Novel Arithmetic Unit Architecture for Low Cost Elliptic Curve Cryptographic Processor

Chang Hoon Kim[1], Soonhak Kwon[2], and Chun Pyo Hong[1]
1) Dept. of Comput. & Info. Eng., Daegu University
Kyungsan, Korea
Tel : 053-850-6574 Fax : 053-850-6589
Email : chkim@dsp.daegu.ac.kr, cphong@daegu.ac.kr
2) Dept. of Mathematics, Sungkyunkwan University
Suwon, Korea
Tel : 031-290-7031 Fax : 031-290-7033
Email : shkwon@math.skku.ac.kr

Abstract

A new VLSI architecture which can be used for both division and multiplication over GF(2^m) is presented. Using substructure sharing between a modified version of the binary extended greatest common divisor (GCD) algorithm and the most significant bit first (MSB-first) multiplication algorithm, we design a compact and fast arithmetic unit (AU) for division and multiplication. The proposed AU can produce division results at a rate of one per $2m$-1 clock cycles in division mode and multiplication results at a rate of one per m clock cycles in multiplication mode respectively. Analysis shows that the computational delay time of the proposed AU, for division, is significantly less than previously proposed bit-serial dividers with reduced transistor counts. Furthermore, since the new architecture does not restrict the choice of irreducible polynomial and has the features of regularity and modularity, it is well suited for division and multiplication circuit of elliptic curve cryptographic processor.

1. Introduction

The motivation of this work is to develop high-speed arithmetic unit for elliptic curve cryptosystems (ECC) with the lowest hardware cost and maximal flexibility. In recent years, ECC have become attractive due to their small key sizes and computational efficiency [1][2]. Their low cost and compact size are critical to some applications, including smart cards and hand-held devices. Thus, a number of software [3-5] and hardware [6-8] implementations have been documented. In those implementations, GF(p), GF(p^m) and GF(2^m) have been used, where p is a prime. In particular, GF(2^m), which is an m-dimensional extension field of GF(2), is suitable for hardware implementation.

In ECC, computing kP is the most important arithmetic operation, where k is an integer and P is a point on the elliptic curve. This operation can be computed by point addition and doubling [1]. In affine coordinates, point addition and doubling can be implemented in one division, one multiplication, one squaring, and some additions over GF(2^m) [2]. Since addition in GF(2^m) is bit independent XOR operation, it can be implemented in fast and inexpensive ways. On the other hand, the multiplication and the division in GF(2^m) are much more complicated. In particular, the division is the most time and area consuming operation.

Three schemes have been used for computing inversion or division over GF(2^m) when the standard basis representation is used: 1) Repeated squaring and multiplication in GF(2^m) [9], 2) Solution of a system of linear equations over GF(2) [10], 3) Use of the extended Euclid's algorithm over GF(2) [11-13]. Among these schemes, the first and second schemes have area-time product of $O(m^3)$, while the last scheme has $O(m^2)$.

The multiplication algorithms can be classified as least significant bit first (LSB-first) and MSB-first scheme [14]. The LSB-first scheme processes the least significant bit of the second operand first, while the MSB-first scheme processes its most significant bit first. The MSB-fist scheme uses two shifting registers, while the LSB-first scheme uses three shifting registers. Therefore, the MSB-first scheme is better than the LSB-first scheme from the point of view of power consumption.

Additionally, these division and multiplication circuits for GF(2^m) can be further classified into two types: bit-serial and bit-parallel architectures. Although the bit-parallel architectures are well suited for high-speed applications, ECC requires large m (at least 163) to support a sufficient security. In other words, since the bit-parallel architectures have an area complexity of $O(m^2)$, it is not suited for this application.

In this paper, we propose a new bit-serial AU for computing both division and multiplication over GF(2^m) using the standard basis representation. Based on a modified version of the binary extended GCD algorithm and MSB-first multiplication algorithm, we design a compact and fast bit-serial AU architecture using substructure sharing. Analysis shows that the computational delay time of the proposed AU, for division, is significantly less than previously proposed bit-serial dividers with reduced chip area. Therefore, the proposed AU is well suited for division and multiplication circuit of ECC processor. Furthermore, since the proposed AU architecture does not restrict the choice of irreducible polynomial, and has the features of regularity and modularity, it provides a high flexibility and scalability with respect to the field size m.

