FAST VLSI ARITHMETIC ALGORITHMS FOR HIGH-SECURITY ELLIPTIC CURVE CRYPTOGRAPHIC APPLICATIONS

Sangook Moon, Jaemin Park and Yongsurk Lee
Department of Electrical and Electronic Engineering, Yonsei University, Seoul, Korea

ABSTRACT

In this paper, we propose new methods for calculating fast VLSI arithmetic algorithms for secure data encryption and decryption in the Elliptic Curve Cryptosystem (ECC), and also verify the proof-of-concepts by numerical expressions and through the use of HDL (Hardware Description Language). We have developed a fast finite field multiplier that utilizes a new concept, and a finite field divider with an improved internal structure, as well as a novel fast algorithm for calculating kP, which is the most time-consuming operation in the ECC data encryption scheme. The proposed multiplier features a higher throughput per cost ratio than any other existing Galois Field (GF) multiplier that can be used in the large prime finite field. Furthermore, our improved divider shows better extensibility. The developed algorithm for point multiplication decreases the steps required for iteration by half compared to that of the traditional double-and-add algorithm. It also reduces the number of field multiplications by about 19% and that of field divisions by about 9%.

1. INTRODUCTION

As an indispensable component of modern consumer electronics, security applications, such as IC cards used for personal authentication, and domestic network applications, play an important role. In fact, such data security receives constant attention, since people tend to communicate with each other by various electronic devices over networks. Security applications are based upon intensive computations of cryptographic algorithms, which generally involve in arithmetic operations in large Galois Fields.

We classify Galois field architectures by basis representation of field elements. The most popular representations are standard (polynomial), normal, and dual basis. The dual basis representation has been used for relatively small Galois field applications, such as 8-bit Reed-Solomon codes. However, the dual basis representation is not suitable for large Galois fields used for cryptography [1]. Thus, polynomial basis and (optimal) normal basis representations have been used for cryptographic applications to date. The normal basis representation allows very efficient exponentiation, while regularity and extensibility for hardware implementation is worse than the standard basis [2]. Polynomial basis offers good solutions to most *GF* computational problems. Also, polynomial basis is the easiest to use among other representations. Therefore, we focus on using the polynomial basis throughout this document.

There are a relatively small number of publications on Galois field architectures specifically designed for cryptographic applications. Previous research on *GF* multiplications in polynomial basis representations includes general serial multiplier architectures [2], bit-systolic parallel multiplier architectures [3], and the hybrid multiplier architecture [4]. Although the hybrid multiplier architecture is known to exhibit the best throughput versus cost ratio, it cannot be used in the prime exponent $GF(2^{m=p})$ (p is prime) but only in the composite exponent $GF(2^{m=nk})$ (m is composite). Thus, the security level of applications is reduced [5].

Publications on implementing hardware *GF* dividers have been rare due to the infrequent use of division in processing finite field applications. Rather, *GF* division has been implemented using software [6]. Furthermore, if the concept of the projective coordinates is adopted, then it need not be used at all [7]. The projective coordinate scheme avoids the inversion operation at *point addition* or at *point doubling* in Elliptic Curve arithmetic operations at the cost of more field multiplications. Using projective coordinates is efficient until a polynomial inversion can be accomplished with less than 11 field multiplications, and thus is not within our consideration [8]. Brunner, *et al.* introduced a fundamental algorithm for obtaining fast *GF* inversion by using the modified Euclidian algorithm [9]. There have been attempts to decrease the throughput of inversion clock cycles based on Brunner's work using systolic arrays, but they resulted in the heavy cost of hardware resources [10].

The standard method for point multiplication has been the double-and-add algorithm, which is analogous to the repeating square-and-multiply algorithm used for exponentiation [11].

We researched the previous literatures in detail, and consolidated the strong points from other studies in order to propose novel types of the *GF* multiplier, *GF* divider, and an efficient algorithm for point multiplication. The outline of the paper is as follows: We start by introducing the mechanism of Elliptic Curve Cryptosystem in Chapter 2. Chapter 3 discusses our proposed finite field multiplier and the performance comparison with other *GF* multiplier architectures, and Chapter 4 describes our finite field divider, which is also compared with other studies. Chapter 5 covers the efficient algorithm for point multiplication, an algorithm that we developed. Our conclusions are given in Chapter 6.

Manuscript received June 25, 2001

2. ELLIPTIC CURVE CRYPTOSYSTEM

2.1 Background

There are mainly two categories of cryptographic methods for enciphering and deciphering data: secret key schemes use the same key for encryption and decryption; public key schemes use different keys. Secret key schemes are represented by DES and public key schemes are represented by RSA [12][13]. Arithmetic operations for public key schemes are much slower than those of secret key schemes having the same security level, however its existence is essential in that the key of the secret key schemes is changed frequently and encrypted by the public key cryptosystem before data transaction. The Elliptic Curve Cryptosystem (ECC), which was first suggested by Victor Miller [14] and Neal Koblitz [15] in the 1970's, has emerged as an alternative to RSA due to its shorter key lengths even at same security level. It is based on a computationally hard problem, which is the so-called discrete logarithm problem in large finite groups. Given a large finite group G and group elements $\{P_0, Q_0\} \in G$ with $Q_0 = k \cdot P_0$, then it is impossible to compute k given only P_0 and Q_0.

