논문 02-27-12C-11 한국통신학회논문지 '02-12 Vol.27 No.12C

타원곡선 암호시스템을 위한 $GF(2^m)$상의 비트-시리얼 나눗셈기 설계

정회원 김 창 훈*, 홍 춘 표*, 김 남 식**,권 순 학**

Design of a Bit-Serial Divider in $GF(2^m)$ for Elliptic Curve Cryptosystem

Chang Hoon Kim*, Chun Pyo Hong*, Nam Shik Kim**, Soonhack Kwon** ***Regular Members***

요 약

타원곡선 암호시스템을 $GF(2^m)$상에서 고속으로 구현하기 위해서는 빠른 나눗셈기가 필요하다. 빠른 나눗셈 연산을 위해선 비트-패러럴 구조가 적합하나 타원곡선 암호시스템이 충분한 안전도를 가지기 위해서는 m의 크기가 최소한 163보다 커야 한다. 즉 비트-패러럴 구조는 $O(m^2)$의 면적 복잡도를 가지기 때문에 이런한 응용에는 적합하지 않다. 따라서, 본 논문에서는 $GF(2^m)$상에서 표준기저 표기법을 사용하여 모듈러 나눗셈 $A(x)/B(x) \bmod G(x)$를 고속으로 수행하는 새로운 비트-시리얼 시스톨릭 나눗셈기를 제안한다. 효율적인 나눗셈기 구조를 얻기 위해, 새로운 바이너리 최대공약수(GCD) 알고리즘을 유도하고, 이로부터 자료의존 그래프를 얻은 후, 비트-시리얼 시스톨릭 나눗셈기를 설계한다. 본 논문에서 제안한 나눗셈기는 $O(m)$의 시간 및 면적 복잡도를 가지며, 연속된 입력 데이터에 대하여, 초기 $5m$-2 사이클의 지연 후, m 사이클 마다 나눗셈의 결과를 출력한다. 제안된 나눗셈기를 동일한 입출력 구조를 가지는 기존의 연구 결과들과 비교 분석한 결과 칩 면적 및 계산 지연시간 모두에 있어 상당한 개선을 보인다. 따라서 제안된 나눗셈기는 적은 하드웨어를 사용하면서 고속으로 나눗셈 연산을 수행할 수 있기 때문에 타원곡선 암호화시스템의 나눗셈 연산기로 매우 적합하다. 또한 제안한 구조는 기약 다항식(irreducible polynomial) 선택에 있어 어떤 제약도 두지 않고, 단 방향의 신호흐름을 가지면서, 매우 규칙적이기 때문에 필드 크기 m에 대해 높은 유연성 및 확장성을 제공한다.

ABSTRACT

To implement elliptic curve cryptosystem in $GF(2^m)$ at high speed, a fast divider is required. Although bit-parallel architecture is well suited for high speed division operations, elliptic curve cryptosystem requires large m(at least 163) to support a sufficient security. In other words, since the bit-parallel architecture has an area complexity of $O(m^2)$, it is not suited for this application. In this paper, we propose a new serial-in serial-out systolic array for computing division operations in $GF(2^m)$ using the standard basis representation. Based on a modified version of the binary extended greatest common divisor algorithm, we obtain a new data dependence graph and design an efficient bit-serial systolic divider. The proposed divider has $O(m)$ time complexity and $O(m)$ area complexity. If input data come in continuously, the proposed divider can produce division results at a rate of one per m clock cycles, after an initial delay of $5m$-2 cycles. Analysis shows that the proposed divider provides a significant reduction in both chip area and computational delay time compared to previously proposed systolic dividers with the same I/O format. Since the proposed divider can perform division operations at high speed with the reduced chip area, it is well suited for division circuit of elliptic curve cryptosystem. Furthermore, since the proposed architecture does not restrict the choice of irreducible polynomial, and has a unidirectional data flow and regularity, it provides a high flexibility and scalability with respect to the field size m.

* 대구대학교 컴퓨터정보공학과 (chkim@dsp.taegu.ac.kr), ** 성균관대학교 수학과 (shkwon@math.skku.ac.kr)
논문번호 : 020352-0810, 접수일자 : 2002년 8월 10일
※ 이 논문은 2002학년도 대구대학교 학술연구비에 의하여 연구되었음

Ⅰ. 서 론

타원곡선 암호시스템은(ECC: Elliptic Curve Cryptosystem) RSA나 DSA와 같은 공개키 암호시스템에 비해 사용되는 키 길이도 현저히 작고(약 1/6정도) 연산도 효율적이다[1-2]. 따라서 최근 ECC의 효율적인 소프트웨어[3-5] 및 하드웨어 구현[6-8]에 관한 많은 연구가 발표되었다. ECC를 위해 사용되는 유한체는 $GF(p)$, $GF(p^m)$, 그리고 $GF(2^m)$이 있다 (여기서 p는 소수이다). 이중 $GF(2^m)$은 0과 1을 원소로 같는 $GF(2)$의 m 차원 확장 필드로 특히 하드웨어 구현에 적합하다. 또한 $GF(2^m)$상의 연산은 기저의 선택에 따라 다르게 정의된다. ECC의 구현에 사용되는 $GF(2^m)$의 기저 표기법에는 정규기저(normal basis) 및 표준기저(standard basis)가 있다. 각 기저표기법은 장단점을 가지는데, 정규기저 표기법을 사용할 경우 제곱연산은 단순히 순환 쉬프트로 구현되기 때문에 지수 연산을 빠르게 할 수 있고, 표준기저를 사용할 경우 곱셈 및 나눗셈 연산을 위한 하드웨어구조가 매우 규칙적이고 필드 크기 m에 대하여 높은 확장성을 제공한다. 따라서 ECC를 하드웨어로 구현할 경우 표준 기저가 정규기저 보다 더 적합하다 할 수 있다.