2. GF(2^m) Field Arithmetic for ECC

In ECC, computing kP is the most important arithmetic operation, where k is an integer and P is a point on the elliptic curve. This operation can be computed using the addition of two points k times. The formulae for point addition on a curve

$$E : y^2 + xy = x^3 + a_2x^2 + a_6 \quad (1)$$

with a_2, $a_6 \in$ GF(2^m), $a_6 \neq 0$. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points in E(GF(2^m)) given in affine coordinates, where the identity of the abelian group is usually denoted by $\boldsymbol{O}$, the point at infinity. Assume $P_1, P_2 \neq \boldsymbol{O}$, and $P_1 \neq -P_2$. The sum $P_3 = (x_3, y_3) = P_1 + P_2$ is computed as follows:

If $P_1 \neq P_2$

$$\lambda = (y_1 + y_2) / (x_1 + x_2),$$
$$x^3 = \lambda^2 + \lambda + x_1 + x_2 + a_2$$
$$y^3 = (x_1 + x_3)\lambda + x_3 + y_1$$

If $P_1 \neq P_2$ (called point doubling)

$$\lambda = y_1 / x_1 + x_1,$$
$$x^3 = \lambda^2 + \lambda + a_2$$
$$y^3 = (x_1 + x_3)\lambda + x_3 + y_1$$

In either case, the computation requires one division, one squaring, and one multiplication. The squaring can be substituted by multiplication. From the point addition formulae, it can be noted that any computations except for addition can not be performed at the same time due to the data dependency. Therefore, it is desirable to share the hardware between division and multiplication circuit.

2. 1. New Architecture for division Over GF(2^m)

Let $A(x)$ and $B(x)$ be two elements in GF(2^m), $G(x)$ be the irreducible polynomial used to generate the field GF(2^m) $\cong$ GF(2)[x] / $G(x)$, and $P(x)$ be the result of the division $A(x)/B(x)$ mod $G(x)$, where

$$A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0 \quad (2)$$
$$B(x) = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_1x + b_0 \quad (3)$$
$$G(x) = x^m + g_{m-1}x^{m-1} + \cdots + g_1x + g_0 \quad (4)$$
$$P(x) = p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \cdots + p_1x + p_0 \quad (5)$$

The coefficient of each polynomial is binary digits 0 or 1. When $A(x) = 1$, $P(x)$ is called the multiplicative inverse of $B(x)$. The division result $P(x)$ can be obtained using the binary extended GCD algorithm. The algorithm is described as follows [15-16]:

[Algorithm I] The Binary Extended GCD Algorithm Over GF(2^m)
Input: $G(x)$, $A(x)$, $B(x)$
Output: V has $P(x) = A(x) / B(x)$ mod $G(x)$
Initialize: $R = B(x)$, $S = G = G(x)$, $U = A(x)$, $V = 0$

```
1.  while R ≠ 0 do
2.    while r0 == 0 do
3.      R = R / x;
4.      if u0 == 0 then U = U / x; else U = (U + G) / x;
5.      end if
6.    end while
7.    while s0 == 0 do
8.      S = S / x;
9.      if v0 == 0 then V = V / x; else V = (V + G) / x;
10.     end if
11.   end while
12.   if R ≥ S then
13.     (S, R) = (S, R + S); (V, U) = (V, U + V);
14.   else
15.     (S, R) = (R + S, R); (V, U) = (U + V, U);
16.   end if
17. end while.
```

The Algorithm I is based on the following three simple facts.

If R and S are both even,
then GCD(S, R) = xGCD(S / x, R / x) (6)

If R is even and S is odd,
then GCD(S, R) = GCD(S, R / x) (7)

If R and S are both odd, then GCD(S, R)
= GCD($|S - R| / x$, R)
= GCD(R, $|S - R| / x$)
= GCD(S, $|S - R| / x$) (8)

Although the Algorithm I is simple, it is not suitable for VLSI realization because the number of iterations is not fixed. In addition, it requires the process of comparisons relative to R and S. We solve such problems without affecting the functionality of the binary extended GCD algorithm, and the resulting algorithm is described as follows:

[Algorithm II] New Division Algorithm Over GF(2^m) for VLSI Implementation
Input: $G(x)$, $A(x)$, $B(x)$
Output: V has $P(x) = A(x) / B(x)$ mod $G(x)$
Initialize: $R = B(x)$, $S = G = G(x)$, $U = A(x)$, $V = 0$, *count* = 0, *state* = 0

```
1.  for i = 1 to 2m do
2.    if state == 0 then
3.      count = count + 1;
4.      if r0 == 1 then
5.        (R, S) = (R + S, R); (U, V) = (U + V, U);
6.        state = 1;
7.      end if
8.    else
9.      count = count - 1;
10.     if r0 == 1 then
11.       (R, S) = (R + S, S); (U, V) = (U + V, V);
12.     end if
13.     if count == 0 then
14.       state = 0;
15.     end if
16.   end if
17.   R = R / x;
18.   if u0 == 0 then
19.     U = U / x;
```

20. **else**
21. $U = (U + G) / x$;
22. **end if**
23. **end for**

The differences between the two algorithms are summarized as follows:

(a) In Algorithm I, since the initial value of S equals to $G(x)$, the constant term s_0 is always 1 at first. In addition, s_0 is always 1 at the end of the Algorithm I. Thus, if we keep the value of s_0 to be 1, the steps from 5 to 9 can be removed. Therefore, we apply two different conditions to get GCD(S, R) depending on the r_0. If r_0 is 0, we apply (7). If r_0 is 1, we apply GCD(R, $|S - R| / x$) or GCD(S, $|S - R| / x$) according to (8). As a result, since s_0 is always 1, we only need to check r_0 whether it is even or odd. Based on this result, we remove the steps from 7 to 11 in the Algorithm I.

(b) Note that, at the initial stage of the Algorithm I, the degree of S is m and the degree of R is at most (m−1). If exactly one of the degree of R or S is reduced by 1 for each iteration step, then (R = 0), (S = 1), and (V = division result) after 2m iterations without the step 12 of the Algorithm I. To realize this condition, we provide new variables *count* and *stat*e. As described in the Algorithm II, depending on the value of *state* and r_0, we apply four different conditions to get GCD(S, R). First, when the variable *state* is equal to 0 and r_0 is 0, by (7), R is scaled down and the value of *count* increases until r_0 becomes 1, respectively. Second, when the variable *state* is equal to 0 and r_0 is 1, we apply GCD(R, $|S - R| / x$) according to (8). Third, when the variable *state* is equal to 1 and r_0 is 0, we apply (7) and the value of *count* is decreased. In addition, if the value of *count* is equal to 0, we set the variable *state* to 0. This means that the degree of both S and R are exactly reduced by the increased *count* value. Finally, when the variable *state* is equal to 1 and r_0 is 1, we apply GCD(S, $|S - R| / x$) according to (8). As described above, the operation of GCD(S, R / x) is always executed for all cases and the exchange operation relative to R and S is executed according to the value of *state* and r_0. By doing this way, we can reduce the degree of R and S recursively. As a result, after 2m iterations, R, S, *state*, and *count* becomes 0, 1, 0, and 0 respectively, and V has the division result $P(x) = A(x) / B(x) \bmod G(x)$.

For the proposed division algorithm, we can find one fact: Since the conditions '*count* = 1 and *state* = 1' must occur at the beginning of the final iteration, V would not be modified. Thus, the division result V is always obtained after 2m-1 iterations.

Before implementing the proposed algorithm, we consider how to compute its main operations. Since R is a polynomial with degree of at most m and S is a polynomial with degree m, and U and V are polynomials with degree of at most m-1, they can be expressed as follows:

$$R = r_m x^m + r_{m-1}x^{m-1} + \cdots + r_1 x + r_0 \quad (10)$$
$$S = s_m x^m + s_{m-1}x^{m-1} + \cdots + s_1 x + s_0 \quad (11)$$
$$U = u_{m-1}x^{m-1} + u_{m-2}x^{m-2} + \cdots + u_1 x + u_0 \quad (12)$$
$$V = v_{m-1}x^{m-1} + v_{m-2}x^{m-2} + \cdots + v_1 x + v_0 \quad (13)$$

As described in the Algorithm II, S and V are a simple exchange operation with R and U respectively, according to the value of *state* and r_0. On the other hand, R and U have two operation parts respectively. Depending on the value of r_0, (R / x) or (($R + S$) / x) is executed. Therefore, we can get the intermediate result of R as follows:
Let

$$R' = r'_m x^m + r'_{m-1}x^{m-1} + \cdots + r'_1 x + r'_0 \quad (14)$$

Substituting (10), and (11) into (14) we get

$$r'_m = 0 \quad (15)$$
$$r'_{m-1} = r_0 s_m = r_0 s_m + 0 \quad (16)$$
$$r'_i = r_0 s_{i+1} + r_{i+1},\ 0 \le i \le m\text{-}2 \quad (17)$$

