Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

High-performance hardware architecture of elliptic curve cryptography processor over $GF(2^{163})$*

Yong-ping DAN†, Xue-cheng ZOU, Zheng-lin LIU, Yu HAN, Li-hua YI

(*Department of Electronic Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*)

†E-mail: dyphhl@gmail.com

Received Jan. 13, 2008; Revision accepted May 12, 2008; Crosschecked Dec. 26, 2008

Abstract: We propose a novel high-performance hardware architecture of processor for elliptic curve scalar multiplication based on the Lopez-Dahab algorithm over $GF(2^{163})$ in polynomial basis representation. The processor can do all the operations using an efficient modular arithmetic logic unit, which includes an addition unit, a square and a carefully designed multiplication unit. In the proposed architecture, multiplication, addition, and square can be performed in parallel by the decomposition of computation. The point addition and point doubling iteration operations can be performed in six multiplications by optimization and solution of data dependency. The implementation results based on Xilinx VirtexII XC2V6000 FPGA show that the proposed design can do random elliptic curve scalar multiplication $GF(2^{163})$ in 34.11 μs, occupying 2821 registers and 13 376 LUTs.

Key words: Elliptic curve cryptography (ECC), Scalar multiplication, Hardware implementation
doi:10.1631/jzus.A0820024 **Document code:** A **CLC number:** TN402

INTRODUCTION

Recently, Elliptic curve cryptography (ECC) has drawn more and more attention due to the fact that its selected key length can be smaller than that in RSA cryptosystems for the same level of security. Literature has shown that ECC is the most widely used public-key cryptosystem, which allows much useful functionality such as digital signature, public-key encryption, and key agreements. For those needs, ECC is indeed an attractive solution as the public-key scheme. However, the computation involved in the scalar multiplication for an elliptic curve is time consuming and more complex than that in RSA.

In many applications a software implementation of ECC might be appropriate, but in some cases better performances are required and consequently hardware implementations should be used instead. As the popularity of ECC increases, the need for efficient hardware solutions that accelerate the computation of elliptic curve point multiplications also increases. The recent research on implementation of ECC has shown that highly efficient implementation of arithmetic operations in a finite field of characteristic two can enormously speed up ECC computation, and thus ECC can be favorable in speed compared to RSA for the same level of security. In particular, the curves based on fields of type $GF(2^m)$ allow efficient implementation in terms of silicon area and execution time.

Recently, several high-speed hardware implementations of ECC over $GF(2^m)$ have been developed (Rodríguez-Henríquez *et al*., 2004; Cheung *et al*., 2005; Shu *et al*., 2005; Sozzani *et al*., 2005; Sakiyama *et al*., 2006; Sakiyama *et al*., 2007; Al-Khaleel *et al*., 2007). A good review can be found in (Meurice de Dormale and Quisquater, 2007). All the previous implementations present good performance. The implementations in (Sozzani *et al*., 2005; Sakiyama *et al*., 2007) were carried out in ASIC (application-specific integrated circuit). The implementations in (Rodríguez-Henríquez *et al*., 2004; Shu *et al*., 2005)

* Project supported by the Hi-Tech Research and Development Program (863) of China (No. 2006AA01Z226), and the Research Foundation of Huazhong University of Science and Technology, China (No. 2006Z001B)

are faster compared to the others implemented in FPGA (field-programmable gate array). However, the implementation in (Rodríguez-Henríquez *et al*., 2004) is not the one of the conversion of projective coordinates to affine coordinates. Shu *et al*.(2005) implemented the Koblitz curves instead of general curves; however, it is not suitable for a random elliptic curve.

This paper focuses on the high-speed hardware implementation of ECC over $GF(2^{163})$ in FPGA. We optimize the algorithm in parallel and design a novel architecture, which can do all the operations including the coordinates conversion over $GF(2^{163})$ random elliptic curve scalar multiplications. Our design was implemented on a Xilinx VirtexII XC2V6000 FPGA device, and the performance comparisons are demonstrated.

The rest of this paper is organized as follows. In Section 2, we present a brief introduction of the mathematical background and algorithm of ECC. In Section 3, we detail the algorithm optimization decomposition in parallel and resource occupation for implementation in hardware. In Section 4, we discuss the proposed architecture of ECC, the modular arithmetic logic unit and the finite field arithmetic unit. In Section 5, the implementation results and performance obtained are compared with those in other published works. Finally, in Section 6, some conclusions are drawn.

MATHEMATICAL BACKGROUND AND LOPEZ-DAHAB ALGORITHM

Elliptic curve cryptosystems can be implemented on $GF(p)$ and $GF(2^m)$. Usually $GF(2^m)$ leads to a smaller and faster processor. In this work, we select the finite field $GF(2^m)$ in polynomial basis representation. For $GF(2^m)$, an elliptic curve is defined as a set of points satisfying

$$y^2 + xy \equiv x^3 + ax^2 + b \pmod{f(x)}, \qquad (1)$$

where x and y are elements of the field $GF(2^m)$, a and b are the curve parameters ($b\neq 0$), and $f(x)$ is an irreducible polynomial. The points on the curve form an additive group when they are combined with the "point at infinity". By definition, the addition of two elements in a group results in another element of the group. Taken two points, P and Q, of the curve, it is possible to define a point addition operation, $P+Q$. The particular case $P+P=2P$ is called point doubling. As a result, any point on the curve, say P, can be added to itself by an arbitrary number of times, and the result will also be a point on the curve. The fundamental and most expensive operation underlying ECC is the point multiplication kP, where k is an integer and P is an elliptic point. A single point (scalar) multiplication requires multiple computations of point additions ($P\neq Q$) and point doubling ($P=Q$). Calculating kP, where P is a point on the curve, will yield a new point on the curve. This procedure forms the basics for public-key cryptography using ECC. Point multiplication is defined by repeated addition:

$$kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}.$$

In order to reduce the computation time of the point multiplication, there are many types of coordinates in which an elliptic curve may be represented. The so-called projective coordinates have some implementation advantages compared to the affine coordinates. The main conclusion is that point addition and point doubling can be done in projective coordinates without inversion. More precisely, only one inversion needs to be performed at the end of a point multiplication operation.