ECC의 구현에 있어 kP는 가장 중요한 연산이다. 여기서 k는 큰 정수이고 P는 $GF(2^m)$상에서 정의된 타원곡선상의 한 포인트이다. kP를 연산하기 위해서는 타원곡선상의 포인트 덧셈 연산과 두 배 연산이 필요한데 좌표계의 선택에 따라 연산은 다르게 정의된다[2]. $GF(2^m)$상의 타원곡선을 위한 좌표계는 크게 Affine과 Projective로 구분될 수 있다. Affine 좌표계에 있어 포인트 덧셈 연산 및 두 배 연산은 $GF(2^m)$상에서 한번의 나눗셈, 곱셈 그리고 제곱연산으로 구현되어지는 반면 Projective 좌표계에서는 포인트 덧셈 연산은 열 다섯 번의 곱셈과 다섯 번의 제곱연산이 필요하며, 포인트 두 배 연산은 다섯 번의 곱셈 및 제곱 연산이 각각 필요하다[2]. 따라서 $GF(2^m)$상에서 표준기저 표기법을 사용할 때, 나눗셈 연산을 고속으로 수행 할 수 있다면, Affine 좌표계를 이용하여 ECC를 고속으로 구현 할 수 있다.

$GF(2^m)$상에서 표준 기저표기법을 사용할 경우 나눗셈 연산을 하드웨어로 구현하기 위해 1) 선형 방정식 집합의 해를 구하는 방법[9], 2) Fermat의 이론[10], 3) Euclid의 GCD 알고리즘[11-12]이 이용되어 왔다. 첫 번째 방법은 $2m$-1개의 미지수를 가진 $2m$-1개의 선형 방정식들의 해를 구함으로써 역원을 찾아낸다. 두 번째 방법은 Fermat의 이론을 이용하여 연속적인 제곱과 곱셈을 함으로써 역원을 찾는다. 즉 $A/B = AB^{-1} = AB^{2^m-2} = A(B(B(B\cdots B(B(B)^2)^2\cdots)^2)^2)^2$ 의 관계식을 이용하는데, 이 방법은 m번의 제곱연산과 약 $\log_2{}^{(m-1)}$번의 곱셈연산을 필요로 한다. 정규기저 표기법에 있어, 제곱연산은 순환 쉬프트 연산이기 때문에, 정규기저 표기법을 사용할 경우 많이 이용된다. 마지막 방법은 Euclide의 GCD 알고리즘을 이용하는 방법으로 $\mathrm{GCD}(A(x), B(x)) = W(x)\cdot A(x)+U(x)\cdot B(x)$에서 $A(x)$에 기약다항식, $G(x)$를 넣으면 $\mathrm{GCD}(G(x),B(x))=W(x)\cdot G(x)+U(x)\cdot B(x) = 1$ 에서 $U(x)\cdot B(x) \equiv 1 \bmod G(x)$가 되어 $U(x)$가 $B(x)$의 역원 즉, $B^{-1}(x)$가 된다. 최근 연구 결과에 의하면 표준 기저표기법을 사용할 경우 Euclide의 GCD 알고리즘을 사용했을 때 가장 적은 하드웨어의 사용으로 가장 낮은 계산 지연시간을 가지는 것으로 나타났으며, 나눗셈 연산을 추가적인 곱셈연산 없이 할 수 있다[11].

이러한 나눗셈기 들은 비트-패러럴[10][11] 혹은 비트-시리얼[9][12] 형태의 구조를 가진다. 고속의 나눗셈 연산을 위해서는 비트-패러럴 구조가 적절하나 ECC에서는 필드의 크기 m이 적어도 163 이상이어야 하기 때문에 비트-패러럴 구조는 높은 면적 복잡도와 핀 개수 문제 때문에 구현에 많은 어려움이 따를 뿐 아니라 비실용적이다. 따라서 적은 하드웨어의 사용으로 고속의 나눗셈 연산을 할 수 있는 하드웨어구조는 필요하다. 또한 [13]과 같은 비-시스톨릭 구조는 시스톨릭 구조에 비해 낮은 칩 면적을 가지지만, m이 크질 경우 전역 신호전파 때문에 심각한 속도저하를 초래할 뿐 아니라, 신뢰성을 제공하지 못한다[11].

본 논문에서는 $GF(2^m)$상에서 표준 기저 표기법을 사용하여, 나눗셈을 위한 비트-시리얼 형태의 시스톨릭 구조를 제안한다. 제안한 나눗셈기는 수정된 이진 GCD 알고리즘에 기반하며, 이 알고리즘으로부터 자료의존 그래프(DG: Dependence Graph)를 얻고, 일차원 신호 흐름 그래프(SFG: Signal Flow Graph)를 설계한 후 컷셋시스톨릭화 기법(cut-set systolization techniques)[14]을 적용하여, $GF(2^m)$상의 완전한 나눗셈 연산기를 얻는다. 제안된 나눗셈기는 $O(m)$의 시간 복잡도와 $O(m)$의 공간 복잡도를 가지며, 연속된 입력 데이터에 대하여, 초기 $5m$-2 사이클의 지연 후, m 사이클 마다 곱셈의 결과를 출력한다.

