word's length used in this part of the processor is G_2=11.

6 Implementation Results

To perform the calculations, the required parameters including curve order (The number of points defined on an elliptic curve over $GF(2^m)$), the coefficients and the base point coordinates are selected based on the NIST proposal in [10].

a = 1
b = 0x20A601907B8C953CA1481EB10512F78744A3205FD
n = 0x400000000000000000000292FE77E70C12A4234C33
x_P = 0x3F0EBA16286A2D57EA0991168D4994637E8343E36
y_P = 0x0D51FBC6C71A0094FA2CDD545B11C5C0C797324F1

The ECC processor was implemented using synthesizable VHDL codes. The placement, route process, and timing analysis of the synthesized designs were accomplished using Xilinx ISE 12.1 software. Before final synthesis, we made low-level or synthesis optimization that is provided by synthesis tool and is performed by its place and route tools. Most implementation tools for FPGA synthesis provide the designer with dozens of optimization options. They are Trade-offs with speed versus area; Resource sharing for area optimization; Pipelining, retiming, and register balancing for performance optimization and etc. The main problem most designers run into is that it is not clear what these options do exactly and more

importantly how these can be used to actually optimize a design. We used Pipelining, retiming, and register balancing that led to better results compared with the other options. Performance of the proposed scalar multiplication is shown in Table 1. As it is seen, a high-speed implementation is obtained with G_1 = 41. The proposed design completes the computations in the projective coordinates in $326 * (\lceil m/G_1 \rceil) + 1304$ cycles and coordinate conversion in $15 * (\lceil m/G_2 \rceil) + 214$ cycles. The term "$\lceil m/G_1 \rceil$" indicates the number of cycles required to perform finite field multiplication in part 2 of the LD algorithm or calculations in the projective coordinate system. The term "$\lceil m/G_2 \rceil$" indicates the number of cycles required to perform finite field multiplication in parts 3 and 4 of the LD algorithm or calculations for converting projective coordinates to affine coordinates. In Table 2, a number of high speed elliptic curve processors are compared with the proposed one. It should be noted that an ideal comparison would be therefore comparing all resources on the similar FPGA device. A design using dedicated resources of the device will show less logic resources as compared to other design which implements the whole logic without using any dedicated unit of the device. It also affects the throughput statistic. It has been experimentally observed that the implementation of even the same code on different grades of the same family of devices influence the final design's throughput. That situation becomes more crucial when the same design targets two different devices by two different manufactures. In such cases, for the purpose of classifying an FPGA design, we can ignore some of those factors. It can be said, as a first approximation, that the fastest design is the one which achieves fastest speed no matter what type of device has been targeted for design implementation. However, when considering a compact design (a design optimized for hardware area), this criterion cannot be applied. The comparison of two compact designs can be only justified if it is made between similar devices. Both area and throughput factors provide a measure for comparing different designs. Additionally, in order to decide how efficient a design is, we utilize the efficiency defined as $\frac{Throughput}{Area}\left(\frac{\frac{Mbit}{s}}{slices}\right)$ as a figure of merit, where Throughput is defined as $\frac{working\ frequency \times Number\ of\ Bits}{Number\ of\ Cycles}$ and hardware area can be defined as number of four inputs LUTs as well as CLB slices. The last column in the table shows the algorithmic efficiency defined as throughput/area. It would be more accurate to use throughput/#slices, but slice counts were not reported by the authors of other designs. Therefore, we have used throughput/#LUTs. As it is seen from Table 2, the proposed design is more efficient than the other designs reported in the open literature. Please notice that the work presented in [17] consumes less than half of hardware resources compared with our implementation, however with the proposed design is three times faster than this implementation.

Conclusions

A high-performance ECC processor was implemented using FPGA technology. We used a careful parallel implementation strategy to reduce the critical path of the Itoh-Tsujii's Finite-Field Inversion and used the binary graph tree idea to reduce the critical path and the required gates for the implementation of the LSD multiplier which in turn will reduce the critical path of the field inverter and the processor. In addition, in the design of the ECC processor by using three parallel multiplier units and reducing the number of unused cycles in each stage we reduced the processor delay which is mainly related to the calculations in the projective coordinate system. Separation of point doubling path from point addition path and using appropriate initial values for the initial setup of the processor reduced the complexity of the processor. In the design of the second part of the processor which involves in converting projective coordinates to affine coordinates and with considering the subject that the field multiplier module is used many times and this part of the processor has little impact on the final delay, we used the minimum possible word length to save our hardware resources as much as possible. The results show that our design is suitable for high speed and/or compact applications and therefore, the designed architecture can be well suited to the applications that require high performance. The proposed design is able to compute $GF(2^{163})$ elliptic curve scalar multiplication operations in 11.92 μs with the maximum achievable frequency of 251 MHz on Xilinx Virtex-4 (XC4VLX200) while 19604 slices or 22% of the chip area is occupied. Although prototyped in reconfigurable logic, the architecture does not exclusively make use of reconfigurability and it is also well-suited for other implementation technologies such as ASIC implementations.