In the following we will only consider so-called non-supersingular elliptic curves, which provide the highest security [7]. Elliptic curve E, $P(x)$, point on the curve P are publicly known factors beforehand. Let E be a non-supersingular elliptic curve:

$$y^2 + xy = x^3 + ax^2 + b \tag{1}$$

defined over F_p ($p = 2^m$). Suppose a point $P(x, y)$ is used to generate the whole addition group of E and the message point is represented as $M(m_x, m_y)$. The elements of F_p are represented with the polynomial basis. An irreducible prime polynomial $P(x)$ is selected according to the order of the given finite field. Fig. 1 shows how the ECC based secure data exchange works. Every field operation, such as addition, multiplication, and division, in the procedure is performed by way of a finite field arithmetic modulo $P(x)$. User *Receiver* randomly chooses an integer k_a and calculates his (her) public key k_aP. The integer k_a is user *Receiver*'s secret key. User *Sender* also randomly chooses an integer k_b and obtains his (her) public key k_bP and calculates the point $k_ak_bP = (x', y')$. Now, multiplying the message M and the point k_ak_bP encrypts the message into M_{cipher}. User *Receiver* refers to the user *Sender*'s public key k_bP to calculate the shared secret point k_ak_bP. To decipher the encrypted message, user *Receiver* divides the encrypted point $M_{ciphier}$ by the shared secret point k_ak_bP modulo $P(x)$.

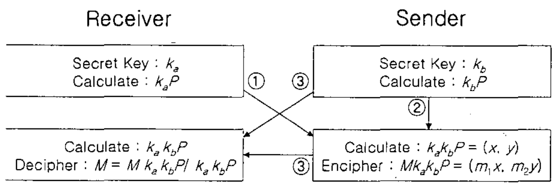

Fig. 1 Data encryption/decryption mechanism in ECC

2.2 Mathematics of the ECC-based cryptosystem

Two main operations are required to multiply an elliptic curve group element by a constant when encrypting a message: point addition (*Add*) and point double (*Double*) operations. We also include point negation (*Neg*) as a miscellaneous operation and point quadruple (*Quad*) operation, which is intended for fast implementation algorithm of kP. The *Quad* operation is explained in detail in Chapter 6.

The elliptic curve E is defined as the set of all solutions (x, y) to the equation $y^2 + xy = x^3 + ax^2 + b$ together with the point at infinity O, where b is not 0. This extra point O is needed to represent the group identity.

Rules for the above mathematical operation routines except for *Quad* operation are presented below. Rules for the *Quad* operation are given in Chapter 6.

Add routine:

Let $P(x_1, y_1)$ and $Q(x_2, y_2)$ be two different points on the curve.
If either point is O, the result is the other point.
If $P = Q$, use Double routine.
If $x_1 = x_2$ and $y_1 \neq y_2$, $P + Q = O$.
If $P \neq Q$, then $P + Q = R(x_3, y_3)$, where

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a,$$
$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1,$$
$$\text{and} \quad \lambda = \left(\frac{y_1 + y_2}{x_1 + x_2}\right). \tag{2}$$

Double routine:

Let $P(x_1, y_1) = Q(x_1, y_1)$ be a point on the curve.
If $x_1 = 0$, the result of $2P$ is O.
If $x_1 \neq 0$, $2(x_1, y_1) = R(x_3, y_3)$, where

$$x_3 = \lambda^2 + \lambda + a,$$
$$y_3 = x_1^2 + (\lambda + 1)x_3,$$
$$\text{and} \quad \lambda = \left(x_1 + \frac{y_1}{x_1}\right). \tag{3}$$

Negation routine:

Let $P(x_1, y_1)$ be a point on the curve.
$-P = R(x_3, y_3)$, or

$$-(x_1, y_1) = (x_1, x_1 + y_1). \tag{4}$$

From the above rules, we can discern the number of field operations required to carry out the routine. In the *Add* routine, 8 additions, 1 multiplication, 1 division, and 1 squaring are required. We should check that the divider of λ or (x_1 + x_2) is not zero. The *Double* routine requires 4 additions, 1 multiplication, 2 squarings, and 1 division. Also, we should check that the divider of λ or x_1 is not zero. The Negation routine requires one addition. This operation is needed when implementing the fast algorithm for the calculation of kP. As explained later in Chapter 6, the values of $(-P)$ and $(-2P)$ are needed in the algorithm we developed.

As basic mathematics for the ECC-based cryptosystem, *GF* multiplication and *GF* division occupy the most important positions. In the next 2 chapters, we present the fast and efficient *GF* multiplier architecture and *GF* divider architecture we have developed, and furthermore discuss their performances in detail.

3. FAST FINITE FIELD MULTIPLIER

3.1 Multiplier Architecture

This section describes the proposed fast finite field multiplier architecture for Galois fields using polynomial basis. *GF* multiplication is considered to be the most critical operation for performance enhancement of practical public-key cryptographic applications, which use more than 150-bit wide finite fields.

We consider arithmetic operations in one of the extension fields of $GF(2)$. The extension degree is denoted by m, so that the field can be represented as $GF(2^m)$. This field is isomorphic to $GF(2)[x]/(P(x))$, where $P(x)=x^m+\sum_{i=0}^{m-1}p_ix^i$ is an irreducible polynomial of degree m with $p_i \in GF(2)$. The calculation of the product of two arbitrary finite elements in $GF(2^m)$, $Z(x) = A(x)\cdot B(x)$, where $A(x)=\sum_{i=0}^{m-1}a_ix^i$, $B(x)=\sum_{i=0}^{m-1}b_ix^i$, proceeds as follows:

$$\begin{aligned} Z(x) &= A(x)\sum_{i=0}^{m-1}b_ix^i = \sum_{i=0}^{m-1}b_i(x^iA(x)) \bmod P(x) \qquad (5)\\ &= [b_0A(x)+\ldots+b_{m-2}x^{m-2}A(x)+b_{m-1}x^{m-1}A(x)] \bmod P(x) \end{aligned}$$