본 연구에서 제안된 시스톨릭 나눗셈기를 동일한 입출력 형태를 가지는 기존의 시스톨릭 나눗셈기들 [9][12]과 비교 분석한 결과 칩 면적 및 계산 지연 시간에 있어 상당한 개선을 보인다. 따라서 제안된 나눗셈기는 적은 하드웨어를 사용하면서 고속으로 나눗셈 연산을 수행할 수 있기 때문에 타원곡선 암호화시스템의 나눗셈 연산기로 매우 적합하다. 또한 제안된 구조는 기약다항식 선택에 있어 어떤 제약도 두지 않고, 단 방향의 신호흐름을 가지면서, 매우 규칙적이기 때문에 필드크기 m에 대해 높은 유연성 및 확장성을 제공한다.

본 논문의 구성은 다음과 같다. 2절에서 $GF(2^m)$상의 연산에 대해서 간단히 살펴 본 후, 나눗셈 연산을 위한 새로운 바이너리 확장 GCD 알고리즘을 얻는다. 3절에서는 새로운 알고리즘에 기반 하여, 비트-시리얼 나눗셈기를 설계하고 FPGA를 이용하여 구현 및 그 기능을 검증한다. 4절에서는 기존의 나눗셈기들과의 성능을 비교, 분석하고, 5절에서 결론을 맺는다.

Ⅱ. $GF(2^m)$상의 새로운 나눗셈 알고리즘

2.1 $GF(2^m)$상의 연산

$GF(p^m)$은 p^m개의 원소들로 구성된다. 여기서 p는 소수이고 m은 양의 정수이다. 특히 $GF(2^m)$은 2^m개의 원소들을 포함하며 0과 1을 원소로 가지는 $GF(2)$의 확장 필드이다. 모든 $GF(2^m)$은 원소 0(zero element), 항등원(unit element), 원시 원소(primitive element) 그리고 적어도 하나의 기약 다항식 $G(x) = x^m + g_{n-1}x^{m-1} + g_{m-2}x^{m-2} + \cdots + g_1x + g_0$을 포함한다. 원시원소 α는 기약다항식 $G(x)$의 근이고 $GF(2^m)$의 0이 아닌 모든 원소들을 생성한다. $GF(2^m)$의 0이 아닌 모든 원소들은 원시원소 α의 지수로 표현되어 질 수 있다. 즉 $GF(2^m) = \{1, \alpha^1, \alpha^2, \cdots, \alpha^{2^m-2}\}$. α가 기약 다항식 $G(x)$의 근이기 때문에 $G(\alpha) = 0$ 이다. 따라서 식 (1)을 얻을수 있다.

$$\alpha^m = g_{m-1}\alpha^{m-1} + g_{m-2}\alpha^{m-2} + \cdots + g_1\alpha + g_0 \quad (1)$$

식 (1)로부터 $GF(2^m)$의 원소들은 $G(x)$로 모듈로 연산을 취해서 차수가 m 보다 작은 다항식으로 유일하게 표현할 수 있다. 즉 α^t $(0 \le t \le 2^m-2)$ mod $G(x)$ 연산을 취한다. 이러한 표현법을 표준 혹은 다항식 표기법이라고 하며, 집합 $\{1, \alpha^1, \alpha^2, \cdots, \alpha^{m-1}\}$이 $GF(2^m)$의 표준 기저를 형성한다.

표준 기저 표기법을 위한 $GF(2^m)$상의 연산들은 일반적인 바이너리 연산들과 매우 다르다. $GF(2^m)$의 두 원소의 덧셈 연산은 간단히 비트별 XOR 연산이다. $A(x)$ 와 $B(x)$를 $GF(2^m)$의 두 원소들에 대한 다항식 표현들이라 하면 $A(x)+B(x) = \sum_{i=0}^{m-1}(a_i + b_i)x^i$ 의 관계식이 성립한다. 여기서 괄호 안의 덧셈은 XOR 즉 mod 2 덧셈 연산이다. 반면 곱셈 및 나눗셈은 복잡한 연산이다. 특히 나눗셈이 가장 복잡한 연산으로 ECC의 성능에 가장 큰 영향을 미친다. 이 두 연산은 $A(x)$에 $B(x)$를 곱하거나 나눈 후 $G(x)$로 모듈로를 취해야만 한다.

2.2 VLSI 구현을 위한 $GF(2^m)$상의 새로운 바이너리 확장 GCD 알고리즘

$A(x)$와 $B(x)$는 $GF(2^m)$의 두 원소이고, $G(x)$는 $GF(2^m) \cong GF(2)[x]/G(x)$를 정의하는 차수 m의 기약 다항식이고, $P(x)$는 $A(x)/B(x)$ mod $G(x)$의 결과라고 하면, 각각의 다항식은 다음과 같이 표현되며 계수들은 이진수 0 혹은 1이다. 특히, $A(x)=1$이면 $P(x)$는 $B(x)$의 역원이다.

$$A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0 \quad (2)$$

$$B(x) = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_1x + b_0 \quad (3)$$

$$G(x) = x^m + g_{m-1}x^{m-1} + \cdots + g_1x + g_0 \quad (4)$$

$$P(x) = p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \cdots + p_1x + p_0 \quad (5)$$

두 정수의 GCD를 구하기 위해 [알고리즘 1]은 사용될 수 있으며, 아래의 3가지 사실에 기반한다 [15][16].