Several scalar multiplication algorithms have been proposed in the literature, such as the double-addition approach, the addition-subtraction algorithm, the fast simultaneous scalar multiplication approach (Akishita, 2001), and the complementary recoding algorithm (Balasubramaniam and Karthikeyan, 2007). In this work, we use the Lopez-Dahab algorithm (Lopez and Dahab, 1999), which is based on the concept of "montgomery", to calculate kP. The Lopez-Dahab algorithm calculates one point addition and one point doubling in each step, which is faster than other algorithms. Moreover, the algorithm requires fewer register resources compared to other algorithms in hardware solutions. In this algorithm, point addition and doubling can be performed in parallel; coordinates transformation needs to be performed at the last step. Another advantage of this algorithm is that it is resistant to timing attacks and simple power attacks. The Lopez-Dahab algorithm is shown in Algorithm 1 (Lopez and Dahab, 1999).

Algorithm 1 Lopez-Dahab algorithm

Input: $k=(k_{n-1}, k_{n-2}, \ldots, k_1, k_0)_2$ with $k_{n-1}=1$, $p(x, y)\in E(GF(2^m))$.

Output: $Q=kP$.

1. set $x_1=x$, $z_1=1$, $x_2=x^4+b$, $z_2=x^2$;
2. for $i=n-2$ to 0 do
3. if $k_i=1$ then
4. $(x_1, z_1)\leftarrow Madd(x_1, z_1, x_2, z_2, x)$;
   $(x_2, z_2)\leftarrow Mdouble(x_2, z_2, b)$;
5. else
6. $(x_2, z_2)\leftarrow Madd(x_2, z_2, x_1, z_1, x)$;
   $(x_1, z_1)\leftarrow Mdouble(x_1, z_1, b)$;
7. endif
8. endfor
9. $Q\leftarrow Mxy(x_1, z_1, x_2, z_2, x, y)$;
10. return Q;

In Algorithm 1, *Madd*() function is the point addition operation on the elliptic curve, *Mdouble*() is the point doubling computation, and *Mxy*() is the conversion of projective coordinates to affine coordinates. The reader is referred to (Lopez and Dahab, 1999) for detailed explanation. Functions *Madd*(), *Mdouble*() and *Mxy*() in Algorithm 1 are defined as follows:

Madd(x_1, z_1, x_2, z_2, x)
{
$X\leftarrow x_1z_2x_2z_1+x(x_1z_2+x_2z_1)^2$;
$Z\leftarrow(x_1z_2+x_2z_1)^2$;
return (X, Z);
}

Mdouble(x_1, z_1, b)
{
$X\leftarrow x_1^4+bz_1^4$;
$Z\leftarrow x_1^2z_1^2$;
return (X, Z);
}

Mxy$(x_1, z_1, x_2, z_2, x, y)$
{
$x_k=x_1/z_1$;
$y_k=[x^2+y+(x+x_1/z_1)(x+x_2/z_2)](x+x_k)/x+y$;
return (x_k, y_k);
}

In these functions, (x, y) is the coordinate of the original point P, which is fixed during the calculation of kP; (x_k, y_k) is the coordinate of kP. The three basic functions in turn rely on finite field operations such as addition, multiplication, and inversion.

ALGORITHM DECOMPOSITION AND OPTIMIZATION IN HARDWARE

As can be seen from Algorithm 1, the whole process of the scalar multiplication operation was divided into three parts: the initialization operation, the middle iterations of point addition and point doubling computing, and the conversion of projective coordinates to affine coordinates operation. In order to implement the algorithm, we propose an efficient finite field modular arithmetic logic unit (MALU) to perform all the operations of scalar multiplication. The finite field MALU includes three parts: a finite field multiplication module, a finite field addition module, and a finite field square module. In the rest of this section, we discuss the scalar multiplication decomposition scheme, which is highly optimized toward the execution of kP through the finite field MALU.

Initialization decomposition and resource analysis

According to the Lopez-Dahab algorithm, the initialization calculation formulae are given as

$$x_1 = x,\ \ x_2 = x^4 + b,\ \ z_1 = 1,\ \ z_2 = x^2. \qquad (2)$$

The initialization calculation can be decomposed as

$$z_2\leftarrow x^2,\ \ x_2\leftarrow z_2^2,\ \ x_2\leftarrow x_2+b,$$

where there are two squares and one addition operation. The x, x_1, z_1, x_2, z_2 and b are the indispensability variables in Algorithm 1. Therefore the initialization calculation needs no additional intermediate variable resource.

Point addition and point doubling decomposition and optimization

1. Decomposition in parallel

The operations of point addition and point doubling are the core of the elliptic curve encryption scalar multiplication, and they consume the most of the computation time. In order to speed up the scalar multiplication, the first idea is to reduce the point addition and point doubling computation time. The point addition is carried out as follows:

$$\begin{cases} X = x_1z_2x_2z_1 + x(x_1z_2 + x_2z_1)^2, \\ Z = (x_1z_2 + x_2z_1)^2. \end{cases} \qquad (3)$$

The input parameters of point addition do not depend on the current bit of the key, but the output is stored to different variables with the different bits of the key. According to Algorithm 1, in point doubling both input and output parameters depend on the current bit of the key, and there are different inputs and outputs for different bits of the key. Eq.(4) is the point doubling calculation equation when the current bit of the key is "1":

$$X = x_2^4 + bz_2^4, \quad Z = x_2^4 z_2^2. \tag{4}$$

As can be seen from Eqs.(3) and (4), the point addition and point doubling operations consist of six multiplications, three additions, and five squares.

