A modular design of elliptic-curve point multiplication for resource constrained devices

Wei, Wei; Zhang, Li; Chang, Chip-Hong

2014

Wei, W., Zhang, L., & Chang, C.-H. (2014). A modular design of elliptic-curve point multiplication for resource constrained devices. 2014 14th International Symposium on Integrated Circuits (ISIC), 596-599.

https://hdl.handle.net/10356/105040

https://doi.org/10.1109/ISICIR.2014.7029487

A Modular Design of Elliptic-Curve Point Multiplication for Resource Constrained Devices

Wei Wei, Li Zhang and Chip-Hong Chang
School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore

***Abstract*—Elliptic curve cryptography (ECC) is a good candidate for protecting secret data on resource constrained devices. FPGA-based implementations of its main operation, i.e., scalar point multiplication, have gained popularity for their apparent speed advantage over the software counterparts. This paper presents a simple design of point multiplication that minimizes the occupied FPGA resources while maintaining an acceptable timing performance. This is achieved by employing Montgomery ladder algorithm in the projective field, simplest arithmetic units and optimized schedule for the operations. Due to the modular design approach adopted, our design can be easily adapted to different implementation requirements.**

I. Introduction

Portable electronic devices like tablets, hand phones and smart cards are pervasive in our daily life. More and more job tasks, online purchases and private information are managed with these devices, which necessitate fast and secure means to protect the confidentiality. Traditional public-key cryptosystems such as RSA, DSA and Elgamal are not the best candidates to be reckoned with for the protection in the resource constrained devices as they require a relatively long key length and hence considerable resource to achieve adequate security. Elliptic curve cryptography (ECC) is more attractive for light-weight application because of its equivalent security strength with a smaller key size, lower computational complexity and less demand on memory [1].

Scalar point multiplication is the main underlying cryptographic operation of ECC. When used in the resource constrained devices, traditional software implementation of this computational intensive operation turns out to be slow, and in some cases hard to meet the timing constraints. As a result, point multiplication is commonly realized by dedicated hardware for efficiency. Compared with the conventional mask programmed application-specific integrated circuit (ASIC), the field programmable gate array (FPGA) has advantages of much lower non-recurring cost and faster turnaround time. Unlike the static ASIC implementation, the circuit can be easily updated in FPGA by reconfiguration, which is highly desired for a cryptosystem to remain robust continuously.

There have been a large number of proposals for efficient implementation of point multiplication on FPGA. However, the goal is typical to reduce the latency of the operation [2]. Optimization techniques such as arithmetic-block duplication [3], deep pipeline [4] and instruction-level parallelism [5] usually improve the performance at the price of a substantial increase in hardware resource consumption. Besides, these techniques usually results in a relatively complex design architecture which is not amenable to engineering change for design specification update. In this work, we present a modular design of point multiplication targeting specifically on resource constrained devices. In particular, we select Montgomery ladder algorithm [6] in the projective coordinate for realizing point multiplication to greatly simplify the intermediate operations. Only the basic arithmetic units such as adder, multiplier and squarer are used in our design for field operations. Unlike some designs which require dedicated hardware for the finite field division (or inversion) operation, which is the most time consuming and costly field operation in hardware, our design performs the operation with a sequence of finite field squaring and multiplications. On the other hand, the latency of our design is kept relatively low with an optimized scheduler of operations. With a modular design methodology, our design can be easily updated for different area-delay tradeoffs and is adaptable for different finite fields to suit different security requirements.

II. Background

A. Elliptic Curves

Elliptic curves are usually divided into two classes, namely supersingular and non-supersingular curves. The latter are shown to be stronger in security strength [6]. It is defined by a set of points satisfying (1) together with a point at infinity ∞.

$$y^2 + x \cdot y = x^3 + a \cdot x^2 + b \tag{1}$$

where $b \neq 0$.

The points on the elliptic curve E form a complete group for cryptographic operations. The scalar point multiplication $Q = k \cdot P$, where Q and P are points on the curve and k is a scalar, is the main operation of ECC. Indeed, ECC relies on the intractability of inferring k given P and Q for public key cryptography. Scalar point multiplication is computed by repeated point addition and doubling.