If R and S are both even,
then $\mathrm{GCD}(S, R) = 2\mathrm{GCD}(S/2, R/2)$ (6)
If R is even and S is odd,
then $\mathrm{GCD}(S, R) = \mathrm{GCD}(S, R/2)$ (7)
If R and S are both odd,
then $\mathrm{GCD}(S, R) = \mathrm{GCD}(|S-R|/2, R)$
$= \mathrm{GCD}(R, |S-R|/2)$ (8)
$= \mathrm{GCD}(S, |S-R|/2)$

[알고리즘 1]에서 x가 y보다 큰 소수이고 모든 연산들이 $GF(x)$상에서 정의된다면 알고리즘의 종료후 b는 y의 역원이 된다. 다시 말하면, x가 소수이기 때문에 $\mathrm{GCD}(x, y) = ax + by = 1$에서 $by \equiv$ mod

$x = 1$ 이 되어 b는 x의 역원을 가진다[15]. 따라서 [알고리즘 1]에 있는 연산들을 $GF(2^m)$상의 연산들로 바꾸고 x에 기약 다항식 $G(x)$를 그리고 y에 $GF(2^m)$상의 한 원소 $B(x)$를 대입하면, [알고리즘 1]은 $GF(2^m)$상의 역원연산 $1/B(x) \bmod G(x)$를 수행할수 있다. 이를 위해 [알고리즘 1]을 기능에는 영향을 주지 않고 수정하면 [알고리즘 2]를 얻을 수 있으며, 수정과정은 아래와 같다.

1) [알고리즘 1]의 두 정수 x와 y는 [알고리즘 2]에서 다항식 $G(x)$와 $B(x)$로 각각 대체 되었고 GCD($G(x)$, $B(x)$)=1 된다는 사실은 알기 때문에 [알고리즘 1]에서 GCD를 구하기 위해 사용된 변수 g는 제거될 수 있다. 또한 $G(x)$는 항상 홀수이기 때문에 [알고리즘 1]의 단계 2부터 4는 제거 될 수 있다. 우리가 구하고자 하는 것은 y의 역원인 b이기 때문에 a를 구하기 위해 사용된 변수 A와 C 역시 제거될 수 있다.

2) $GF(2^m)$상의 뺄셈과 덧셈은 비트별 XOR 연산이기 때문에 식 (9) 의 관계가 성립한다.

$$S - R = R - S = S + R \tag{9}$$

식 (9)로부터 [알고리즘 1]의 '-'는 모두 '+'로 모두 대체 되었다.

두 단계의 수정 과정에서 알 수 있듯이 GCD를 구하는 기능에는 아무런 영향을 미치지 않았기 때문에 [알고리즘 2]는 $GF(2^m)$상에서 역원연산 $1/B(x) \bmod G(x)$를 수행할 수 있다. 또한 [11]에 따라, U에 1대신 다항식 $A(x)$를 대입하면 나눗셈 연산 $A(x)/B(x) \bmod G(x)$을 수행할 수 있다. 표 1은 [알고리즘 2]에 기반한 나눗셈의 한 예를 보여준다. 여기서 m=4, $G(x) = x^4 + x + 1$, $A(x) = x^3 + x^2 + x$, 그리고 $B(x) = x^3 + x + 1$과 같다. 표 1에 나타나듯이 알고리즘의 종료 시 V는 $A(x)/B(x) \bmod G(x)$의 정확한 결과인 $x + 1$을 가진다.

표 1. [알고리즘 2]에 따른 $GF(2^4)$상에서의 나눗셈 예

step	R	S	U	V
Init.	x^3+x+1	x^4+x+1	x^3+x^2+x	0
1	x^3+x+1	x^4+x^3	x^3+x^2+x	x^3+x^2+x
2	x^3	$x+1$	x^3+x+1	x^2+1
3	1	x	$x+1$	x^2+x
4	0	1	0	$x+1$

[알고리즘 1] : 정수상에서의 바이너리 확장 GCD 알고리즘

Input: two positive integers x and y.
Output: integers a, b, and z such that $ax + by = z$, where $z = \mathrm{GCD}(x, y)$.

1. g=1;
2. **while** x and y are both even **do**
3. x=x/2, y=y/2, g=2g;
4. **end while**
5. s=x, r=y, A=1, V=0, C=0, U=1;
6. **while** r is even **do**
7. r=r/2;
8. **If** $C \equiv U \equiv 0 \pmod 2$ **then**
9. C=C/2, D=U/2;
10. **else**
11. C=(C+y)/2, U=(U-x)/2;
12. **end if**
13. **end while**
14. **while** s is even **do**
15. s=s/2;
16. **If** $A \equiv V \equiv 0 \pmod 2$ **then**
17. A=A/2, V=V/2;
18. **else**
19. A=(A+y)/2, V=(V-x)/2;
20. **end if**
21. **end while**
22. **If** $r \geq s$ **then**
23. r=r-s, C=C-A, U=U-V;
24. **else**
25. s=s-r, A=A-C, V=V-U;
26. **end if**
27. **If** r==0 **then**
28. a=A, b=V, **return**(a, b, z=$g \cdot s$);
29. **else**
30. go to step 6;
31. **end if**

[알고리즘 2] : $GF(2^m)$상의 바이너리 확장 GCD 알고리즘

Input: $G(x)$, $A(x)$, $B(x)$
Output: V has $P(x)=A(x)/B(x) \bmod G(x)$
Initialize: $R=B(x)$, $S=G=G(x)$, $U=A(x)$, V=0