Addition can be performed in one clock cycle; multiplication and square operations need many clock cycles. However, in the field $GF(2^m)$, when the irreducible polynomial defining the field is known in advance, the complexity of square is significantly lower than that of multiplication and generally becomes comparable to that of addition. In practice, square can be performed in one clock cycle, which will be discussed in detail in Section 4. Therefore, finite field multiplication becomes the bottleneck of the elliptic curve scalar multiplication. The key for speeding up the ECC computation is how to implement the multiplication in a fast and efficient way. Bit parallel multipliers complete a multiplication in a single iteration. All the m bits of both input operands are considered at the same time, and the result is immediately generated. Unfortunately, the parallel multiplication occupies too many resources and the critical path delay is too long. The disadvantage of the serial multiplication is the small number of iterations required for the loop. The m iterations translate to a minimum of m clock cycles, which is not suitable for high-performance implementation in hardware. A compromise among these architectures is the most significant digit (MSD) serial multiplication. In this study, we focus on the ECC implementation over $GF(2^{163})$. In order to minimize the multiplication computation time and shorten the critical path delay, we propose a grouping MSD multiplication over $GF(2^{163})$, which can be performed in two clock cycles and is detailed in Section 4.

As multiplication needs two clock cycles, and addition and square can be completed in one clock cycle, we consider addition, square, and multiplication to be computed in parallel, which will further reduce the computing time of the point addition and point doubling. Based on this idea, the point addition and point doubling operations can be decomposed as in Algorithm 2. The K_i=1 and K_i=0 represent that the current bit of the private key is "1" and "0", respectively.

Algorithm 2 Point addition and point doubling operation decomposition

	Operation order	Parallel computation
K_i=1	1. $T_1 \leftarrow x_2 z_1$	$x_2 \leftarrow x_2^2$
	2. $x_1 \leftarrow x_1 z_2$	$z_2 \leftarrow z_2^2$, $T_2 \leftarrow z_2^2$
	3. $z_2 \leftarrow x_2 z_2$	$z_1 \leftarrow x_1 + T_1$, $x_1 \leftarrow z_1^2$
	4. $T_2 \leftarrow bT_2$	
	5. $x_1 \leftarrow x_1 T_1$	$x_2 \leftarrow x_2^2$, $x_2 \leftarrow x_2 + T_2$
	6. $T_1 \leftarrow xz_1$	
	7. $x_1 \leftarrow x_1 + T_1$	
K_i=0	1. $T_1 \leftarrow x_2 z_1$	$z_1 \leftarrow z_1^2$
	2. $x_2 \leftarrow x_1 z_2$	$x_1 \leftarrow x_1^2$, $T_2 \leftarrow z_1^2$
	3. $z_1 \leftarrow x_1 z_1$	$z_2 \leftarrow x_2 + T_1$, $z_2 \leftarrow z_2^2$
	4. $T_2 \leftarrow bT_2$	
	5. $x_2 \leftarrow x_2 T_1$	$x_1 \leftarrow x_1^2$, $x_1 \leftarrow x_1 + T_2$
	6. $T_1 \leftarrow xz_2$	
	7. $x_2 \leftarrow x_2 + T_1$	

As shown in Algorithm 2, the operations of point addition and point doubling can be completed in turn with six multiplications and one addition when the corresponding addition and square are computed in parallel. Therefore the computation time of a single point addition and point doubling operation is approximately equal to that of six multiplications and one addition, which is less than that of six multiplications, three additions and five squares. The resources required are only two intermediate variables T_1 and T_2 besides x, b, x_1, z_1, x_2, z_2.

Optimization and solving dada dependency at iteration transitions

As the multiplication computation time is two clock cycles, two extra clock cycles are spent on loading inputs and unloading outputs. This leads to a total of four clock cycles for one multiplication in practice. Therefore the loading and unloading of multiplier consume too many clock cycles. In order to

reduce the operation time, the loading input of the next multiplication can be considered at the same time as the previous unloading outcome, which would reduce the number of iterations of point addition and point doubling by five clock cycles. Because of data dependency, the sixth multiplication needs four clock cycles and the last addition needs one clock cycle. The input of the first multiplication for the next iteration can be loaded after the completion of the last addition of the previous iteration, which cannot be computed with no idle cycle in the entire point addition and point doubling loop for scalar multiplication. The entire operation of a single point addition and point doubling takes 20 clock cycles.

The previous multiplication stores the result into the register. At the same time, the next multiplication input can be loaded, which shortens multiplication by one clock cycle. We consider that the six multiplications should form a loop without an idle clock cycle if the data dependency problem can be resolved for the whole process of point addition and point doubling iterations, and then the computation time can be reduced from 20 to 18 clock cycles. The point addition and point doubling computation time can be shortened by 324 clock cycles for the effective 163 bits of the key, which is considerable for the high-speed elliptic curve scalar multiplication in hardware.

In Algorithm 2, we observe that z_1, z_2 and x_2 are ready before the last addition when K_i=1. The unloading of the sixth operation and the loading of the first operation for the next iteration can be performed in the same clock cycle. Then the seventh operation of the previous iteration and the first operation of the next iteration can be performed in parallel, which does not affect the input of the other operations. Not only can the same treatment be done, but also operation steps 1 and 2 in Algorithm 2 need to swap positions when K_i=0. Based on this idea, the improved decomposition algorithm of the point addition and point doubling is shown in Algorithm 3.