B. Finite Field

Elliptic curve operations over the real numbers are slow and inaccurate due to the round-off errors, whereas the operations can be performed much faster and more precise over a finite field. A finite field or Galois Field is a set of q elements denoted as $GF(q)$. ECC is usually defined over two finite fields, i.e., the prime field $GF(p)$ and the binary extension field $GF(2^m)$. The latter is preferred for hardware implementation due to its efficient modulo-2 arithmetic.

Each element in $GF(2^m)$ can be written as a polynomial, $A(x) = \sum_{i=0}^{m-1} a_i \cdot x^i$, where $a_i \in \{0,1\}$. It can be simply

978-1-4799-4833-8/14/$31.00 (c)2014 IEEE

represented by m binary bits in hardware. A binary extension field $GF(2^m)$ can be constructed using an irreducible polynomial $F(x)$ of order m and all finite field operations are performed modulo $F(x)$. In this paper, the NIST-recommended B-163 elliptic curve [7] is used with $F(x)$ given by:

$$F(x) = x^{163} + x^7 + x^6 + x^3 + 1 \tag{2}$$

C. Montgomery Ladder Algorithm

Montgomery ladder algorithm is one of the most commonly used algorithm among the algorithms surveyed for elliptic curve scalar point multiplication implementation [8]. Let $(k_{m-1}, k_{m-2}, ..., k_0)_2$ be the binary representation of a finite field scalar k over $GF(2^m)$. The algorithm is depicted in Fig. 1.

Algorithm 1

Input: $k = (k_{m-1},...,k_0)_2$ with $k_{m-1} = 1$,

$P = (x_P, y_P) \in E(GF(2^m))$;

Output: $Q = (x_Q, y_Q) = k \cdot P$;

1. $Q_1 \leftarrow P, Q_2 \leftarrow 2P$;
2. **for** $i = m-2$ **downto** 0 **do**
   - **if** $k_i = 1$ **then** $Q_1 \leftarrow Q_1 + Q_2,\ Q_2 \leftarrow 2Q_2$;
   - **else** $Q_2 \leftarrow Q_1 + Q_2,\ Q_1 \leftarrow 2Q_1$;
3. $Q = Q_1$;

Fig. 1. Montgomery ladder algorithm.

This algorithm is advantageous in that intermediate point addition and doubling operations are only performed on the x-coordinate, which greatly reduces the number of intermediate field operations required. The computation of the y-coordinate of the resultant point Q is performed after the loop.

Let the x-coordinate of the points Q_1 and Q_2 be denoted as x_1 and x_2, respectively. From (1), the following equations for the computation of the x-coordinate of the point addition Q_1+Q_2 (denoted as x_{1+2}) and the point doubling $2Q_2$ (denoted as x_{2+2}) can be derived:

$$x_{1+2} = x_P + x_2(x_1 + x_2)^{-1} + (x_2(x_1 + x_2)^{-1})^2 \tag{3}$$

where $Q_1, Q_2 \in E(GF(2^m)), Q_1, Q_2 \neq \infty, Q_1 \neq Q_2, Q_1 \neq -Q_2$.

$$x_{2+2} = x_2^2 + b(x_2^2)^{-1} \tag{4}$$

where $x_2 \neq 0$ and b is the EC constant in (1).

At the final stage, the y-coordinate of the resultant point Q is calculated by:

$$y_Q = x_P^{-1}(x_1 + x_P)[(x_1 + x_P)(x_2 + x_P) + x_P^2 + y_P] + y_P \tag{5}$$

D. Projective Coordinates

As shown in (3) and (4), both point addition and doubling in affine coordinates require finite field inversion operations, which are very costly to implement in hardware. Projective coordinates can be adopted to avoid the inversion operations during point addition and doubling. If the standard projective coordinate is used, the point (x, y) in the affine coordinate will be mapped to (X, Y, Z) in the form of $x = X/Z$ and $y = Y/Z$. With appropriate expressions of Z for the resulting points of point addition and doubling as shown in (6) and (7), (3) and (4) can be simplified to (8) and (9), respectively [9]:

$$Z_{1+2} = (X_1 Z_2 + X_2 Z_1)^2 \tag{6}$$

$$Z_{2+2} = X_2^2 Z_2^2 \tag{7}$$

$$X_{1+2} = x_P Z_{1+2} + X_1 X_2 Z_1 Z_2 \tag{8}$$

$$X_{2+2} = X_2^4 + b Z_2^4 \tag{9}$$

The Montgomery ladder algorithm in the affine coordinate, i.e., Algorithm 1 in Fig. 1, can then be converted to the Montgomery algorithm in projective coordinate in Fig. 2.