Expression (5) describes the traditional Mastrovito's serial multiplier structure [2]. The basic concept of our proposed multiplier architecture is to divide the expression into n parts to obtain n-times speed-up. We explain our architecture in detail for the case of speed-up factor n = 2. To double the overall throughput, we divide the second row of (5) into even and odd parts.

$$\begin{aligned} Z_{even}(x) &= [b_0A(x)+\ldots+b_{m-3}x^{m-3}A(x)+b_{m-1}x^{m-1}A(x)] \bmod P(x)\\ Z_{odd}(x) &= [b_1xA(x)+b_3x^3A(x)+\ldots+b_{m-2}x^{m-2}A(x)] \bmod P(x)\\ &= x[b_1A(x)+b_3x^2A(x)+\ldots+b_{m-2}x^{m-3}A(x)] \bmod P(x) \qquad (6) \end{aligned}$$

In (6), the expression of $Z_{even}(x)$ is similar to the traditional serial multiplier, except that the orders are even integers and increase by a step of 2. However, $Z_{odd}(x)$ looks complicated at a glance. We modified the expression by taking out the x term out of the brackets. Now, we can handle the $Z_{odd}(x)$ in a similar fashion. We perform the modulo reduction using the following property:

$$\begin{aligned} x^m &\equiv p_0+p_1x+\ldots+p_{m-1}x^{m-1} \bmod P(x) \qquad (7)\\ x^{m+1} &= p_0x+p_1x^2+\ldots+p_{m-2}x^{m-1}+p_{m-1}x^m \bmod P(x) \end{aligned}$$

Here, p_{m-1} in the second row of the equation (7) can be substituted with zero for simplification without any loss of generality, because we use prime polynomials with only low hamming weights (trinomials or pentanomials having p_i=1 in the lower order of the polynomial) in real-world applications. Consequently, the modulo reduction equation by x^2 is derived as follows:

$$\begin{aligned} x^2A(x) &= a_0x^2+a_1x^3+a_2x^4+\ldots+a_{m-2}x^m+a_{m-1}x^{m+1}\\ &= a_{m-2}p_0+(a_{m-2}p_1+a_{m-1}p_0)x+(a_{m-2}p_2+a_{m-1}p_0+a_0)x^2+\ldots\\ &\quad +(a_{m-2}p_{m-1}+a_{m-1}p_{m-2}+a_{m-3})x^{m-1} \qquad (8) \end{aligned}$$

This expression allows the reduction of x^2 to simultaneously produce the results of $Z_{even}(x)$ and $Z_{odd}(x)$ in the same clock cycle, thereby doubling the throughput. The first row of the equation (6) tells us that there needs to be a circuit which multiplies x^2 to the $A(x)$ represented as x^2-multiplying circuit, so that we can recursively calculate the result of the even part of $Z(x)$ in $\lceil m/2 \rceil$ clock cycles. In the third row of the equation (6), recursively calculating the same way in $\lceil (m-1)/2 \rceil$ clock cycles results in multiplying x to the overall equation. This requires one extra clock cycle while producing the result of the odd part of $Z(x)$. Thus, by choosing an odd exponent m, we can simultaneously calculate the results of Z_{even} and Z_{odd}, and obtain the final result of $Z(x)$ by adding (XORing) $Z_{even}(x)$ and $Z_{odd}(x)$ without wasting any cycles. Since the exponents used in high-security applications are always prime numbers, the choice of an odd exponent provides these advantages at one move.

Fig. 2 shows the schematic of the x^2-multiplying circuit based on (8). The symbol $\otimes$ represents the logical AND gate, while the symbol $\oplus$ represents the logical XOR gate. The p_i components in the block diagram can be thought of as on/off switches. Fig. 3 represents the implementation of $Z_{even}(x)$ in the expression (6). The part below the shaded box functions as the x^2-multiplying circuit. In the $Z_{odd}(x)$ circuit, however, there should be a part that can be operated as the x^2-multiplying circuit, as well as the traditional x-multiplying circuit required for treating the x term out of the brackets in the third row of (6). We implement $Z_{odd}(x)$ in expression (6) by simply adding (m-1) times 2-to-1 multiplexers (muxes), plus ω_t – 1 extra muxes for selecting either x- or x^2-multiplying circuit, where ω_t is the hamming weight of the prime polynomial whose value is very small relative to the order of the finite field. Fig. 4 represents the implementation of $Z_{odd}(x)$ with added components. The complementary input '0' in the $B(x)$-coefficient in the last clock cycle is required for treating the x term out of the brackets, as explained above.

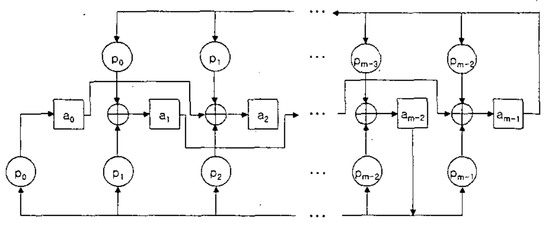

Fig. 2 x^2-multiplying circuit

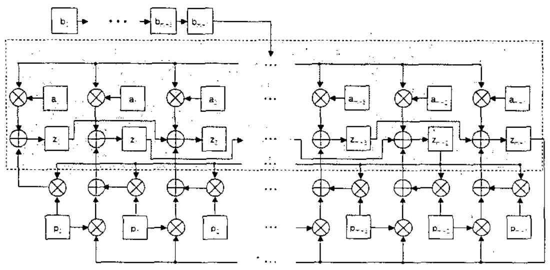

Fig. 3 Implementation of $Z_{even}(x)$

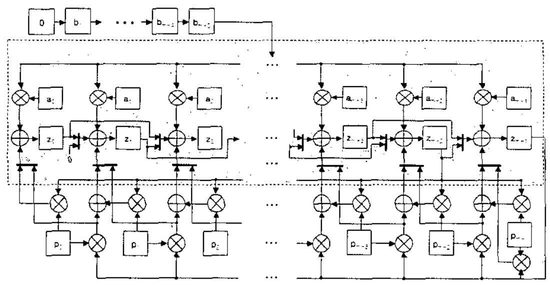

Fig. 4 Implementation of $Z_{odd}(x)$

This way, the number of clock cycles for one field multiplication is reduced by a factor of two. The most resource-consuming factor is the number of registers. The impact of the other logic gates is negligible if we customize the design of the circuits. Our double fast multiplier consumes approximately 1.5 times as many resources as the traditional multiplier architecture. We do not have to consume double the resources to obtain two-times the speed, due to sharing the $A(x)$-coefficient register.