To get the intermediate result of U, First, we consider $U = (U + V)$ operation.
Let

$$U'' = u''_{m-1}x^{m-1} + \cdots + u''_1 x + u''_0 = U + V \quad (18)$$

From r_0, (12), and (13), we get

$$u''_i = r_0 v_i + u_i,\ 0 \le i \le m\text{-}1 \quad (19)$$

Assume that U''' is the operation result from the step 18 to 22 of Altorithm II and is given by (20),

$$U''' = u'''_{m-1}x^{m-1} + \cdots + u'''_1 x + u'''_0 = U / x \quad (20)$$

Substituting (12) and (4) into (20), we get the following (21) and (22).

$$u'''_{m-1} = u_0 = u_0 g_m + 0 \quad (21)$$
$$u'''_i = u_{i+1} + u_0 g_{i+1},\quad 0 \le i \le m\text{-}2 \quad (22)$$

Let

$$U' = u'_{m-1}x^{m-1} + \cdots + u'_1 x + u'_0 = U'' / x \quad (22)$$

By substituting (18) and (20) into (22), we can derive the following equations.

$$u'_{m-1} = u''_0 = u''_0 g_m + r_0 0 + 0 \quad (23)$$
$$u'_i = u''_{i+1} + u''_0 g_{i+1},\quad 0 \le i \le m\text{-}2 \quad (24)$$

In addition, the corresponding control functions of the Algorithm III are given as follows:

$$\text{Ctrl1} = (r_0 == 1) \quad (25)$$
$$\text{Ctrl2} = u_0 \text{ XOR } (v_0 \,\&\, r_0) \quad (23)$$
$$\text{Ctrl3} = (state == 0)\&(r_0 == 1) \quad (26)$$

$$count' = \begin{cases} count + 1, if\ state == 0 \\ count - 1, if\ state == 1 \end{cases} \quad (27)$$

$$state = \overline{state},\ if \begin{cases} ((r_m == 1) \& (state == 0))\ or \\ ((count == 0) \& (state == 1)) \end{cases} \quad (28)$$

Based on the main operations and control functions, we can derive a new bit-serial divider as shown in Fig. 1. The divider in Fig. 1 consists of control logic, RS-block, *m*-bit bi-directional shift register block (SR-block), and UV-block. The corresponding each function block is described in Fig. 2, Fig. 3, and Fig. 4 respectively. As shown in Fig. 3, we remove s_0 and r_m because it is always 1 and 0 respectively. To trace the value of *count*, we use *m*-bit bi-directional shift register instead of $\log_2(m+1)$-bit up/down counter to achieve a higher clock rate and it is used to multiplication, as will be explained in section 2.3. The control logic generates the control signals Ctrl1, Ctrl2, and Ctrl3 for the present iteration and updates the values of *state*. The 1-bit *c-flag* register is used to cooperate with *m*-bit bi-directional shift register and it has 1 at first. The RS-cell computes the value of *R* and *S* of the Algorithm II, and propagates the r_0 signal the control logic. As shown in Fig. 4, the *m*-bit bi-directional shift register is shifted to left or to right according to the value of *state*. When cnt_n is 1, it indicates that the value of *count* becomes *n*. In addition, when the value of *count* reduces to 0, the *z-flag* becomes 1. As a result, all the cnt_n becomes 0, the *c-flag* has 1, and the *state* is updated to 0 respectively. This is the same condition of the first computation. The UV-cell computes the value of *U* and *V* of the Algorithm II, and propagates the u_0 and v_0 signal to the control logic.