Algorithm 2

Input: $k = (k_{m-1},...,k_0)_2$ with $k_{m-1} = 1$,

$P = (x_P, y_P) \in E(GF(2^m))$;

Output: $Q = (x_Q, y_Q) = k \cdot P$;

1. $X_1 \leftarrow x_P, Z_1 \leftarrow 1, X_2 \leftarrow x_P^4 + b, Z_2 \leftarrow x_P^2$;
2. **for** $i = m-2$ **downto** 0 **do**
   - **if** $k_i = 1$ **then**
     - $(X_1, Z_1) \leftarrow M\_add(X_1, Z_1, X_2, Z_2)$;
     - $(X_2, Z_2) \leftarrow M\_double(X_2, Z_2)$;
   - **else**
     - $(X_2, Z_2) \leftarrow M\_add(X_1, Z_1, X_2, Z_2)$;
     - $(X_1, Z_1) \leftarrow M\_double(X_1, Z_1)$;
3. **return** $Q(x_Q, y_Q) \leftarrow M\_xy(X_1, Z_1, X_2, Z_2)$;

Fig. 2. Montgomery algorithm in projective coordinates.

In Algorithm 2, *M_add* and *M_double* denotes the point addition and doubling in projective coordinates as expressed by (6) and (8), and (7) and (9), respectively. *M_xy* denotes the conversion operation from the projective coordinates to the affine coordinates, where the resulting point Q can be calculated based on (5):

$$x_Q = X_1 Z_1^{-1}$$

$$y_{INT} = (X_1 + x_P Z_1)(X_2 + x_P Z_2) + (x_P^2 + y_P) Z_1 Z_2 \tag{10}$$

$$y_Q = (x_P + x_Q) y_{INT} (x_P Z_1 Z_2)^{-1} + y_P$$

III. Proposed Implementation of Elliptic Curve Point Multiplication

The top-level architecture of the proposed design is depicted in Fig. 3. It consists of a control block, three arithmetic units and some registers. The arithmetic units perform three basic types of finite field operations, i.e., multiplication, squaring, and addition. The controller coordinates the data flow between the arithmetic units and the registers. The EC constants and intermediate results generated by the arithmetic units are stored in the registers. The registered data are fed to the arithmetic units guided by the controller.

A. Arithmetic Units

To minimize the required hardware resources, the hardware intensive divider is avoided in our design. Instead, the

inversion operation is realized with a sequence of squaring and multiplications based on the inversion algorithm in [10]. For the multiplier, the LSB-first design [11] is adopted, which occupies a small area at the expense of computing the multiplication in m clock cycles. The inclusion of squarer, which performs one squaring operation per clock cycle, can drastically reduce the number of required multiplications and hence improve the computation speed with a slight area overhead. The adder consists of only m XOR gates and the addition can be performed in one clock cycle.

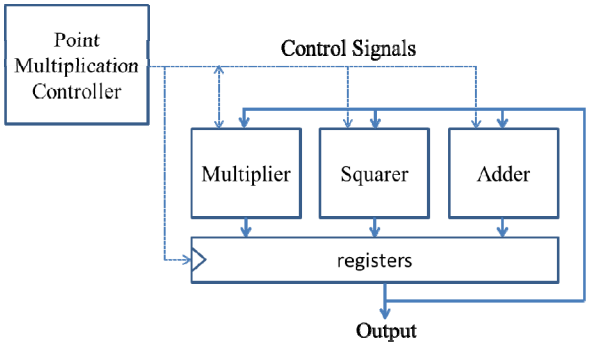

Fig. 3. Top-level architecture of the proposed design.

B. Point Multiplication Controller

The block diagram of the point multiplication controller, as shown in Fig. 4, is adapted from [12] and realized with a main state machine and three sub-controllers. The three sub-controllers are used respectively to schedule the point addition (*M_add*), the point doubling (*M_double*) and the projective-to-affine conversion (*M_xy*) in Algorithm 2. After executing step 1 of Algorithm 2, the main state machine will give control to the sub-controllers and coordinate their operations.