Table 1 Performance of the proposed scalar multiplier.

G_1	G_2	Freq. (MHz)	Time (µs)	No. of Cycles	Area (Slices)	Area (LUT)	Efficiency
41	11	251.054	11.92	2993	19604	36727	372

Table 2 Performance of the scalar multipliers.

Ref.	m	FPGA	Freq. (MHz)	Time (µs)	Area (slices)	Area (LUT)	Area (FF)	Efficiency
Orlando and Paar [10]	167	XCV400E	76.7	210	-	3002	1769	265
N. Gura [11]	163	XCV2000E	66.4	144	-	20068	6321	56
Jarvinen et. al. [12]	163	VinexII V8000	90.2	106	18079	-	-	44
Rodriguez et. al. [13]	163	XCV2600E	46.5	63	18314 + 24 RAMs	-	-	-
Cheung [14]	163	XC2V600-4	54	60	-	-	-	-
Sakiyama [15]	163	Virtex II pro 30	100	280	8450	-	-	-
Chelton and Benaissa [16]	163	Virtex-4 VLX200	153.9	19.55	16209	26364	-	316
Ansari and Hasan [17]	163	XC2V2000	100	46.5	3416	7559	-	532
Yong et.al. [18]	163	XC2V6000	93.3	34.11	13376	2812	-	340
Kim et. al. [19]	163	XC4VLX80	143	10	24,363	-	-	-
Lutz and Hasan [23]	163	Virtex 2000E	66	75	-	10017	1930	70
Jarvinen and Skytta [24]	163	Stratix II	-	49	-	-	-	-

Appendix

The ITIMA Algorithm

Let a be any arbitrary nonzero element in the field $GF(2^m)$. Let us consider an addition chain U of length for $m - 1$ and its associated sequence V. Then the multiplicative inverse $a^{-1} \in GF(2^m)$ of a can be found by repeatedly applying Eq. (A-1).

$$\beta_{k+j}(a) = \left[\beta_k(a)\right]^{2^j} \beta_j(a) \quad for\ any\ j, k \geq 0 \qquad \text{(A-1)}$$

Hence, given $\beta_{u_0}(a) = a^{2^1-1} = a$, for each u_i, $1 \leq i \leq t$, compute:

$$\left[\beta_{u_{i_1}}(a)\right]^{2^{u_{i_2}}} \beta_{u_2}(a) = \beta_{u_{i_1}+u_{i_2}}(a) = \beta_{u_i}(a) = a^{2^{u_i}-1} \qquad \text{(A-2)}$$

A final squaring step yields the required result since:

$$[\beta_{u_t}(a)]^2 = \left(a^{2^{m-1}-1}\right)^2 = a^{-1} \qquad \text{(A-3)}$$

Table A-1 represents $\beta_i(a)$ coefficient generation for $m = 163$.

Require: $Q_1 = (X_1, Z_1)$, $Q_2 = (X_2, Z_2)$, $P = (x_P, y_P) \in E(GF(2^m))$
Ensure: (x_3, y_3)/* affine coordinates */
1: $\lambda_1 = Z_1 * Z_2$;
2: $\lambda_2 = Z_1 * x_P$;
3: $\lambda_3 = \lambda_2 + X_1$;
4: $\lambda_4 = Z_2 * x_P$;
5: $\lambda_5 = \lambda_4 * X_1$;
6: $\lambda_6 = \lambda_4 + X_2$;
7: $\lambda_7 = \lambda_3 * \lambda_6$;
8: $\lambda_8 = x_P^2 + y_P$;
9: $\lambda_9 = \lambda_1 * \lambda_8$;
10: $\lambda_{10} = \lambda_7 + \lambda_9$;
11: $\lambda_{11} = x_P * \lambda_1$;
12: $\lambda_{12} =$ inverse (λ_{11});
13: $\lambda_{13} = \lambda_{12} * \lambda_{10}$;
14: $x_3 = \lambda_{14} = \lambda_5 * \lambda_{12}$;
15: $\lambda_{15} = \lambda_{14} + x_P$;
16: $\lambda_{16} = \lambda_{15} * \lambda_{13}$;
17: $y_3 = \lambda_{16} + y_P$;
18: **Return** (x_3, y_3)

Fig. A-1 Standard projective to affine coordinates algorithm [2].