Because only the order of operation is adjusted, Algorithm 3 and Algorithm 2 occupy the same resource. As can be seen from Algorithm 3, the iteration transition has no problem when the current and next bits of the key are the same. However, there is a mistake when the current and next bits of the key are different; in such a case, we must analyze and improve the iteration transition operation. Let the iteration transition jump from K_i=1 to K_i=0. Since z_1, x_2 and z_2 have been completed, a few steps at the beginning of K_i=0 need to be dealt with in the same way as K_i=1. The corresponding adjustment also needs to be done when the transition jumps from K_i=0 to K_i=1. The solution of iteration transition operation is shown in Algorithm 4 when the current and the next bits of the key are different.

Algorithm 3 Improved decomposition of point addition and point doubling

	Operation order	Parallel computation
K_i=1	1. $T_1 \leftarrow x_2 z_1$	$x_1 \leftarrow x_1 + T_1$
	2. $x_1 \leftarrow x_1 z_2$	$z_2 \leftarrow z_2^2$, $T_2 \leftarrow z_2^2$
	3. $T_2 \leftarrow T_2 b$	$x_2 \leftarrow x_2^2$
	4. $z_2 \leftarrow x_2 z_2$	$z_1 \leftarrow x_1 + T_1$, $z_1 \leftarrow z_1^2$
	5. $x_1 \leftarrow x_1 T_1$	$x_2 \leftarrow x_2^2$, $x_2 \leftarrow x_2 + T_2$
	6. $T_1 \leftarrow x z_1$	
K_i=0	1. $T_1 \leftarrow x_1 z_2$	$x_2 \leftarrow x_2 + T_1$
	2. $x_2 \leftarrow x_2 z_1$	$z_1 \leftarrow z_1^2$, $T_2 \leftarrow z_1^2$
	3. $T_2 \leftarrow T_2 b$	$x_1 \leftarrow x_1^2$
	4. $z_1 \leftarrow x_1 z_1$	$z_2 \leftarrow x_2 + T_1$, $z_2 \leftarrow z_2^2$
	5. $x_2 \leftarrow x_2 T_1$	$x_1 \leftarrow x_1^2$, $x_1 \leftarrow x_1 + T_2$
	6. $T_1 \leftarrow x z_2$	

Algorithm 4 Solution of iteration transition for point addition and point doubling

	Operation order	Parallel computation
From K_i=0 to K_i=1	1. $T_1 \leftarrow x_1 z_2$	$x_2 \leftarrow x_2 + T_1$
	2. $x_1 \leftarrow x_2 z_1$	$z_2 \leftarrow z_2^2$, $T_2 \leftarrow z_2^2$
	3. $T_2 \leftarrow T_2 b$	$x_2 \leftarrow x_2^2$
	4. $z_2 \leftarrow x_2 z_2$	$z_1 \leftarrow x_1 + T_1$, $z_1 \leftarrow z_1^2$
	5. $x_1 \leftarrow x_1 T_1$	$x_2 \leftarrow x_2^2$, $x_2 \leftarrow x_2 + T_2$
	6. $T_1 \leftarrow x z_1$	
From K_i=1 to K_i=0	1. $T_1 \leftarrow x_2 z_1$	$x_1 \leftarrow x_1 + T_1$
	2. $x_2 \leftarrow x_1 z_2$	$z_1 \leftarrow z_1^2$, $T_2 \leftarrow z_1^2$
	3. $T_2 \leftarrow T_2 b$	$x_1 \leftarrow x_1^2$
	4. $z_1 \leftarrow x_1 z_1$	$z_2 \leftarrow x_2 + T_1$, $z_2 \leftarrow z_2^2$
	5. $x_2 \leftarrow x_2 T_1$	$x_1 \leftarrow x_1^2$, $x_1 \leftarrow x_1 + T_2$
	6. $T_1 \leftarrow x z_2$	

As can be seen from the comparison between Algorithms 3 and 4, only the order of the two multiplications and the addition in parallel are different at the beginning of the iteration operation, as shown by shadow. The other operations are exactly the same.

Therefore the problem of iteration transition can be solved by the additional specialized operations. The normal iteration transition will be implemented based upon Algorithm 3 when the current and the next bits of the key are the same. The different transitions will be dealt with by the additional specialized operation based upon Algorithm 4 at the beginning of the iteration operation. This solution does not add to the number of clock cycles.

As discussed above, the six multiplications of point addition and point doubling consume 18 clock cycles besides the first iteration. One extra clock cycle needed by the first iteration is spent on loading the inputs of the first multiplication. Therefore, the total computations of point addition and point doubling need 2917 clock cycles for the effective 163 bits of the key through optimization decomposition in parallel and the solution of data dependency.

Coordinates conversion decomposition and resource analysis

From the formulae for point operations given in Algorithm 1, the inversion is avoided by use of projective coordinates. While projective coordinates are used, a conversion to affine coordinates is required at the end of the whole computation. In order to reduce inversion operations, the coordinates conversion computation equation can be modified as follows:

$$\begin{cases} x_k = z_2 x_1 x/(z_1 z_2 x), \\ y_k = (x + x_k)[(x_1 + xz_1)(x_2 + xz_2) \\ \quad + (x^2 + y)z_1 z_2](z_1 z_2 x)^{-1} + y. \end{cases} \tag{5}$$

There is a finite field inversion operation in Eq.(5). The implementation of finite field inversion in hardware is the most difficult and expensive. In order to minimize the resource, the specifically multiplicative inversion is not designed. We adopted (Itoh and Tsujii, 1988)'s method to compute the inversion by decomposition of the finite field MALU, which needs 9 multiplications and 162 squares, and thus requires no specific hardware resource. Coordinates conversion and inversion operation can be decomposed in Algorithm 5.