1. **while** $R \neq 0$ **do**
2. **while** r_0 == 0 **do**
3. R = R/x;
4. **if** u_0 == 0 **then**
5. U = $U/x \bmod G$;
6. **else**
7. U = (U+G)/x;
8. **end if**
9. **end while**
10. **while** s_0 == 0 **do**
11. S = S/x;
12. **if** v_0 == 0 **then**
13. V = $V/x \bmod G$;
14. **else**
15. V = (V+G)/x;
16. **end if**
17. **end while**
18. **if** $R \geq S$ **then**
19. (S, R)=(S, R+S); (V, U)=(V, U+V);
20. **else**
21. (S, R)=(R+S, R); (V, U)=(U+V, U);
22. **end if**
23. **end while.**

[알고리즘 2]는 비록 간단하지만, 반복 회수가 고정되어있지 않은 뿐 아니라 R과 S에 대한 비교 부분이 있기 때문에 VLSI 구현에 적합하지 않다. 따라서 우리는 이런한 문제들을 해결하기 위해 식 (6)부터 (8)에 기술되어 있는 GCD의 사실을 이용한다. 그 결과 다음의 유도과정에 따라 [알고리즘 3]과 같은 $GF(2^m)$상의 나눗셈을 위한 새로운 바이너리 확장 GCD 알고리즘을 얻을 수 있다.

1) [알고리즘 2]에서 S의 초기값은 $G(x)$이기 때문에 처음엔 항상 홀수이고, GCD(S, R) = 1 이기 때문에 알고리즘이 종료할 때도 홀수이다. 따라서 우리는 S를 항상 홀수로 유지한다면 [알고리즘 2]의 단계 10부터 17을 제거할 수 있다. 이를 구현하기 위해 우리는 식 (7)과 (8)로부터, R이 짝수라면 식 (7)을 적용하고 R이 홀수이면 식 (8)에 따라 (S, R) = (R, $|S-R|/2$) 또는 (S, $|S-R|/2$)를 적용한다. 따라서 S의 값은 항상 홀수이기 때문에 우리는 단지 R의 값이 홀수인지 또는 짝수 인지만 검사하면 된다.

2) S의 차수는 m이고 R의 차수는 기껏해야 m-1이기 때문에 각 반복에 있어 S 혹은 R의 차수를 1씩 감소시킨다면 [알고리즘 2]의 단계 18에 있는 R과 S에 관한 비교 조건 없이 $2m$의 반복 후 R은 항상 0이 되고 S는 1이 되어 (GCD(S, R) = 1) V는 나눗셈 결과를 가진다. 다시 말하면, 알고리즘의 초기에 R이 n만큼 차수가 감소 된 후 r_0가 1일 때, 식 (8)에 따라 (S, R) = (R, $|S-R|/2$)을 적용한다면, 그 다음 반복에 있어 R은 S의 차수를 감소시키게 된다. 여기서 S가 R의 차수가 감소된 양 n만큼 감소되지 않은 상황에서 r_0가 1일때, 우리는 식 (8)에 따라 (S, $|S-R|/2$)을 적용한다면 S의 차수 역시 n만큼 감소시킬 수 있다. 이를 구현하기 위해 우리는 새로운 변수 *count*와 *state*를 추가한다. 따라서 [알고리즘 3]에 기술된 것처럼 우리는 *state*와 R에 따라 네 가지의 다른 조건을 적용한다: ① *state*가 0이고 R이 짝수이면, 식 (7)을 적용하고 R이 홀수가 될 때까지 R의 차수는 1식 감소하고 *count*는 1씩 증가한다, ② *state*가 0이고 R이 홀수이면, 식 (8)에 따라 (S, R) = (R, $|S-R|/2$)와 같이 적용한다, ③ *state*가 1이고 R이 짝수이면, 식 (7)을 적용하고 *count*를 감소시킨다. 또한 *count*가 0이면, 초기의 반복 조건과 같기 때문에 state는 0으로 설정한다, ④ *state*가 1이고 R이 홀수이면, 식 (8)에 따라 (S, R) = (S, $|S-R|/2$)을 적용한다. 위의 4가지 조건을 보면 GCD(S, $R/2$)는 항상 적용되고 *state*와 r_0 따라 S 와 R은 바뀌기 때문에 우리는 R과 S의 차수를 재귀적으로 감소 시킬수 있다. 마지막으로 나눗셈 결과를 얻기 위해 R과 S의 연산에 따라 U와 V를 확장하면, $2m$의 반복 후 R, S, *state* 그리고 *count*는 각각 0, 1, 0, 0이 되고 V는 나눗셈 결과 $P(x)=A(x)/B(x) \bmod G(x)$를 가진다.

[알고리즘 3]을 분석해보면 알고리즘의 마지막 반복을 시작할 때 *count*와 *state*는 항상 1이 된다는 사실을 알 수 있다. 따라서 V값에는 아무런 영향을 미치지 못하기 때문에 우리는 $2m$-1의 반복 후에 정확한 나눗셈결과 V를 얻을 수 있다. 표 2에 [알고리즘 3]을 수행하는 한 예를 보인다. 표 2에 나타나듯이 7($2m$-1)번 반복 후 V는 정확한 나눗셈 결과인 $x+1$을 가진다.