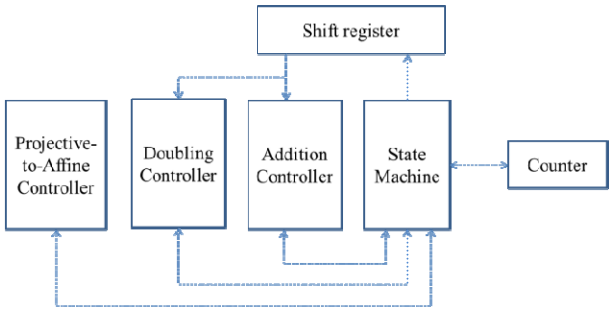

Fig. 4. Overview of the point multiplication controller.

An $(m-1)$-bit shift register is used to register the scalar multiplier k and the MSB of this shift register is k_i in Algorithm 2. The point addition and doubling controllers take k_i as an input to determine which branch of Step 2 is to be executed. A counter is employed to count up to $m-1$ times for the execution of the loop, after which the state machine will start the projective-to-affine controller to perform the final conversion.

The designs of the three sub-controllers are discussed in details as follows:

1) Addition and doubling controller

Point addition and doubling controllers each contains a state machine that implements the *M_add* and *M_double* operations in Algorithm 2, respectively. The schedules for these two operations are shown in Table I, where the five m-bit registers (i.e., X_1, X_2, Z_1, Z_2, T) are needed.

TABLE I. Point addition and doubling schedule.

Addition Schedule	Doubling Schedule
1. $X_1 \leftarrow X_1 \times Z_2$	1. $X \leftarrow X^2$
2. $Z_1 \leftarrow Z_1 \times X_2$	2. $Z \leftarrow Z^2$
3. $T \leftarrow X_1 \times Z_1$	3. $T_1 \leftarrow Z \times \sqrt{b}$
4. $Z_1 \leftarrow Z_1 + X_1$	4. $Z \leftarrow Z \times X; T \leftarrow T^2$
5. $Z_1 \leftarrow Z_1^2$	5. $X \leftarrow X^2$
6. $X_1 \leftarrow x_P \times Z_1$	6. $X \leftarrow X + T$
7. $X_1 \leftarrow X_1 + T$	

2) Projective-to-affine controller

Projective-to-affine controller consists of an inversion controller and a state machine that schedules the *M_xy* operation in Algorithm 2. As shown in Table II, the schedule for the projective-to-affine conversion is adapted from [13]. It effectively reduces the number of required inversions in (10) from two to one. It also needs five registers (i.e., X_1, X_2, Z_1, Z_2, T). It should be noted that the steps marked with '*' can be executed concurrently with the step before them.

TABLE II. Projective-to-affine schedule.

Schedule in [13]	Schedule in Our Design
1. $\lambda_1 \leftarrow Z_1 \times Z_2$	1. $T \leftarrow Z_1 \times Z_2$
2. $\lambda_2 \leftarrow Z_1 \times x_P$	2. $Z_1 \leftarrow Z_1 \times x_P$
3. $\lambda_3 \leftarrow \lambda_2 + X_1$	3. $Z_1 \leftarrow Z_1 + X_1$
4. $\lambda_4 \leftarrow Z_2 \times x_P$	4. $Z_2 \leftarrow Z_2 \times x_P$ *
5. $\lambda_5 \leftarrow \lambda_4 \times X_1$	5. $X_1 \leftarrow Z_2 \times X_1$
6. $\lambda_6 \leftarrow \lambda_4 + X_2$	6. $Z_2 \leftarrow Z_2 + X_2$
7. $\lambda_7 \leftarrow \lambda_3 \times \lambda_6$	7. $Z_2 \leftarrow Z_2 \times Z_1$
8. $\lambda_8 \leftarrow x_P^2 + y_P$	8. $X_2 \leftarrow x_P^2 + y_P$ *
9. $\lambda_9 \leftarrow \lambda_1 \times \lambda_8$	9. $X_2 \leftarrow T \times X_2$
10. $\lambda_{10} \leftarrow \lambda_7 + \lambda_9$	10. $X_2 \leftarrow Z_2 + X_2$
11. $\lambda_{11} \leftarrow x_P \times \lambda_1$	11. $Z_1 \leftarrow x_P \times T$ *
12. $\lambda_{12} \leftarrow \lambda_{11}^{-1}$	12. $Z_1 \leftarrow Z_1^{-1}$
13. $\lambda_{13} \leftarrow \lambda_{12} \times \lambda_{10}$	13. $X_2 \leftarrow Z_1 \times X_2$
14. $x_Q = \lambda_{14} \leftarrow \lambda_5 \times \lambda_{12}$	14. $x_Q = X_1 \leftarrow Z_1 \times X_1$
15. $\lambda_{15} \leftarrow \lambda_{14} + x_P$	15. $Z_1 \leftarrow X_1 + x_P$
16. $\lambda_{16} \leftarrow \lambda_{15} \times \lambda_{13}$	16. $Z_1 \leftarrow Z_1 \times X_2$
17. $y_Q \leftarrow \lambda_{16} + y_P$	17. $y_Q = Z_1 \leftarrow Z_1 + y_P$