We can extend this idea to build a multiplier that is 3-times faster than the traditional serial one by splitting the expression (5) into 3 parts: $Z_{3k}(x)$, $Z_{3k-1}(x)$, and $Z_{3k-2}(x)$. The range of the value k should depend on the situation, as we considered in the above case of the double fast multiplier.

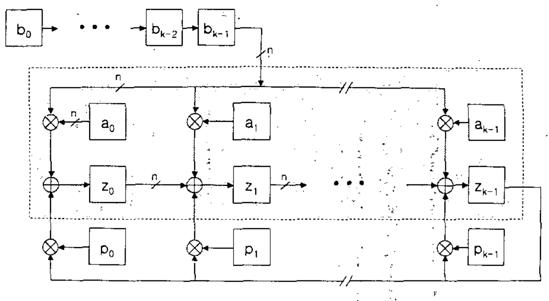

Fig. 5 Block diagram of hybrid multiplier

The hardware implementation of the $Z_{3k}(x)$ is similar to traditional serial architecture using an x^3-multiplying circuit, except that the orders are integers of multiples of 3. In implementing $Z_{3k-1}(x)$ and $Z_{3k-2}(x)$, support for an x^3-multiplying circuit and x-multiplying circuit with additional selecting muxes is required. Likewise, we can design a multiplier which is t-times as fast as the traditional serial one by using an x^t-multiplying circuit, while consuming approximately $(t+1)/2$-times the resource. Since the proposed multiplier architecture basically follows the traditional one, the critical delay path is nearly as short as that of the serial multiplier architecture independent of the speed-up factor t.

3.2 Multiplier performance evaluation

The proposed multiplier architecture has been simulated and verified with results obtained from a traditional serial multiplier modeled with HDL. In order to obtain simple and reasonable performance evaluations, we considered an implementation in $GF(2^{nk=2k})$ and compared it with the hybrid architecture, which is known to show the best throughput versus cost ratio, yet contains the critical defect that it cannot be used in the prime exponent $GF(2^{m=p})$ (p is prime), but only in the composite exponent $GF(2^{m=nk})$ (m is composite). Fig. 5 is the block diagram of the hybrid multiplier architecture. If $n = 1$, the structure operates as the traditional bit-serial architecture with all lines having one-bit connections. If $n > 1$, however, all connections are n bit wide buses and all arithmetic is performed in the subfield $GF(2^n)$, thereby producing results in k clock cycles.

We focus our comparison point on the shaded area of Fig. 3 and Fig. 4, since the width of the serial $B(x)$-coefficient register is the same in both architectures, and, according to the SEG-1 [16], the hamming weight of the suggested prime polynomials is very small compared to the exponent m, and can thus be ignored. Table 1 shows the comparison results. In the proposed architecture, there are m ANDs, m XORs and m registers in Z_{even}(x). Similarly, there are m ANDs, m XORs, $(m-1+\omega_r-1 = m+\omega_r-2)$ 2-to-1 muxes and m registers in $Z_{odd}(x)$. There are also m registers for storing A(x)-coefficients, and m XORs for the final addition. Compared to the traditional serial architecture, a critical delay path exists in $Z_{odd}(x)$ with an additional mux, while the hybrid architecture includes a parallel $GF(2^{n=2})$ multiplier through which delay increases severely as n increases. The resulting cycle time is one clock cycle longer than in the hybrid architecture, because the exponent in the case above is even, as is not the case for the prime order finite field. However, this is insignificant compared to the overall throughput.

In implementing encryption in the elliptic curve cryptosystem, finite field multiplication is the most time-critical operation, and we have proposed the above multiplier, which is t-times as fast as the traditional serial multiplier and consumes about $(t+1)/2$-times the resources, and have verified it with numerical expressions and HDL. It can also be used directly for GF squarers without any additional logic if we control the inputs. In the following section, we propose an advanced type of area/time efficient GF divider based on the modified Euclidian algorithm.

Table 1. Multiplier performance comparison results (*m*=2*k*)

$GF(2^{2k=m})$	AND	XOR	Reg	Mux	Clock	Delay
Serial architecture	m	m	$2m$	0	m	(*)
Hybrid architecture	$2m$	$\frac{3}{2}m$	$2m$	0	$\frac{m}{2}$	(**)
Proposed architecture	$2m$	$3m$	$3m$	$m+\omega_t-2$	$\frac{m}{2}+1$	(***)

(*) Register -> AND -> XOR
(**) Register -> parallel $GF(2^{n=2})$ multiplier (2 AND -> 2 XOR) -> XOR
(***) Register -> AND -> mux -> XOR

4. ADVANCED FINITE FIELD DIVIDER

4.1 Galois field division

Finite field division in the $GF(2^m)$ has the form $A(x)/B(x)$ modulo $P(x)$, where the degrees of $A(x)$ and $B(x)$ are always lower than m, and $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$ is an irreducible polynomial of degree m with $p_i \in GF(2)$. Due to the infrequent use of division in processing finite field applications, the studies of GF dividers have not been very active since they have been implemented using software. However, the need for hardware implementation has risen with the higher level of security in complicated applications. GF division proceeds through the following two steps:

Step 1. Find the inverse of $B(x)$, $B(x)^{-1}$

Step 2. $A(x)/B(x) = A(x)\cdot B(x)^{-1}$

Consequently, GF division can be expressed as GF multiplication by the inverse. The simplest way to find the inverse of a field element is by table-lookup. This table-lookup method is efficient on a relatively small finite field such as $GF(2^m)$ ($m < 8$), but it cannot be effectively implemented in VLSI in large finite fields because memory requirement increases exponentially as m increases [9]. For this reason, algorithm-based methods are employed for GF division in large finite fields. The two most frequently used algorithms for GF division are as follows:

- *Fermat's theorem*: using the theorem that $B^{2m} = B$ for any element $B \in GF(2^m)$, we can obtain the inverse of the field element B by recursive squaring and multiplication, since the field element B can be expressed as $B^{-1} = B^2 B^{2^2} B^{2^3} \dots B^{2^{m-1}}$ in $GF(2^m)$. Many software implementations adopt this method, but it cannot be effectively implemented in VLSI, since the performance versus cost ratio is not suitable [2].

- *Euclid's algorithm*: we can find the inverse of a field element in the course of obtaining the GCD of two polynomials. This method is especially useful when the field elements are represented using polynomial basis.

We propose an efficient and advanced type of GF divider using the Euclidian algorithm. There have been studies on GF dividers based on the Euclidian algorithm [9][10], and the results in [9] are area-efficient but show relatively low throughput and low extensibility. [10] employs the concept of a systolic array implementing modular structure to achieve higher throughput using a pipelining technique, but it has a very high area complexity. Our proposed divider architecture is based on the division algorithm in [9]. We improved the reference divider architecture by merging an n-bit look-up table method into a new-type GF divider that enhances area efficiency and n-times throughput. We present the adopted algorithm and the proposed idea below.

4.2 Division algorithm & improved architecture

In [9], Brunner *et al.* suggested a modified Euclidian algorithm applicable to GF inversion/division. Our divider architecture is based on this modified Euclidian algorithm as follows:

Division algorithm using the modified Euclidian method

```
Division:

P:=P(x);
S:=P(x); V:=0; (deg S = m)
R:=B(x); U:=A(x); (assume deg R = m)
delta:=0; (delta = deg S – deg R)
for i:=1 to 2m do
    if $r_m$ = 0 then
        R:=xR; U:=(xU) MOD P;
        delta+=1 (deg R-=1)
    else ($r_m$ = 1)
        if $s_m$ = 1 then
            S:=S – R; V:=(V – U) MOD P;
        end;
        S:=xS (deg S-=1)
        if delta = 0 then
            (deg S < deg R : division done)
            (R $\leftrightarrow$ S); (U $\leftrightarrow$ V);
            U:=(xU) MOD P;
            delta:=1 (deg R $\leftrightarrow$ deg S)
        else
```

```
            U:=(U/x) MOD P;
            delta-=1;
        end
    end
end (A(x)·B^-1(x) = U = V)
```

The derivation of this algorithm and the expressions for temporary values *R*, *S*, *U*, and *V* are referred to in [9]. As the above algorithm executes polynomial division, the degree of *R* or *S* decreases by one per iteration during $2m$ iterations, depending on the values of MSBs of the *R*, *S* registers and a status bit which indicates whether the value of delta is zero or not.

We improve the division algorithm suggested above by depending on the 2 most significant bits of the *R* and *S* registers simultaneously, in order to obtain the division result in *m* iterations. As a small overhead, we have to prepare a ROM table to store all possible operations since the possible number of operations increases from $2^3 = 8$ to $2^5 = 32$, which results from depending on the bits r_m, r_{m-1}, s_m, s_{m-1}, and the indicator bit.

Fig. 6 represents the block diagram of our proposed *GF* divider. *R*, *S*, *U*, and *V* circuit blocks are designed for executing 32 operations stored in the ROM table. *R* and *S* blocks are $(m+1)$-bits wide, since the irreducible polynomial $P(x)$ is stored in *S*, and there is an initial assumption that the degree of the polynomial $B(x)$ is *m*. Table 2 shows all of the operations of our *GF* divider. The reference bits r_m, r_{m-1}, s_m, s_{m-1}, and the indicator bit of the delta are combined in each iteration to produce the control signals from the CTL_ROM. *R*, *S*, *U*, and *V* cells execute the operation given in Table 2, dependent on the control signals from CTL_ROM cycle by cycle. Since we process double the work per one iteration in the same division algorithm, a modified structure of the *U* cell is introduced by inducing the expressions below, as in the design of the double-fast *GF* multiplier explained in the previous chapter:

$$x^2U(x) = u_0x^2 + u_1x^3 + u_2x^4 + \ldots + u_{m-2}x^m + u_{m-1}x^{m-1} \qquad (9)$$

$$U(x)/x^2 = u_0x^{-2} + u_1x^{-1} + u_2 + \ldots + u_{m-2}x^{m-4} + u_{m-1}x^{m-3} \qquad (10)$$

Table 2 Operations for each functional cells

Module	Operations
***R* cell**	$R, xR, x^2R, xS, x^2S,$ $x(S\text{-}R), x^2(S\text{-}R)$
***S* cell**	$S, xS, x^2S, R, x(S\text{-}xR), x^2(S\text{-}R), x(xS\text{-}R), x(R\text{-}x(S\text{-}R))$
***U* cell**	$U, x^2U, U/x^2, V, x^2V,$ $V\text{-}U, x^2(V\text{-}U)$
***V* cell**	$V, U, V\text{-}U, V\text{-}xU, U\text{-}xV, V\text{-}U/x, U\text{-}x(V\text{-}U), V\text{-}U\text{-}U/x$
Counter	delta, 0, delta+2, delta-2