[알고리즘 3] : $GF(2^m)$상의 나눗셈을 위한 새로운 바이너리 확장 GCD 알고리즘

Input: $G(x)$, $A(x)$, $B(x)$
Output: V has $P(x)=A(x)/B(x) \bmod G(x)$
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.              (S, R)=(R, R+S); (V, U)=(U, U+V);
6.              state = 1;
7.          end if
8.      else
9.          count = count-1;
10.         if r0 == 1 then
11.             (S, R)=(S, R+S); (V, U)=(V, U+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
```

표 2. [알고리즘 3]에 따른 $GF(2^4)$상에서의 나눗셈 예

i	*state*	*count*	R	S	U	V
Init	0	0	x^4+x+1	x^4+x+1	x^4+x^2+x	0
1	1	1	x^4+x^2	x^4+x+1	x^2+x+1	x^4+x^2+x
2	0	0	x^2+x	x^4+x+1	x^4+x	x^4+x^2+x
3	0	1	$x+1$	x^4+x+1	x^2+1	x^4+x^2+x
4		2	x^2	$x+1$	x^4+x^2	x^2+1
5	1	1	x	$x+1$	x^2+x	x^2+1
6	0	0	1	$x+1$	$x+1$	x^2+1
7	1	1	1	1	$x+1$	$x+1$
8	0	0	0	1	0	$x+1$

Ⅲ. 비트-시리얼 시스톨릭 어레이 설계 및 FPGA 구현

3.1 핵심연산

2절에서 제안된 알고리즘을 구현하기 전에 제안된 알고리즘의 핵심연산에 대해서 고려한다. [알고리즘 3]의 S와 R은 차수가 m인 다항식 그리고 U와 V는 차수가 m-1인 다항식으로 각각 표현 할 수가 있으며, 각 다항식은 아래와 같다.

$$R = r_m x^m + r_{m-1}x^{m-1} + \cdots + r_1 x + r_0 \quad (10)$$

$$S = s_m x^m + s_{m-1}x^{m-1} + \cdots + s_1 x + s_0 \quad (11)$$

$$U = u_{m-1}x^{m-1} + u_{m-2}x^{m-2} + \cdots + u_1 x + u_0 \quad (12)$$

$$V = v_{m-1}x^{m-1} + v_{m-2}x^{m-2} + \cdots + v_1 x + v_0 \quad (13)$$

[알고리즘 3]의 단계 5와 11에 나타나듯이 S와 V 연산은 $state$와 r_0에 따른 간단한 교환 연산이다. [알고리즘 3]의 단계 17의 R에 관한 연산을 살펴보면, 단계 5와 11에서 S에는 항상 홀수를 유지하고, R이 홀수일 때 S와 R을 비트별 XOR 연산 후 R에 대입하기 때문에 r_0는 항상 0이다. 따라서 단계 17의 (R/x)는 1-비트 오른쪽 쉬프트 연산이다. R'을 식 (14)와 같이 (R/x)의 연산 결과라 가정하고, 식 (10)을 식 (14)에 대입하면 우리는 식 (15)와 (16)을 얻는다.

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

$$r'_m = 0 \quad (15)$$

$$r'_i = r_{i+1} \quad 0 \le i \le m-1 \quad (16)$$

[알고리즘 3]의 단계 18부터 22에 있는 U의 연산 결과는 아래의 식 (17)과 동일하다.

$$U = U / x \bmod G \quad (17)$$

식 (1)로부터 우리는 아래의 식 (18)을 얻는다.

$$x^m \bmod G(x) = g_{m-1}x^{m-1} + g_{m-2}x^{m-2} + \cdots + g_1 x + g_0 \quad (18)$$

식 (18)에서 g_0는 1이기 때문에 식 (18)은 식 (19)와 같이 다시 표현할 수 있다.

$$1 = (x^{m-1} + g_{m-1}x^{m-2} + \cdots + g_2 x + g_1)x \bmod G(x) \quad (19)$$