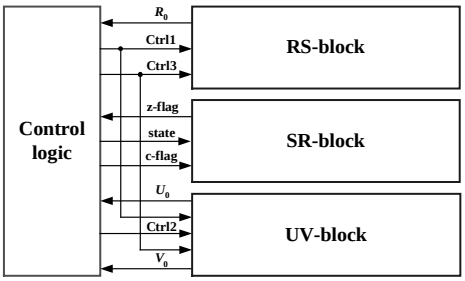

Fig. 1. New division architecture for GF(2^m)

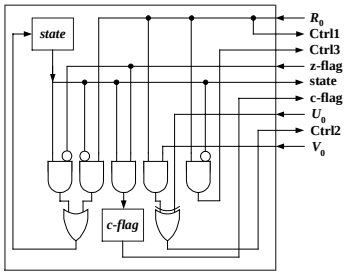

Fig. 2. Circuit of the control logic in Fig. 1

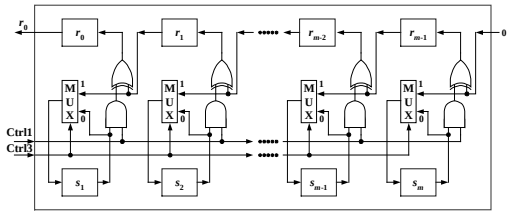

Fig. 3. Circuit of the RS-block in Fig. 1

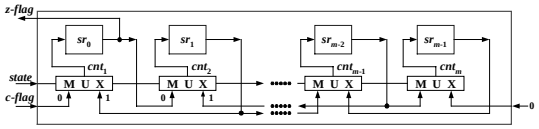

Fig. 4. Circuit of the SR-block in Fig. 1

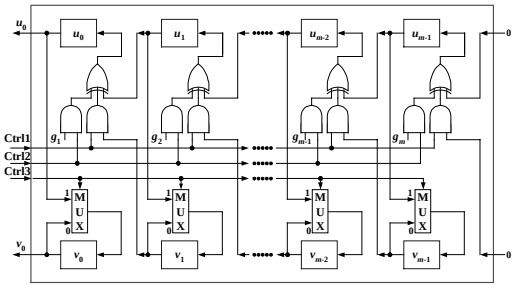

Fig. 5. Circuit of the UV-block in Fig. 1

2. 2. Multiplier Architecture Over GF(2^m)

The MSB-first multiplication of two elements in GF(2^m) can be performed as follows [14]:

$$\begin{aligned} P(x) &= A(x)B(x) \bmod G(x) \\ &= \{\cdots[A(x)b_{m-1}x \bmod G(x) + A(x)b_{m-2}]x \bmod G(x) \\ &\quad +\cdots+ A(x)b_1\}x \bmod G(x) + A(x)b_0 \end{aligned} \quad (30)$$

From (30), we can obtain the intermediate result

$$\begin{aligned} P^{(i)}(x) &= (P^{(i-1)}(x)x + b_{m-i}A(x)) \bmod G(x) \\ &= (P^{(i-1)}(x)x \bmod G(x)) + b_{m-i}A(x) \end{aligned} \quad (31)$$

where $P^{(0)}(x) = 0$ and $1 \le i \le m$.

Base on (31), we can easily implement the MSB-first multiplier as described in Fig. 6. From (31), the Multiplier in Fig. 6 can produce the multiplication results at a rate of one per *m* clock cycles.

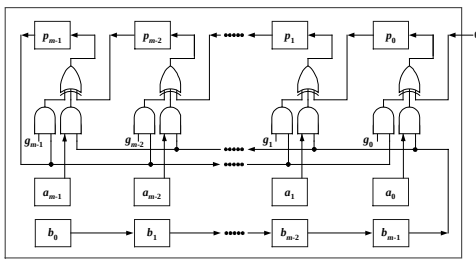

Fig. 6. MSB-first multiplier for GF(2^m)

2. 3. New Architecture for Division and Multiplication Over GF(2^m)

By observing the division and multiplication algorithm in the previous subsection, we can find the similarities: 1) The *U* operation of the divider is identical with the *P* operation of the multiplier except for the input values, 2) The *SR* register in Fig. 4 is shifted to bi-directional, while the *B* register in Fig. is shifted to unidirectional.

To perform the multiplication with the divider in Fig. 1, we modify the SR-block of Fig. 4 and the UV-block of Fig. 5. The modified SR-block and the UV-block is shown in Fig. 7 and Fig. 8 respectively. The modification procedures are summarized as follows:

(a) SR-block: we add one multiplexer and mult/div signal into the SR-block of Fig. 4 to shift the *SR* register only to the left direction when it performs multiplication. In other words, when it performs multiplication, the value of *state* is always 1 and the mult/div signal is used to distinguish between the division and the multiplication.