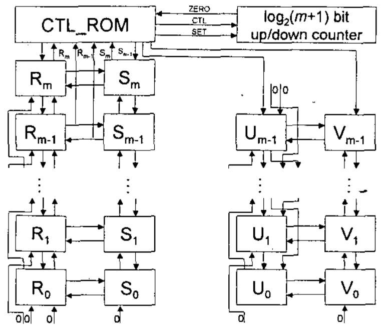

Fig. 6 Block diagram of proposed *GF* divider

We arranged the expressions using the property in (7) and implemented the expressions (9) and (10) just as we did in the design of the x^2-multiplying circuit. We finally obtain the division result $A(x)/B(x)$ after *m* iterations.

4.3 Divider performance evaluation

We compare the performance of our proposed *GF* divider with that of reference dividers implemented using the division algorithm which employs the modified Euclidian algorithm in terms of area, clock cycles, critical delay, and extensibility.

The algorithm that Brunner has suggested for designing a *GF* divider is the basis for *GF* dividers using the Euclidian algorithm, and it can be extended to produce one division result in $2m/n$ cycles with an *n*-folded form. However, Brunner's divider implementation architecture is not fit for extension on a large scale due to the fact that the critical path delay and the area in each functional cell except for the flip/flops increase proportionally to *n* with *n*-folded form. According to a paper on *GF* dividers by Guo, one cycle per one division is possible using a systolic array together with a pipelining technique, but has a weak point in area efficiency costing as much as $O(m\log_2 m)$ to achieve the throughput of one cycle/division and $O(m^2)$ for the throughput of *m* cycles/division.

In our modification of Brunner's work, the proposed structure of our *GF* divider has the throughput of $2m/n$ clock cycles per division depending on the *n* highest significant bits of *R* and *S* cells. Table 3 represents the comparative results of our proposed *GF* divider architecture, implemented to produce one division result in *m* clock cycles, with previous dividers having the same throughputs. Table 3 shows that the performance of Brunner's work and ours does not differ when implementing for a throughput of *m* clock cycles per division. Considering extensibility, however, our proposed *GF* divider requires far less overhead than in the Brunner's work. Our proposed divider, which utilizes depending on the two most significant bits simultaneously, is more advantageous than using an *n*-fold structure in that the number of cells in ours does not change requiring more operations of *R*, *S*, *U*, and *V* cells and larger ROM table exponentially proportional to *n*; yet, the number of cells and the critical delay increases in proportion to *n* in the Brunner's.

Table 3 Comparison of $GF(2^m)$ dividers

	Brunner [1]	Guo [2]	Proposed
Throughput (1/cycles)	$\frac{1}{m}$	$\frac{1}{m}$	$\frac{1}{m}$
Latency (cycles)	$3m$	$8m$-1	$3m$
Area Complexity	O(m)	O($m$log2$m$)	O(m)
Critical Delay for Extensibility	$O(n)$	$O(1)$	$O(1)$

5. $k{\cdot}P$ IMPLEMENTATION ALGORITHM

5.1 Traditional algorithm for obtaining *kP*

The most time-consuming and prevalent operation in ECC data transaction is point multiplication: kP, where k is an integer and P is a point on the curve.

$$kP = \underbrace{P+P+\ldots+P}_{k} \tag{11}$$

A single point multiplication requires multiple computations of point addition ($P \neq Q$) and point doubling ($P = Q$). When implementing encryption systems using ECC, we must use an efficient method to compute the equation (11). The standard method for point multiplication is the double-and-add algorithm, which is analogous to the repeating square-and-multiply algorithm used for exponentiation. If $k=\sum_{i=0}^{m-1} b_i 2^i$, $b_i \in \{0, 1\}$, then $kP=\sum_{i=0}^{m-1} b_i(2^i P)$. The traditional algorithm for computing kP is as follows :

Traditional algorithm for computing kP

```
kP:
  $k=\sum_{i=0}^{m-1} b_i 2^i$ , $b_i \in \{0, 1\}$
  $P:=P(x_1, y_1)$
  $Q:=P$
  for i from m-1 down to 0 do
    $Q:=Double(Q)$
    if $b_i=1$ then
    $Q:=Add\ P\ to\ Q$
  End ($Q = kP$)
```

It is clear that we need m-1 *Double* operations and ω_t *Add* operations in the traditional double-and-add algorithm. To reduce the number of iteration steps and field operations, in this chapter, we apply the modified Booth's algorithm which is well known for fast binary multiplication in computer arithmetic.

5.2 New algorithm for computing kP using redundant form

We focus on the adoption of redundancy in the modified Booth's algorithm, which is based on the fact that fewer partial products have to be generated for groups of consecutive zeros and ones when realizing fast binary multiplication [17]. Our proposed algorithm for obtaining kP is as follows:

New algorithm for computing kP using radix-4 redundancy

```
kP:
  add 0 to the left of MSB of k to make it unsigned number
  $k=\sum_{i=0}^{\lceil \frac{m}{2} \rceil -1} r_i 4^i$ , $r_i \in$ {denotations for 0P, +P, +2P, -P, -2P}
  $P:=P(x_1, y_1)$
  $2P:=Double(P)$
  $-P:=Neg(P)$
  $-2P:=Neg(2P)$
  $Q:=$ one of {0P, +P, +2P, -P, -2P} depending on $r_i$
  for i from $\lceil \frac{m}{2} \rceil - 1$ down to 0 do
    $Q:=Quad(Q)$
    if $r_i=+P$ then
    $Q:=Add\ P\ to\ Q$
    if $r_i=+2P$ then
    $Q:=Add\ 2P\ to\ Q$
    if $r_i=-P$ then
    $Q:=Add\ -P\ to\ Q$
    if $r_i=-2P$ then
    $Q:=Add\ -2P\ to\ Q$
  End ($Q = kP$)
```

(We have employed terminology from radix-4 Booth's recoding in the above algorithm).