식 (19)의 양변을 x로 나누면 식 (20)을 얻는다.

$$x^1 \bmod G(x) = x^{m-1} + g_{m-1}x^{m-2} + \cdots + g_2 x + g_1 \quad (20)$$

식 (17)의 결과를 식 (21)과 같이 두고 식 (12)와 (20)을 식 (21)에 대입하면 식 (22)과 (23)을 얻는다.

$$U' = u'_{m-1}x^{m-1} + u'_{m-2}x^{m-2} + \cdots + u'_1 x + u'_0 \quad (21)$$

$$u'_{m-1} = u_0 \quad (22)$$

$$u'_i = u_{i+1} + u_0 g_{i+1} \quad 0 \le i \le m-2 \quad (23)$$

3.2 제안된 알고리즘의 자료의존 그래프

제안된 알고리즘으로부터 유도된 $GF(2^m)$상의 역원 및 나눗셈연산을 위한 DG는 그림 1과 같다 (여기서 m=3). 그림 1의 DG는 (2m-1)개의 Type-1 셀과 (2m-1)·m개의 Type-2 셀로 구성되며, Type-1 및 Type-2 셀의 구조는 각각 그림 2와 3과 같다. DG의 i번째 열은 알고리즘의 i번째 반복을 수행하며, 첫 번째 열은 $A(x)$, $B(x)$ 그리고 $G(x)$를 입력으로 받고, 마지막 열은 나눗셈 결과인 $P(x)$를 출력한다. [알고리즘 3]으로부터 초기에 s_0는 1과 같고 알고리즘이 종료 될 때까지 1을 유지하기 때문에 생략 될 수 있다. 또한 식 (15)로부터 r_m은 항상 0이기 때문에 역시 생략 될 수 있다.

제안된 알고리즘으로부터 Type-1 셀은 현재와 다음 반복을 위한 제어신호들을 생성하고, Type-2 셀

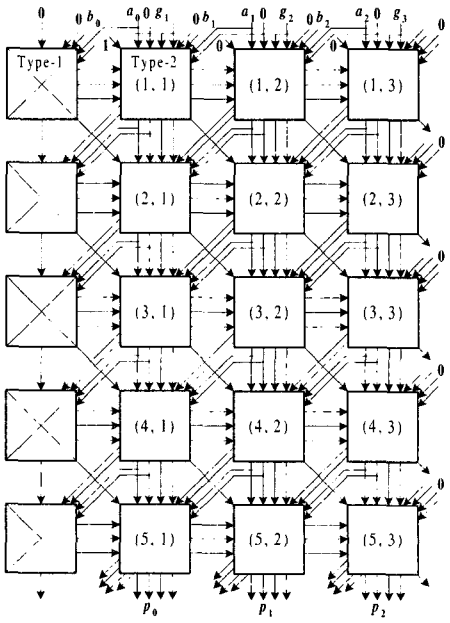

그림 1. 역원 및 나눗셈을 위한 GF(23)상의 새로운 자료의존 그래프

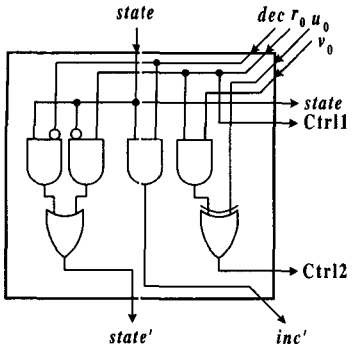

그림 2. 그림1의 Type-1 셀의 회로도

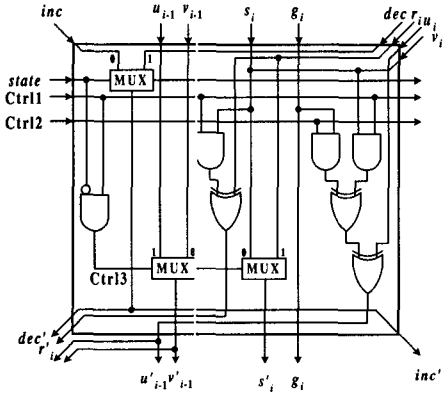

그림 3. 그림 1의 Type-2 셀의 회로도

은 하나의 제어신호를 포함하며, Type-1 셀의 제어 신호로부터 3.1절에 설명된 연산들 및 *count*의 값을 계산한다는 사실을 알 수 있다.

각 셀에 대한 자세한 설명에 앞서 우리는 *count* 의 구현에 대해 고려한다. [알고리즘 3]에서 *count* 의 값은 최대 m까지 증가한다. 이를 구현하기 위해서는 Type-1 셀에 $\log_2^{(m+1)}$ 비트의 덧셈기를 추가하여야 한다. 이 경우, [14]의 투영절차에 의해 일차원 SFG를 얻으면, 각 처리기(PE: Processing Element)는 $\log_2^{(m+1)}$의 덧셈기를 포함해야 한다. ECC와 같이 m이 크다면 이는 분명히 비실용적이다. 이 문제를 해결하기 위해 우리는 [12]에서 사용된 방법을 적용하며, 다음과 같이 요약할 수 있다.

*count*의 값은 기껏해야 m까지 증가하기 때문에 m-비트 크기의 양방향 쉬프트 레지스터를 이용해서 이를 구현할 수 있다. 다시 말해서, 현재의 *count* 값이 n이면($0 \le n \le m$) n번째 레지스터의 값은 1이고 나머지는 모두 0으로 처리하고, 이를 현재의 *count* 값이 n이라고 가정하면 된다. 따라서 그림 1에 있는 DG의 각 열에 m 비트 크기의 양방향 쉬프트 레지스터를 구현하기 위해 각각의 Type-2 셀에 2-to-1 멀티플렉서를 추가하고, (1, 1)의 Type-2 셀에 1을 입력하고, 첫 번째 열의 나머지 Type-2 셀에는 0을 입력한다. 이것을 현재의 *count* 값이 0이라고 간주하고, *state*로 *count*값을 증감시킨다. *count*의 구현 결과와 함께 Type-1 및 Type-2 셀의 기능에 대해 아래에 기술한다.

1) Type-1 셀: Type-2 셀을 제어하기 위해 아래의 네 가지 제어신호를 생성한다.

$$\text{Ctrl1} = (r_0 == 1) \tag{24}$$