(b) UV-block: we add (m + 3)multiplexers and mult/div signal into the UV-block of Fig. 5. As shown in Fig. 5 and Fig. 6, because the positions of input coefficients of $G(x)$ are different, we add (m-1)multiplexers because $g_m = g_0 = 1$. Additional four multiplexers are added to perform multiplication. In multiplication mode, the u_0, *z-flag*, 0, and v_0 is selected instead of Ctrl2, Ctrl1, Ctrl3, and 0 respectively.

As described in Fig. 7 and Fig. 8, by adding (m + 4)mulitplexers and mult/div signal, we can perform both division and multiplication operations. In division mode, we clear the *B*/*SR* register in Fig. 7 and *A*/*V* register in Fig. 8, load *P*/*U* register with *U* in Fig. 8, and set mult/div signal with 1. After 2m-1 iterations, the *A*/*V* register contains the division result. In multiplication mode, we clear the *P*/*U* register in Fig. 8, load *A*/*V* register with *A* in Fig. 8, and load *B*/*SR* register with *B* in Fig. 7, and set mult/div signal with 0. After m iterations, the *P*/U register contains the multiplication result.

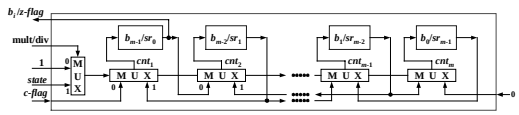

Fig. 7. Modified SR-block of Fig. 4

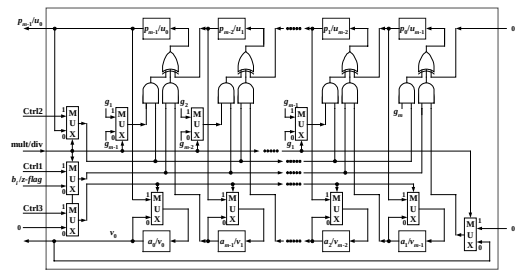

Fig. 8. Modified UV-block of Fig. 5

3. Performance Analysis

The proposed AU for division and multiplication was modeled in VHDL and simulated to verify its functionality. After verifying the proposed AU's functionality, we compared the performance of the AU with previously proposed dividers. It should be mentioned that there is no circuit for both division and multiplication at this moment to the authors' knowledge. Table 1 shows the comparison results. From Table 1, we can see that the computational delay time of the proposed architecture is significantly less than the previously proposed dividers, and it has the smallest number of transistor (TR). In Table 1, a 3-input XOR gate is constructed using two 2-input XOR gates and the number of TR estimation is based on the following assumptions: a 2-input AND gate, a 2-input XOR gate, a 2-to-l multiplexer, a 2-input OR gate, and a 1-bit latch consist of 4, 6, 6, 6, and 8 transistors respectively [17]. Although the circuit in [12] has relatively lower maximum cell delay, it involves latency of 5m-4 and a high TR counts. It may be impractical for applications requiring small chip area such as smart card. In addition, the circuit in [13] has a lower TR counts, but it involves a high maximum cell delay. In general, since ECC requires large field size m (more than 163), the AU proposed in this paper can provide the maximum throughput performance with reduced chip area.

4. Conclusions

In this paper, we proposed a compact and fast bit-serial AU for division and multiplication over GF(2^m) which can be used for ECC processor. Unlike previously architectures which can compute only one of the multiplication or division, the proposed AU architecture can perform both division and multiplication. In other words, the proposed AU produce division results at a rate of one per 2m-1 clock cycles in the division mode and multiplication results at a rate of one per m clock cycles in the multiplication mode respectively. As mentioned in the section 2, since the division and multiplication can not be performed at the same time, the proposed AU is more efficient than separated implementation of division and multiplication circuit from the point of view of hardware cost.

In addition, the computational delay time of the proposed architecture is significantly less than previously proposed bit-serial dividers with reduced TR counts. As a result, the AU proposed in this paper is well suited for division and multiplication circuit for ECC processor. Furthermore, since the proposed architecture does not restrict the choice of irreducible polynomial, and has the features of regularity and modularity, it provides a high flexibility and scalability with respect to the field size m.