To reduce the number of steps by half, we draw an equation for computing $4P$ from (3), as seen below:

Quad routine:

Let $P(x_1, y_1)$ be a point on the curve.
If $x_1 = 0$, the result of $4P$ is O.
If $x_1 \neq 0$, $4(x_1, y_1) = R(x_3, y_3)$, where

$$x_3 = \lambda'^2 + \lambda' + a,$$
$$y_3 = x_1^2 + (\lambda'+1)x_3,$$
$$\lambda' = x_2 + \lambda + 1 + \frac{x_1^2}{x_2},$$
$$x_2 = \lambda^2 + \lambda + a,$$
$$\text{and } \lambda = \left(x_1 + \frac{y_1}{x_1}\right). \tag{12}$$

From this formula, we can determine the number of field operations. The *Quad* routine usually requires 10 additions, 1 multiplication, 2 divisions, and 4 squarings. Fig. 7 shows a simple example of comparison between the traditional double-and-add algorithm and our proposed new algorithm using radix-4 redundant recoding.

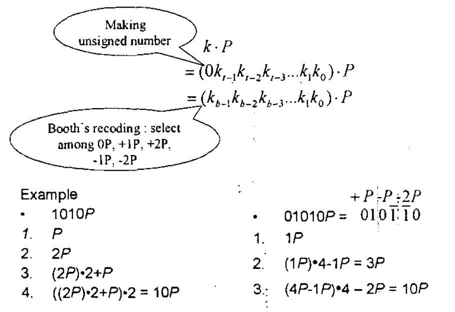
Making unsigned number
$k \cdot P$
$= (0k_{t-1}k_{t-2}k_{t-3}...k_1k_0) \cdot P$
$= (k_{b-1}k_{b-2}k_{b-3}...k_1k_0) \cdot P$
Booth's recoding : select among 0P, +1P, +2P, -1P, -2P
Example
• 1010P
1. P
2. 2P
3. (2P)•2+P
4. ((2P)•2+P)•2 = 10P
+P -P -2P
• 01010P = $010\bar{1}\bar{1}0$
1. 1P
2. (1P)•4-1P = 3P
3. (4P-1P)•4 – 2P = 10P

Fig. 7 Comparison example of two algorithms

The number of iterations decreases from m to $\left\lceil \frac{m}{2} \right\rceil + 1$ steps. Table 4 summarizes the improvement in the number of steps and required EC operations. The number of operations in Table 4 is calculated based on the probability that is dependent on the hamming weight of the prime polynomial. The probability of the existence of 1 in the binary representation of k during m steps in the double-and-add algorithm is 0.5, and the probability of the existence of non-zero Booth's recoding term is 6/8 [18]. The new algorithm exhibits a reduction of about 12.5% in handling *Add* operations. Furthermore, the new algorithm is also advantageous because of using *Quad* operations. The *Quad* routine is induced from manipulating the expressions in the *Double* routine resulting in a reduction of 1 field multiplication, and the proposed algorithm can be far more efficient by enhancing the *Quad* routine using higher mathematics in future. The proposed algorithm requires Booth's recoding circuit and memory space for storing the values of P, $2P$, $-P$, and $-2P$ additionally. The number of field operations for calculating kP are represented in detail in Table 5, which demonstrates the efficiency of our proposed algorithm. We achieved performance improvement of about 19% in multiplication considering that our GF multiplier can be used as a GF squarer, and about 9% in GF division.

Table 4 Comparison of the # of steps and EC operations between Double-and-add and our algorithm

	# of steps	*Add*	*Double*	*Neg*	*Quad*
Double-and-add	m	$\frac{1}{2}m$	m	0	0
Proposed algorithm	$\left\lceil \frac{m}{2} \right\rceil + 1$	$\frac{3}{8}m$	1	2	$\left\lceil \frac{m}{2} \right\rceil + 1$

6. CONCLUSION

We have developed new methods of calculating fast VLSI arithmetic algorithms for secure data encryption and decryption for the Elliptic Curve Cryptosystem and verified the proof-of-concepts using HDL and by numerical expressions.

Table 5 Reduction in field operations in our proposed algorithm

	Multiplication	Division	Square	Addition
Double-and-add	$\frac{3}{2}m$	$\frac{3}{2}m$	$\frac{5}{2}m$	$8m$
Proposed algorithm	$\frac{7}{8}m + 1$	$\frac{11}{8}m + 1$	$\frac{19}{8}m + 2$	$8m$+6
Reduction ratio	≈ 0.58	≈ 0.91	≈ 0.95	≈ 1

First, we designed a new type of fast finite field multiplier. We achieved as much as t-times the speed-up compared to the traditional serial multiplier architecture by manipulating the expression for the serial multiplication algorithm, resulting in a total cost of $(t+1)/2$-times the resources of the traditional serial GF multiplier. The proposed multiplier architecture has the highest throughput per cost ratio of any other GF multiplier published to date. Additionally, it can be used in the secure large prime Galois field.

Secondly, we improved the algorithm introduced in [9] in order to design a fast GF divider architecture. We improved the existing GF division algorithm by depending on the n most significant bits of temporary polynomials simultaneously. This allows us to obtain GF division results in $2m/n$ clock cycles using only an increased number of cell operations, resulting in an increase in the operational ROM table. Our GF divider architecture is especially useful when a division time of less than m clock cycles is required with minimal additional resource.