The schedule in Table III is designed to realize the inversion algorithm in [10], which uses three registers (i.e., Z_1, Z_2 and T). The inversion controller consists of a state machine and two counters. One counter is used for counting the number of steps executed (i.e., from Step 1 to Step 11), while the other counter counts the iteration number of the *for* loop in some steps (e.g., Step 2). In total, 9 multiplications and 162 squaring operations are required to complete an inversion.

TABLE III. Algorithm and schedule for finite field inversion

Algorithm	Schedule
1. $b_1 \leftarrow A$	$Z_1, Z_2 \leftarrow A$
2. $b_2 \leftarrow b_1^{2^1} \times b_1$	**for** i = 1 to 1, $Z_1 \leftarrow Z_1^2$; $Z_1, T \leftarrow Z_1 \times Z_2$
3. $b_3 \leftarrow b_2^{2^2} \times b_2$	**for** i = 1 to 2, $Z_1 \leftarrow Z_1^2$; $Z_1, T \leftarrow Z_1 \times T$
4. $b_4 \leftarrow b_3^{2^1} \times b_1$	**for** i = 1 to 1, $Z_1 \leftarrow Z_1^2$; $Z_1, T \leftarrow Z_1 \times Z_2$
5. $b_5 \leftarrow b_4^{2^5} \times b_4$	**for** i = 1 to 5, $Z_1 \leftarrow Z_1^2$; $Z_1, T \leftarrow Z_1 \times T$
6. $b_6 \leftarrow b_5^{2^{10}} \times b_5$	**for** i = 1 to 10, $Z_1 \leftarrow Z_1^2$; $Z_1, T \leftarrow Z_1 \times T$
7. $b_7 \leftarrow b_6^{2^{20}} \times b_6$	**for** i = 1 to 20, $Z_1 \leftarrow Z_1^2$; $Z_1, T \leftarrow Z_1 \times T$
8. $b_8 \leftarrow b_7^{2^{40}} \times b_7$	**for** i = 1 to 40, $Z_1 \leftarrow Z_1^2$; $Z_1, T \leftarrow Z_1 \times T$
9. $b_9 \leftarrow b_8^{2^1} \times b_1$	**for** i = 1 to 1, $Z_1 \leftarrow Z_1^2$; $Z_1, T \leftarrow Z_1 \times Z_2$
10. $b_{10} \leftarrow b_9^{2^{81}} \times b_9$	**for** i = 1 to 81, $Z_1 \leftarrow Z_1^2$; $Z_1, T \leftarrow Z_1 \times T$
11. $A^{-1} = b_{10}^2$	$A^{-1} = Z_1 \leftarrow Z_1^2$

C. Modular Design for Reconfigurability

The proposed architecture is reconfigurable for different design considerations. In this work we target the design for resource constrained devices, but it can be easily modified to improve latency. Finite field multiplication is the most time-consuming operation in our design, which may be accelerated by replacing the LSB-first multiplier with a digit-serial multiplier. Instead of generating one-bit output per clock cycle, the digit-serial multiplier computes g bits per clock cycle and thus the latency for multiplication is reduced by g times. The area that the multiplier occupies generally increases with g. An investigation of the latency-area trade-off can be found in [11]. It is noted that no updating of the controller is needed due to the handshaking signals between the controller and multiplier.

Our design also makes update of different key lengths easy. Customizable arithmetic units can be instantiated with a parameter representing the finite field. Updating of the register length and EC constants is trivial, where all these values can be set as user-defined parameters. The same applies to the shift register and the counting value in the controller. No update is needed for the state machines for point addition, doubling and projective-to-affine conversion. The only module that needs specific attention is the inversion controller for which different algorithms are used for different finite fields.