Table 1. Comparison with serial dividers

	Brunner *et al.* [13]	Guo *et al.* [12]	Proposed divider
Throughput (1/cycles)	$1/2m$	$1/m$	$1/2m$
Latency (cycles)	$2m$	$5m$-4	$2m$
Maximum cell delay	Tzero-detector + $2T_{AND2} + 2T_{XOR2} + 2T_{MUX2}$	$T_{AND2} + 3T_{XOR2} + T_{MUX2}$	$2T_{AND2} + 3T_{XOR2} + T_{XOR2}$
Basic Components and Their numbers	AND_2: $3m+\log_2(m+1)$ XOR_2: $3m+\log_2(m+1)$ FF: $4m+\log_2(m+1)$ MUX_2: $8m$	AND_2: $16m$-16 XOR_2: $10m$-10 FF: $44m$-43 MUX_2: $22m$-22	AND_2: $3m$+5 XOR_2: $3m$+1 OR_2: 1 FF: $5m$+2 MUX_2: $4m$+4
# of TR	$110m+18\log_2(m+1)$	$608m$-432	$94m$+72
Operation	Division	Division	Division/Multiplication

AND_i: i-input AND gate
XOR_i: i-input XOR gate
OR_i: i-input OR gate
MUX_i: i-to-1 multiplexer
T_{ANDi}: the propagation delay through one AND_i gate
T_{XORi}: the propagation delay through one XOR_i gate
T_{MUXi}: the propagation delay through one MUX_i gate
T zero-detector: the propagation delay of $\log_2(m+1)$-bit zero-detector

Reference

[1] IEEE P1363, *Standard Specifications for Publickey Cryptography*, 2000.

[2] I. F. Blake, G. Seroussi, and N. P. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999

[3] M. Rosing, *Implementing Elliptic Curve Cryptography*, Manning, 1999

[4] D. Hankerson, J. L. Hernandez, and A. Menezes, "Implementation of Elliptic Curve Cryptography Over Binary Fields," *CHES 2000*, LNCS 1965, Springer-Verlag, 2000.

[5] D. Bailey, C. Paar, "Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography," *J. of Cryptology*, vol. 14, no. 3, pp. 153-176, 2001.

[6] G. Orlando and C. Parr, "A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m)," *CHES 2000*, LNCS 1965, Springer-Verlag, 2000.

[7] G. B. Agnew, R. C. Mullin, and S. A. Vanstone, "An Implementation for Elliptic Curve Cryptosystems Over F_2^{155}", *IEEE J. Selected Areas in Comm.*., vol. 11, no. 5, pp. 804-813, June 1993.

[8] L. Gao, S. Shrivastava and G. E. Solbelman, "Elliptic Curve Scalar Multiplier Design Using FPGAs," *CHES 2000*, LNCS 1717, Springer-Verlag, 1999.

[9] S.-W. Wei, "VLSI Architecutres for Computing exponentiations, Multiplicative Inverses, and Divisions in GF(2^m)," *IEEE Trans. Circuits Syst. II*, vol 44, pp. 847-855, Oct. 1997.

[10] C. L. Wang and J. L. Lin, "A Systolic Architecture for Computing Inverses and Divisions in Finite Fields GF(2^m)," *IEEE Trans. Comput.*, vol. 42, no. 9, pp. 1141-1146, Sep. 1993.

[11] J.H. Guo and C.L. Wang, "Hardware-Efficient Systolic Architecture for Inversion and Division in GF(2^m)," *IEE Proc. Comput. Digit. Tech.*, vol. 145, no. 4, pp. 272-278, July 1998.

[12] J.H. Guo and C.L. Wang, "Bit-serial Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in GF(2^m)", *Proc. 1997 Int. Symp. VLSI Tech., Systems and Applications*, pp. 113-117, 1997.

[13] H. Brunner, A. Curiger and M. Hofstetter, "On Computing Multiplicative Inverses in GF(2^m)," *IEEE Trans. Comput.*, vol. 42, no. 8, pp. 1010-1015, Aug. 1993.

[14] S. K. Jain, L. Song, and K. K. Parhi, "Efficient Semi-Systolic Architectures for Finite Field Arithmetic," *IEEE Trans. VLSI Syst.*, vol. 6, no. 1, pp. 101-113, Mar. 1998.

[15] D. E. Knuth, *The art of computer programming*: *Seminumerical algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1998.

[16] E. Bach and J. Shallit, *Algorithmic Number Theory - Volume I: Efficient Algorithms*, MIT Press, 1996.

[17] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*: A System Perspective. Reading, MA: Addison-Wesley, 1985.