Finally, we developed a novel fast algorithm for calculating kP, which is the most time-consuming operation in the ECC data encryption scheme. We adopted the radix-4 Booth's redundancy concept to decrease the steps required for calculating kP by half compared to that of the traditional double-and-add algorithm. We also decreased the number of field operations by defining and drawing a point quadrupling operation expression.

The GF multiplier and GF divider, together with the algorithm for calculating kP, are the basic components of the ECC cryptosystem for secure data encryption and decryption. In this paper, we aimed to extract the best performance by adding the minimum overhead to exhibit a high performance versus cost ratio. At present, semiconductor-manufacturing technology is rapidly developing into nano-technology. Investment in resources to improve performance is essential. As the Internet and information technology continues to grow, the need for security applications for consumer electronics, such as IC cards for private authentication, and domestic network applications, become more and more complicated, thereby requiring intensive computations. Our proposed high-speed cryptographic algorithms for ECC are extremely well suited for such applications and exhibit versatile extensibility.

REFERENCES

[1] E. R. Berlekamp, "Bit-Serial Reed-Solomon Encoders," *IEEE Transactions on Information Theory*, Vol. IT-28, No. 6, pp. 869-874, Nov 1982.

[2] E. Mastrovito . : "VLSI Architectures for Computation in Galois Fields," PhD thesis, Dept. of Electrical Eng., Linkoping Univ., Sweden, 1991.

[3] Chin-Liang Wang and Jung-Lung Lin. : "Systolic Array Implementation of Multipliers for Finite Fields GF(2^m)," *IEEE Transactions on Circuits and Systems*, July 1991, Vol. 38, No. 7, pp. 796-800.

[4] C. Paar, P. Fleischmann, P. S-Rodriguez: "Fast Arithmetic for Public-Key Algorithms in Galois Fields with Composite Exponents," *IEEE Transactions on Computers*, October 1999, Vol. 48, No. 10, pp. 1025-1034.

[5] L. Adleman and J. DeMarrais, "A Subexponential Algorithm for Discrete Logarithms over All Finite Fields," *Advances in Cryptography-CRYPTO '93*, D. Stinson, ed., pp. 147-158, 1993.

[6] G.B. Agnew, R.C. Mullin, I.M. Onyszchuk, and S.A. Vanstone, "An Implementation for a Public-Key Cryptosystems," *J. Cryptology*, vol. 3, pp. 63-79, 1991.

[7] G.B. Agnew, R.C. Mullin, and S.A. Vanstone, "An Implementation of Elliptic Curve Cryptosystems Over $F_{2^{155}}$," *IEEE Journal on Selected Areas in Communications*, Vol. 11, No. 5, June 1993.

[8] M. Rosner, "Elliptic Curve Cryptosystems on Reconfigurable Hardware," Master's thesis, ECE Dept., Worcester Polytechnic Institute, USA, May 1998.

[9] H. Brunner, A. Curiger, and M. Hofstetter, "On Computing Multiplicative Inverses in GF(2^m)," *IEEE Transactions on Computers*, Vol. 42, No. 8, Aug. 1993.

[10] J.H Guo and C.L Wang, "Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in GF(2^m)," *IEEE Transactions on Computers*, Vol. 47, No. 10, Oct. 1998.

[11] A. J. Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, 1993.

[12] NBS, *Data Encryption Standard*, FIPS Pub. 46, U.S, National Bureau of Standards, Washington DC. Jan. 1977.

[13] Rivest, R. L. and Shamir, A. and Adleman, L., "A Method for Obtaining Digital Signature and Public Key Cryptosystem," *Comm. ACM* 21, pp. 120-126, Feb. 1978.

[14] Victor S. Miller, "Use of Elliptic Curves in Cryptography", Advances in Cryptology – CRYPTO '85 Proceedings, Lecture Notes in Computer Science, 218 (1986), Springer-Verlag, pp. 417-426.

[15] Neal Koblitz, "Elliptic Curve Cryptosystems", Mathematics of Computation, 48 n.177 (1987), pp. 203-209.

[16] SEC 1. *Elliptic Curve Cryptography*, Standards for Efficient Cryptography Group, September 1999, Working Draft.

[17] Israel Koren, *Computer Arithmetic Algorithms*, Chapter 6, pp. 99-106. Prentice Hall International, 1993.

[18] L. P. Rubinfield. "A Proof of the Modified Booth's Algorithm for Multiplications," *IEEE Transactions on Computers*, Vol. C-24, No. 10, pp. 1014-1015, Oct. 1975.

BIOGRAPHY

Sangook Moon was born in Kang-nung, Korea, in 1971. He received the B.S. and M.S. degree in electronic engineering from Yonsei University, Korea in 1995 and 1997 respectively. He is currently pursuing his Ph.D. degree at the university. His current research interests include VLSI, crypto-processors, microprocessors, computer arithmetic, and IC card design.

Jae Min Park was born in Seoul, Korea, in 1976. He received the B.S. degree in electronic engineering from Yonsei University, Korea in 2000. He is currently pursuing his M.S. degree at the university. His current research interests include VLSI, crypto-processors, network processors, microprocessors, and simultaneous multi-threading.

Yong Surk Lee was born in Seoul, Korea, in 1950. He received the B.S. degree in electrical engineering from Yonsei University, Korea in 1973 and his M.S. and Ph.D. degrees in electrical engineering, University of Michigan, Ann Arbor in 1977 and 1981 respectively. In 1993, he joined the Department of Electronic Engineering at Yonsei University where he is currently a professor. His current research interests include VLSI design, microprocessors, simultaneous multi-threading, graphic accelerators, encryption processors, and communication chip designs.