Algorithm 5 Coordinates conversion and inversion operation decomposition

Conversion decomposition	Inversion decomposition
1. $T_1 \leftarrow z_1 z_2$	1. $z_1 \leftarrow T_2^{\,2}$
2. $z_1 \leftarrow xz_1$	2. $x_2 \leftarrow z_1 T_2$
3. $z_1 \leftarrow x_1 + z_1$	3. $z_1 \leftarrow x_2^{\,2^2}$
4. $z_2 \leftarrow xz_2$	4. $x_2 \leftarrow x_2 z_1$
5. $x_1 \leftarrow x_1 z_2$, $z_2 \leftarrow x_2 + z_2$	5. $z_1 \leftarrow x_2^{\,2}$
6. $z_2 \leftarrow z_1 z_2$, $x_2 \leftarrow x^2$, $x_2 \leftarrow x_2 + y$	6. $x_2 \leftarrow z_1 T_2$
7. $x_2 \leftarrow x_2 T_1$	7. $z_1 \leftarrow x_2^{\,2^5}$
8. $T_2 \leftarrow xT_1$, $z_2 \leftarrow x_2 + z_2$	8. $x_2 \leftarrow x_2 z_1$
9. $z_1 \leftarrow \mathrm{inv}(T_2)$	9. $z_1 \leftarrow x_2^{\,2^{10}}$
10. $T_1 \leftarrow x_1 z_1$	10. $x_2 \leftarrow x_2 z_1$
11. $x_2 \leftarrow x + T_1$	11. $z_1 \leftarrow x_2^{\,2^{20}}$
12. $z_1 \leftarrow x_2 z_1$	12. $x_2 \leftarrow x_2 z_1$
13. $z_1 \leftarrow z_1 z_2$	13. $z_1 \leftarrow x_2^{\,2^{40}}$
14. $z_1 \leftarrow z_1 + y$	14. $x_2 \leftarrow x_2 z_1$
	15. $z_1 \leftarrow x_2^{\,2}$
	16. $x_2 \leftarrow z_1 T_2$
	17. $z_1 \leftarrow x_2^{\,2^{81}}$
	18. $x_2 \leftarrow x_2 z_1$
	19. $z_1 \leftarrow x_2^{\,2}$

Inversion operation requires a total of three registers through computing decomposition. Six registers are enough for the whole operation of coordinates transformation and inversion, and no additional intermediate variables are needed. Because of the data dependency, only some operations of coordinates transformation can be performed in parallel, and most of them are computed in serial. Because the coordinates transformation operates only once at the final step, it has little impact on the timing of scalar multiplication.

PROPOSED PROCESSOR AND IMPLEMENTATION

Architecture of the processor

The proposed architecture of the scalar multiplication processor is shown in Fig.1.

The processor consists of an MALU module, a control module, a register file module, and two multiplexers. The control module consists of a finite state machine and three counters. Some control signals are created for the initialization operation, the point addition and point doubling, and the final coordinates

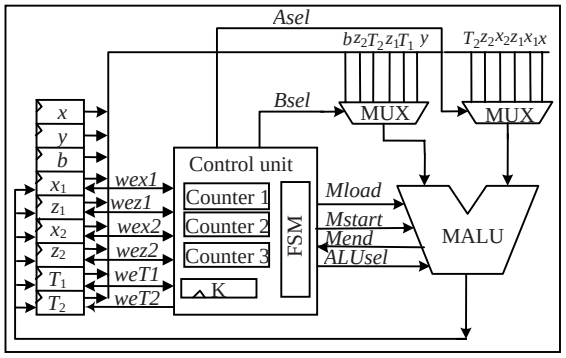

Fig.1 Architecture of the scalar multiplication processor

transformation operation based on the private key, according to the Lopez-Dahab algorithm. The register file module includes the input parameters x, y, b, k and intermediate results such as x_1, z_1, x_2, z_2, T_1, T_2, which are a total of ten 163-bit registers. The interface is relatively simple, which is not shown in Fig.1. Two multiplexers are used for the processor, which can reduce the time of loading data from the register files. The input of multiplexers can be arranged reasonably to reduce the resources through the optimization of algorithm decomposition.

Modular arithmetic logic unit

MALU is an important module for the elliptic curve encryption processor, which can perform finite field addition, multiplication, square, and inversion (converted into multiplication and square) operations. The architecture of MALU is shown in Fig.2. It consists of six parts: two 163-bit registers, one multiplication, one addition, one square, and one multiplexer module. In Fig.2, *Mload* and *Mstart* are the load and the start signals of multiplication, respectively; *Mend* is the completion signal of the multiplication; *ALUin1* and *ALUin2* are the two inputs of the MALU; *ALUsel* is the selection signal of MALU output, which can be used to select the output of the addition, multiplication and square. *ALUout* is the output signal of MALU.

The initialization, the point addition and point doubling, and the coordinates conversion computations consist of $GF(2^m)$ field arithmetic, so the efficient implementation of $GF(2^m)$ field arithmetic is very important to ECC in hardware. The $GF(2^m)$ field arithmetic includes addition, multiplication, and square. Addition in a binary Galois field is trivial, since it is a bit-by-bit addition, having no carry bit, which can be implemented by simply using XOR gates. The implementations of multiplication and square are complicated in hardware. In the following we will discuss how to implement the multiplication and square arithmetics efficiently.