IV. Implementation Results and Discussion

The execution time for point multiplication is calculated as follows:

$$T \approx T_{mul} \times 6 \times (m-1) + T_{mul} \times 10 + T_{inv} \qquad (11)$$

where T_{mul} and T_{inv} represent the time taken for finite field multiplication and inversion, respectively.

Neglecting the contribution from start-up, squaring and addition operations, the estimated time taken is $T_{mul} \times 6 \times (m-1)$ for point addition and doubling, and $T_{mul} \times 10 + T_{inv}$ for projective-to-affine conversion.

In the case when the LSB-first multiplier is employed, the total number of clock cycles for point multiplication in $GF(2^m)$ is approximately $6m^2$. For the finite field $GF(2^{163})$ used in this work, the number of required cycles is $6 \times 163^2 = 159414$.

The design over $GF(2^{163})$ was implemented on Spartan-3. It occupies 3383 slices (6138 LUTs and 1871FFs). It can operate at a maximum frequency of 71.4MHz, corresponding to the computation time of 2.23 ms for point multiplication.

V. Conclusion

In this work, we presented an FPGA implementation of elliptic curve scalar point multiplication targeting resource constrained devices. With the use of Montgomery algorithm in the projective coordinate, simplest arithmetic units and optimized scheduling of finite field operations, the point multiplication is realized with minimum resource utilization on FPGA with reasonably good performance. An area-efficient architecture that makes the design modular to separately optimize the arithmetic units and controllers has been implemented. The design is made easy for adaptation to different implementation requirements.

References

[1] H. Eberle *et al.*, "A public-key cryptographic processor for RSA and ECC," in *Proc. IEEE Int. Conf. on Application-Specific Syst., Architectures and Processors,* 2004, pp. 98-110.

[2] G. D. Sutter, J. P. Deschamps, and J. L. Imana, "Efficient elliptic curve point multiplication using digit-serial binary field operations," *IEEE Trans. Industrial Electronics,* vol. 60, pp. 217-225, Jan 2013.

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

[4] W. N. Chelton and M. Benaissa, "Fast Elliptic Curve Cryptography on FPGA," *IEEE Trans. VLSI Syst.,* vol. 16, no. 2, pp. 198-205, Feb. 2008.

[5] Y. Zhang *et al.*, "A high performance ECC hardware implementation with instruction-level parallelism over GF(2^{163})," *Microprocess. Microsyst.,* vol. 34, pp. 228-236, 2010.

[6] G. B. Agnew, R. C. Mullin, and S. A. Vanstone, "An implementation of elliptic curve cryptosystems over F_2155," *IEEE J. Selected Areas in Comm.,* vol. 11, pp. 804-813, Jun 1993.

[7] NIST, U.S. Dept. of Commerce, "FIPS PUB 186-3 Digital Signature Standard (DSS)," June 2009.

[8] G. M. de Dormale and J. J. Quisquater, "High-speed hardware implementations of elliptic curve cryptography: A survey," *J. of Syst. Architecture,* vol. 53, pp. 72-84, Feb-Mar 2007.

[9] D. R. Hankerson, S. A. Vanstone, and A. J. Menezes, *Guide to elliptic curve cryptography*. New York: Springer, 2003.

[10] M. A. Bencherif, H. Bessalah and A. Guessoum, "Reconfigurable elliptic curve crypto-hardware over the Galois field GF(2^{163})," *American J. of Applied Sciences,* vol. 6, pp. 1596- 1603, 2009.

[11] J.-P. Deschamps, J. L. Imaña, and G. D. Sutter, *Hardware Implementation of Finite-field Arithmetic*. New York: McGraw-Hill, 2009.

[12] M. Bednara *et al.*, "Reconfigurable implementation of elliptic curve crypto algorithms," in *Proc. 9-th Reconfigurable Architecture Workshop*, Forida, USA, Apr. 2002, pp. 157-164.

[13] N. A. Saqib, F. Rodriguez-Henriquez and A. Diaz-Perez, "A reconfigurable processor for high speed point multiplication in elliptic curves," *Int. J. Embedded Syst.,* vol. 1, pp. 237-249, 2005.