Table A-1 $\beta_i(a)$ Coefficient Generation for m-1 =162 [2].

i	u_i	$rule$	$\left[\beta_{u_{i_1}}(a)\right]^{2^{u_{i_2}}} \cdot \beta_{u_{i_2}}(a)$	$\beta_{u_i}(a) = a^{2^{u_i}-1}$
0	1	-	-	$\beta_{u_0}(a) = a^{2^1-1}$
1	2	$2u_0$	$\left[\beta_{u_0}(a)\right]^{2^{u_0}} \cdot \beta_{u_0}(a)$	$\beta_{u_1}(a) = a^{2^2-1}$
2	4	$2u_1$	$\left[\beta_{u_1}(a)\right]^{2^{u_1}} \cdot \beta_{u_1}(a)$	$\beta_{u_2}(a) = a^{2^4-1}$
3	5	u_0+u_2	$\left[\beta_{u_2}(a)\right]^{2^{u_0}} \cdot \beta_{u_0}(a)$	$\beta_{u_3}(a) = a^{2^5-1}$
4	10	$2u_3$	$\left[\beta_{u_3}(a)\right]^{2^{u_3}} \cdot \beta_{u_3}(a)$	$\beta_{u_4}(a) = a^{2^{10}-1}$
5	20	$2u_4$	$\left[\beta_{u_4}(a)\right]^{2^{u_4}} \cdot \beta_{u_4}(a)$	$\beta_{u_5}(a) = a^{2^{20}-1}$
6	40	$2u_5$	$\left[\beta_{u_5}(a)\right]^{2^{u_5}} \cdot \beta_{u_5}(a)$	$\beta_{u_6}(a) = a^{2^{40}-1}$
7	80	$2u_6$	$\left[\beta_{u_6}(a)\right]^{2^{u_6}} \cdot \beta_{u_6}(a)$	$\beta_{u_7}(a) = a^{2^{80}-1}$
8	81	u_0+u_7	$\left[\beta_{u_7}(a)\right]^{2^{u_0}} \cdot \beta_{u_0}(a)$	$\beta_{u_8}(a) = a^{2^{81}-1}$
9	162	$2u_8$	$\left[\beta_{u_8}(a)\right]^{2^{u_8}} \cdot \beta_{u_8}(a)$	$\beta_{u_9}(a) = a^{2^{162}-1}$

References

[1] Hankerson D., Menezes A. and Vanstone, S., *Guide to elliptic curve cryptography*, Springer-Verlag, 2004.

[2] Rodriguez-Henriquez F., Saqib N. A., Diaz-Prez A. and Koc C. K., *Cryptographic Algorithms on Reconfigurable Hardware,* Springer, 2006.

[3] Wollinger T., Guajardo J. and Paar C., "Security on FPGAs: State-of-the-art and Implementations Attacks", *ACM Trans. on Embedded Computing Sys*., Vol. 3, No 3, pp. 534-574, 2004.

[4] Masoumi M., "A DPA-resistant FPGA implementation of AES cryptosystem with very low hardware overhead", *Iranian Journal of Electrical and Electronic Engineering,* Vol. 8, No. 1, 2012, pp. 16-27, 2012.

[5] Fereidunian A., Lesani H., Lucas C., Lehtonen M. and Nordman M. M., "A Systems approach to information technology (IT) infrastructure design for utility management automation systems", *Iranian Journal of Electrical and Electronic Engineering*, Vol. 2, No 3, pp. 91-104, 2006.

[6] Karimi G. R. and Mirzakuchaki S., "Behavioral modeling and simulation of semiconductor devices and circuits using VHDL-AMS", *Iranian Journal of Electrical and Electronic Engineering*, Vol. 4, No. 4, pp.165-175, 2008.

[7] Lopez J. and Dahab R., "Fast multiplication on elliptic curves over GF(2^m) without precomputation", *the Workshop on Cryptographic Hardware Embedded Syst. (CHES)*, Worcester, MA, USA, 1999.