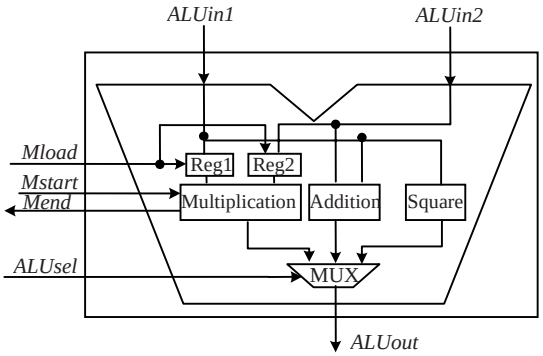

Fig.2 Architecture of MALU

Finite field multiplication

Finite field multiplication is the bottleneck of scalar multiplication, especially when using projective coordinates. Finite field multiplication must be implemented with high efficiency for a high performance design. In Section 3, we have discussed that the MSD serial modular multiplication is a good choice for our design.

Multiplication of two elements $A(x)$, $B(x)\in GF(2^m)$, with $A(x)=a_{m-1}x^{m-1}+\ldots+a_1x+a_0$ and $B(x)=b_{m-1}x^{m-1}+\ldots+b_1x+b_0$, satisfies

$$\begin{aligned} C(x) &= A(x)\times B(x) \mod f(x) \\ &= \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} a_i b_j x^{i+j} \mod f(x), \end{aligned} \tag{6}$$

where $f(x)$ is an irreducible polynomial. By expanding $B(x)$ and distributing $A(x)$ through, we obtain

$$C(x) = b_{m-1}x^{m-1}A(x) + \ldots + b_1xA(x) + b_0A(x) \mod f(x). \tag{7}$$

We select m=163 for $GF(2^m)$ and perform the operation in two clock cycles by using MSD multiplication, in which the size of D is 81 and the critical path delay of the MSD multiplication is 10 XOR gates. A too long delay makes the system clock frequency

very low. In order to speed up the computation, the finite field multiplication must be optimized. We propose to implement the multiplication by grouping in parallel, which needs more registers. It means that the 163 bits are divided into five groups, each group implemented using MSD multiplication, and the five MSD multiplications can be computed in parallel. Based on this idea, Eq.(7) can be transformed to

$$\begin{aligned}C(x)=&(x^{132}(0+0+A(x)b_{162}x^{30}+\dots\\&+A(x)b_{133}x+A(x)b_{132})\bmod f(x))\bmod f(x)\\&+(x^{99}(A(x)b_{131}x^{32}+A(x)b_{130}x^{31}+\dots\\&+A(x)b_{100}x+A(x)b_{99})\bmod f(x))\bmod f(x)\\&+(x^{66}(A(x)b_{98}x^{32}+A(x)b_{97}x^{31}+\dots\\&+A(x)b_{67}x+A(x)b_{66})\bmod f(x))\bmod f(x)\\&+(x^{33}(A(x)b_{65}x^{32}+A(x)b_{64}x^{31}+\dots\\&+A(x)b_{34}x+A(x)b_{33})\bmod f(x))\bmod f(x)\\&+(A(x)b_{32}x^{32}+A(x)b_{31}x^{31}+\dots\\&+A(x)b_1x+A(x)b_0)\bmod f(x).\end{aligned}\tag{8}$$

In Eq.(8), each group has 33 bits. The highest bit of each group is used to judge the initial value of production that is either 0 or A. The remaining 32 bits can be performed in two clock cycles when the size of D is 16. The critical path delay is only seven XOR gates. From Eq.(8), the multiplier requires one sum module to add the results of the partition product. In order to reduce the number of clock cycles of the multiplier, we consider that the sum model can be performed in the clock cycle of unloading of the multiplier, which can reduce one clock cycle for the multiplier and does not affect the critical path delay. Therefore the proposed multiplier can be performed in two clock cycles besides the two extra clock cycles that are spent on loading inputs and unloading outputs.

The proposed architecture of the multiplier can be adopted in generic size field $GF(2^m)$ to speed up the computation. However, this method requires more resource than the general MSD multiplication. The grouping number and the MSD size are considered by using different sizes to balance the area and performance when the field size is large. In this study we take the field size of 1027 as an example. The 1027 bits are divided into four groups, which are computed in parallel. Based on this idea, Eq.(7) can be transformed to

$$\begin{aligned}C(x)=&(x^{771}(0+A(x)b_{1026}x^{255}+A(x)b_{1023}x^{254}+\dots\\&+A(x)b_{772}x+A(x)b_{771})\bmod f(x))\bmod f(x)\\&+(x^{514}(A(x)b_{880}x^{256}+A(x)b_{879}x^{255}+\dots\\&+A(x)b_{515}x+A(x)b_{514})\bmod f(x))\bmod f(x)\\&+(x^{257}(A(x)b_{513}x^{256}+A(x)b_{512}x^{255}+\dots\\&+A(x)b_{258}x+A(x)b_{257})\bmod f(x))\bmod f(x)\\&+(A(x)b_{256}x^{256}+A(x)b_{255}x^{255}+\dots\\&+A(x)b_1x+A(x)b_0)\bmod f(x).\end{aligned}\tag{9}$$

In Eq.(9), each group has 257 bits, which can be performed in 16 clock cycles when the size of D is 16. The critical path delay is affected by the MSD size. The computation time is decided by the number of groups.

Finite field square

The finite field square is a specific case of general multiplication and can be performed by the multiplication, but it takes too much time. Performance can be improved significantly by optimizing the architecture, specifically for the case of square. In (Wu, 2002), a parallel implementation of the square was presented. The square is computed as follows:

$$\begin{aligned}C=A^2\bmod f(x)=&(a_{m-1}x^{2(m-1)}+a_{m-2}x^{2(m-2)}+\cdots\\&+a_1x^2+a_0)\bmod f(x).\end{aligned}\tag{10}$$

The finite field square can be implemented by expanding A to double its bit-length by interleaving 0 bit in-between the original bits of A and then reducing the double length result. Eq.(10) can be changed to

$$C=A^2\bmod f(x)=x^mA_h(x)\bmod f(x)+A_l(x),\tag{11}$$

where

$$\begin{aligned}A_h(x)=&0+a_{m-1}x^{m-2}+0+\cdots\\&+a_{(m+3)/2}x^3+0+a_{(m+1)/2}x+0,\\A_l(x)=&a_{(m-1)/2}x^{m-1}+0+a_{(m-3)/2}x^{m-3}+0+\cdots\\&+a_1x^2+0+a_0.\end{aligned}$$

The irreducible polynomial $f(x)$ has a small number of non-zero coefficients, which is the trinomial polynomials $f(x)=x^m+x^l+1$ or pentanomial polynomials $f(x)=x^m+x^i+x^j+x^k+1$. Then the high part of Eq.(11) can be computed using Eq.(12) or Eq.(13) with different irreducible polynomials:

$$x^m A_h(x) \bmod f(x) = (x^l + 1)(0 + a_{m-1}x^{m-2} + 0 + \cdots + a_{(m+1)/2}x + 0), \quad (12)$$

$$x^m A_h(x) \bmod f(x) = (x^i + x^k + x^l + 1)(0 + a_{m-1}x^{m-2} + 0 + \cdots + a_{(m+1)/2}x + 0). \quad (13)$$

According to Eqs.(12) and (13), the shift operation is relatively simple, and then the square can be changed to the finite field addition.

We select m=163 for $GF(2^m)$, and the irreducible polynomial is $f(x)=x^{163}+x^7+x^6+x^3+1$, recommended by NIST (National Institute of Standards and Technology, USA). Therefore the implementation of square can be optimized by the fixed irreducible polynomial. Eq.(10) can be changed to

$$A^2 \bmod f(x) = x^{163}A_h(x) \bmod f(x) + A_l(x), \quad (14)$$

where

$$A_h(x) = 0 + a_{162}x^{161} + 0 + a_{161}x^{159} + \cdots + a_{82}x + 0,$$
$$A_l(x) = a_{81}x^{162} + 0 + a_{80}x^{160} + 0 + \cdots + 0 + a_0.$$

The high part of Eq.(14) can be computed using

$$x^{163}A_h(x) \bmod f(x) = (x^7 + x^6 + x^3 + 1)(0 + a_{162}x^{161} + 0 + a_{161}x^{159} + \cdots + a_{82}x + 0). \quad (15)$$

The architecture of square is shown in Fig.3. The square can be efficiently implemented to generate the result in one single clock cycle without huge area requirement. The critical path is the delay of three XOR gates.

IMPLEMENTATION RESULTS AND PERFORMANCE COMPARISON

The design is coded in VHDL. We synthesized and simulated the architecture using a Xilinx VirtexII XC2V6000 FPGA device, using the Synprify Pro 8.1 and Modelsim Se 6.1. It requires only 3182 clock cycles for a scalar multiplication. The work frequency of the ECC processor is around 93.3 MHz, and it takes only 34.11 μs to complete one scalar multiplication over $GF(2^{163})$. Table 1 shows the results of implementation and the hardware performance comparison for the computation of elliptic curve scalar multiplication over $GF(2^m)$.

For a fair comparison, our design was ported into the same FPGA device, Xilinx XCV2000E, as that in (Shu *et al*., 2005). All the designs listed in Table 1 have been published within the last five years. The designs in Table 1 represent the state-of-the-art with respect to scalar multiplication performance on hardware implementations. Table 1 shows that our ECC implementation has a better performance than other implementations, except that in (Sozzani *et al*., 2005). The reason is that in our proposed architecture, addition, square, and multiplication could be operated in parallel by decomposition and the data dependency was solved by optimization. Another reason is our carefully designed multiplication. The design in (Sakiyama *et al*., 2007) is almost as fast as our design due to their ASIC design. As shown in Table 1, fewer resources are used in our design compared to (Rodríguez-Henríquez *et al*., 2004; Shu *et al*., 2005).