[8] Itoh T. and Tsujii S., "A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Basis", *Information and Computing*, Vol. 78, pp. 171-177, 1988.

[9] Stallings W., *Cryptography and Ntework Security*, 4th Ed., Prentice-Hall, 2006.

[10] FIPS 186-2, Available at: http://csrc.nist.gov/publications/fips/

[11] Orlando G. and Paar C., "A high-performance reconfigurable elliptic curve processor for GF(2^m)", *Cryptographic Hardware and Embedded Systems (CHES)*, Worcester, MA, USA, 2000.

[12] Gura N., Shantz S. C., Eberle H., Gupta S., Gupta V., Finchelstein D., Goupy E. and Stebila D., "An end-to-end systems approach to elliptic curve cryptography", *Workshop Cryptographic Hardware. Embedded Syst. (CHES)*, CA, USA, 2002.

[13] Jarvinen K., Tommiska M. and Skytta J., "A scalable architecture for elliptic curve point multiplication", *ICFPT*, Brisbane, Australia, 2004.

[14] Rodriguez-Henriquez F., Saqib N. A. and Diaz-Perez A., "A fast parallel implementation of elliptic curve point multiplication over GF(2^m)," *Microprocessors Microsyst*., Vol. 28, pp. 329-339, 2004.

[15] Cheung R. C. C., Telle N. J., Luk W. and Cheung P. Y. K., "Customizable elliptic curve cryptosystems", *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*., Vol. 13, No. 9, pp. 1048-1059, Sep. 2005.

[16] Sakiyama K., Batina V, Preneel K. and Verbauwhede I., "Superscalar coprocessor for high-speed curve-based cryptography", *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Yokohama, Japan, Oct. 10-13, 2006, Lecture Notes in Computer Science, Vol. 4249, Springer, pp. 415-429, 2006.

[17] Chelton W. N. and Benaissa M., "Fast elliptic curve cryptography on FPGA", *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, Vol. 16, No. 2, pp. 198-205, 2008.

[18] Ansari B. and Hasan A., "High-Performance Architecture of Elliptic Curve Scalar

multiplication", *IEEE Trans. on Comp.,* Vol. 57, No. 11, pp. 1443-1453, 2008.

[19] Yong-Ping D., "High-performance hardware architecture of elliptic curve cryptography processor over GF(2^{163})", *J. Zhejiang Univ. Sci. A*, Vol. 10, No. 2, pp. 301-310, 2009.

[20] Kim C. H., Kwon S. and Hong C. P., "FPGA Implementation of High Performance Elliptic Curve Cryptographic Processor Over GF(2^{163})", *J. of Systems Architecture,* Vol. 54, No. 10, pp. 893-900, 2008.

[21] Deschamps J-P., *Hardware Implementation of Finite-Field Arithmetic*, McGraw Hill, 2009.

[22] Song L. and Parhi K. K., "Low-Energy Digit-Serial/Parallel Finite Field Multipliers*," J. of VLSI Signal Processing,* Vol. 19, pp. 149-166, 1998.

[23] Kummar S., Wollinger T. and Paar C., "Optimum Digit Serial GF(2^m) Multipliers for Curve Based Cryptography", *IEEE Trans. Comp.*, Nol. 55, No. 10, pp. 1306-1311, 2006.

[24] Lutz J. and Hasan A., *"High performance FPGA based elliptic curve cryptographic coprocessor"*, *ITCC*, Las Vegas, USA, Vol. 2, pp. 486-492, 2004.

[25] Jarvinen K. and Skytta J., "On parallelization of high-speed processors for elliptic curve cryptography*,*" *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 16, No. 9, pp. 1162-1175, 2008.

Massoud Masoumi received his BSc from Guilan University, Rasht, Iran in 1996 and M.Sc., Ph.D. and Postdoctoral from K. N. Toosi University of Technology, Tehran, Iran, in 1999, 2006 and 2009 respectively, all in electronics engineering and all with honor degree. His research interests include side-channel attacks to encryption systems and related countermeasures and efficient VLSI architecture design for digital signal processing systems, error-correcting codes, and cryptosystems.

Hossein Mahdizadeh received his B.Sc. and M.Sc. in communication engineering from K. N. Toosi University of Technology, Tehran, Iran. His research interests include high speed architecture design for cryptographic systems, particularly FPGA and hardware implementation of asymmetric encryption systems.