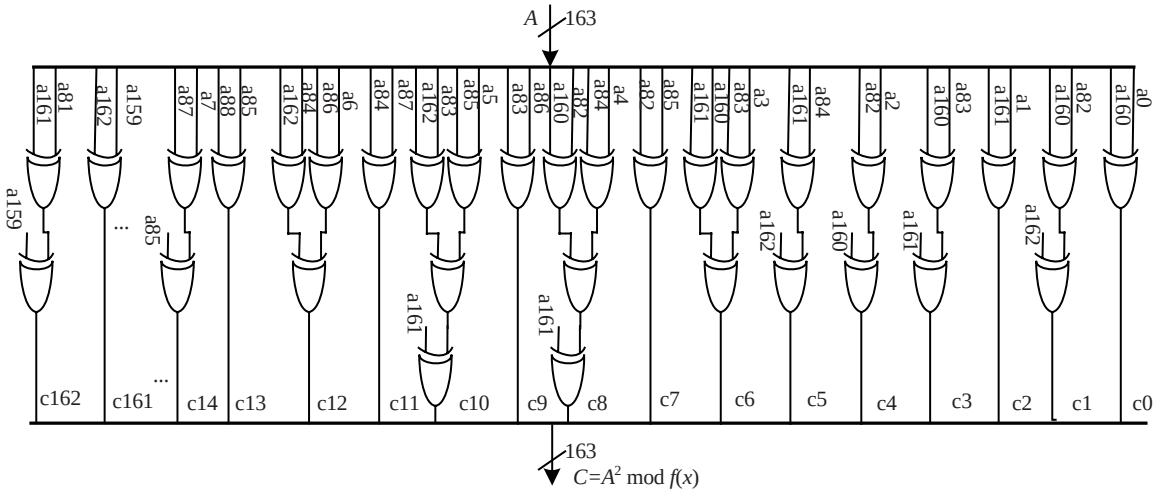

Fig.3 Architecture of the square

Table 1 Implementation results and performance comparison

Reference	Computation	Device	Resource	Frequency (MHz)	Time (μs)	Multiplier/Remarks
Rodríguez-Henríquez *et al*., 2004	2^m, 191-bit	XCV3200E	18 314 slices, 24 RAMs	9.99	56	Parallel Karatsuba, No final inversion
Shu *et al*., 2005	2^m, 163-bit	XCV2000E-7	7467 Regs, 25 763 LUTs	68.9	48	MSD, D=32, 8
Sakiyama *et al*., 2007	2^m, 163-bit	0.13 μm CMOS	115k gates	500	38	
Cheung *et al*., 2005	2^m, 162-bit	XC2V6000			60	
Sakiyama *et al*., 2006	2^m, 163-bit	Virtex-2 Pro	8450 slices	100	280	
Sozzani *et al*., 2005	2^m, 163-bit	0.13 μm CMOS	113k gates	416.7	30	
This paper	2^m, 163-bit	XC2V6000	2812 Regs, 13 376 LUTs	93.3	34.11	
This paper	2^m, 163-bit	XCV2000E-7	2819 Regs, 13 264 LUTs	53.7	59.25	

Although our design has specifically targeted the field generated by the irreducible polynomial $f(x)=x^{163}+x^7+x^6+x^3+1$, all the mechanisms discussed in this work has made no assumption about the specific field targeted, and hence can be easily adapted to accommodate other designs with different field sizes.

CONCLUSION

We propose a high-performance hardware architecture of an elliptic curve point multiplication processor over $GF(2^{163})$ scheme based on the Lopez-Dahab scalar multiplication algorithm. The finite field $GF(2^{163})$ is represented in polynomial basis and an irreducible polynomial is recommended by NIST. Our proposed architecture was implemented on a Xilinx VirtexII XC2V6000 FPGA device. Our design shows a good timing for elliptic curve scalar multiplication over $GF(2^{163})$ by performing a scalar multiplication over $GF(2^{163})$ in 34.11 μs with an optimized parallelism decomposition algorithm and carefully designed finite field multiplication. Our design is very suitable for applications such as smart cards and cellular telephones.

References

Akishita, T., 2001. Fast simultaneous scalar multiplication on elliptic curve with montgomery form. *LNCS*, **2259**:255-267. [doi:10.1007/3-540-45537-X_20]

Al-Khaleel, O., Papachristou, C., Wolff, F., Pekmestzi, K., 2007. An Elliptic Curve Cryptosystem Design Based on FPGA Pipeline Folding. 13th IEEE Int. On-line Testing Symp., p.71-78. [doi:10.1109/IOLTS.2007.15]

Balasubramaniam, P., Karthikeyan, E., 2007. Elliptic curve scalar multiplication algorithm using complementary recoding. *Appl. Math. Comput.*, **190**(1):51-56. [doi:10.1016/j.amc.2007.01.015]

Cheung, R.C.C., Telle, N.J., Luk, W., Cheung, P.Y.K., 2005. Customizable elliptic curve cryptosystems. *IEEE Tran. Very Large Scale Integr. Syst.*, **13**(9):1048-1059. [doi:10.1109/TVLSI.2005.857179]

Itoh, T., Tsujii, S., 1988. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Inf. Comput.*, **78(**3):171-177. [doi:10.1016/0890-5401(88)90024-7]

Lopez, J., Dahab, R., 1999. Fast multiplication on elliptic curves over GF(2^m) without precomputation. *LNCS*, **1717**:316-327.

Meurice de Dormale, G., Quisquater, J.J., 2007. High-speed hardware implementations of elliptic curve cryptography: a survey. *J. Syst. Archit.*, **53**(2-3):72-84. [doi:10.1016/j.sysarc.2006.09.002]

Rodríguez-Henríquez, F., Saqib, N.A., Díaz-Pérez, A., 2004. A fast parallel implementation of elliptic curve point multiplication over GF(2^m). *Microprocess. Microsyst.*, **28**(5-6):329-339. [doi:10.1016/j.micpro.2004.03.003]

Sakiyama, K., Batina, L., Preneel, B., Verbauwhede, I., 2006. Superscalar Coprocessor for High-speed Curve-based Cryptography. Proc. 8th Int. Workshop Cryptographic Hardware and Embedded Systems, **4249**:415-429. [doi:10.1007/11894063_33]

Sakiyama, K., Batina, L., Preneel, B., Verbauwhede, I., 2007. Multicore curve-based cryptoprocessor with reconfigurable modular arithmetic logic units over GF(2^m). *IEEE Trans. Comput.*, **56**(9):1269-1282. [doi:10.1109/TC.2007.1071]

Shu, C., Gaj, K., El-Ghazawi, T., 2005. Low Latency Elliptic Curve Cryptography Accelerators for NIST Curves on Binary Fields. Proc. IEEE Int. Conf. on Field-programmable Technology, p.309-310. [doi:10.1109/FPT.2005.1568575]

Sozzani, F., Bertoni, G., Turcato, S., Breveglieri, L., 2005. A Parallelized Design for an Elliptic Curve Cryptosystem Coprocessor. Proc. Int. Conf. on Information Technology: Coding and Computing, p.626-630. [doi:10.1109/ITCC.2005.25]

Wu, H., 2002. Bit-parallel finite field multiplier and squarer using polynomial basis. *IEEE Trans. Comput.*, **51**(7):750-758. [doi:10.1109/TC.2002.1017695]