$$\text{Ctrl2} = u_0 \text{ XOR } (v_0 \ \& \ r_0) \tag{25}$$

$$inc' = (state == 1) \ \& \ (dec == 1) \tag{26}$$

$$state = \overline{state}, \text{ if } \begin{cases} ((r_m == 1) \& (state == 0)) \text{ or} \\ ((dec == 1) \& (state == 1)) \end{cases} \tag{27}$$

Ctrl2는 [알고리즘 3]의 단계 18에 있는 u_0를 결정하기 위해 사용된다. *state*가 1이 되고 *dec*가 1이 되면, *count*가 0이 되는 것을 의미한다. 따라서 *state*는 다시 0이 되고, *inc*' = 1이 된다. 또한 그 열의 모든 Type-2 셀의 *inc*' = *dec*' = 0이 되어 결국 알고리즘이 시작될 때와 같이 *count* = 0이 된다.

2) Type-2 셀: Type-1 셀로부터 *state*, Ctrl1 그리고 Ctrl2를 받아 3.1에 설명된 연산들 및 *count*의 값을 계산한다. 또한 [알고리즘 3]의 단계 5와 11을 수행하기 위해 식 (28)의 제어신호 Ctrl3을 생성한다. 그림 3에 나타나듯이 *state*가 0이면, *inc*가 선택되어 *count* = *count* +1이 되고 그렇지 않으면 *dec*가 선택되어 *count* = *count* -1이 된다.

$$\text{Ctrl3} = (state == 0) \ \& \ \text{Ctrl1} \tag{28}$$

3.3 비트-시리얼 시스톨릭 어레이 구현

그림 1의 자료의존 그래프를 동쪽으로 투영시키면[14] 그림 4의 SFG를 얻을 수 있다. 그림 4는 $2m$-1개의 PE를 가지며, PE구조는 그림 5에 나타난다. 그림 5에서 '•' 은 1-비트 1-사이클 지연 소자이다. 그림 4의 SFG 어레이는 길이 m의 100…00 비트열에 의해 제어된다. 또한 투영 절차에 따라 각 처리기는 Type-1과 Type-2셀 들을 포함해야 한다. 즉 PE_i는 그림 1 DG의 i번째 열의 연산을 수행한다. 그림 1의 DG로부터, 각각의 열 반복에서 Ctrl1, Ctrl2, 그리고 *state*는 모든 Type-2 셀에 전파되어야 되기 때문에 3개의 2-to-1 멀티플렉서와 3개의 플립

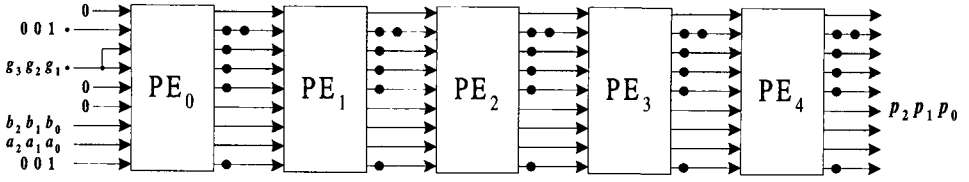

그림 4. $GF(2^3)$의 역원 및 나눗셈을 위한 일차원 SFG

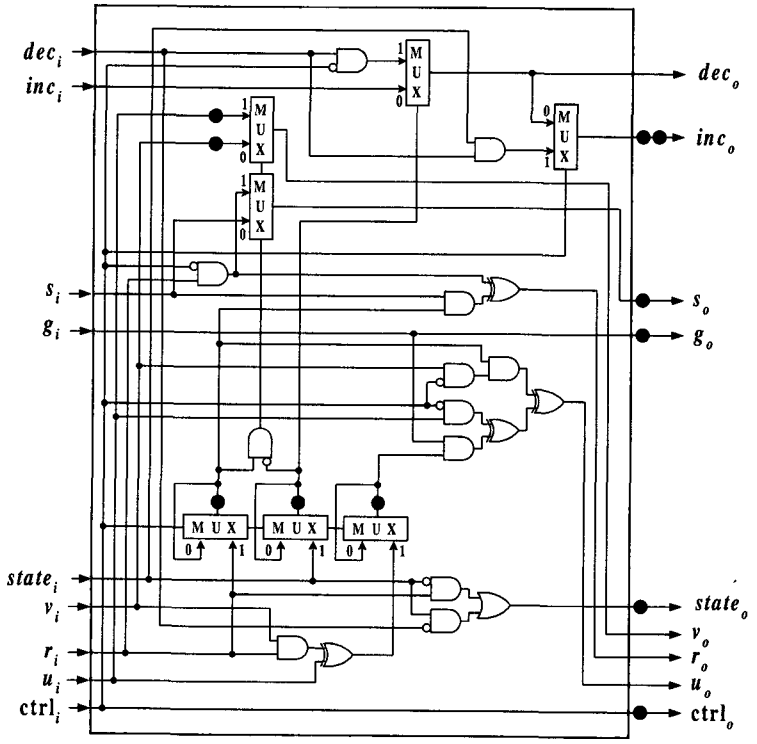

그림 5. 그림 4의 PE 회로도

-플롭을 추가한다. 또한 DG의 가장 오른쪽 셀에는 4개의 0이 입력되기 때문에 4개의 AND게이트를 추가하였으며, 제어 논리가 1일때 각각 0을 생성한다.

완전한 비트-시리얼 시스톨릭 나눗셈기를 얻기 위해 컷-셋 시스톨릭화 기법[14]을 적용하면, 그림 6의 나눗셈기를 얻을 수 있다. 그림 6에 기술된 것처럼 각 입력 데이터는 LSB부터 입력이 되며, 결과 역시 LSB부터 출력된다. 또한 연속된 입력에 대해서, 초기 $5m$-2 클럭 사이클 지연 후, 매번 m 클럭 사이클 비율로 나눗셈 연산을 수행할 수 있다.

3.4 제안한 나눗셈기의 FPGA 구현 및 검증

그림 6에 있는 나눗셈기의 기능 검증을 위해, VHDL로 회로를 기술하였고, Mento Graphics사의 합성 툴(LeonardSpectrum Version: 2002c.15)을 사용하여 회로를 합성, Net-list 파일을 추출한 후, Altera사의 Quartus II 2.1을 이용하여 Place & Route 과정을 거친 후, 타이밍 분석 및 시뮬레이션을 수행하였다. 여기서 Altera사의 150만 게이트급인 EP2A40F1020C7을 대상 디바이스로 선택하였다.

그림 7은 $GF(2^8)$상의 나눗셈기 회로에 대한 타이밍 시뮬레이션 결과를 나타내며, 214(MHz)의 최대 동작 주파수를 얻었다.

시뮬레이션 과정에서 입력된 2개의 연속된 데이터는 식 (29)부터 (31)과 같고 정확한 결과는 식 (32)와 같다.

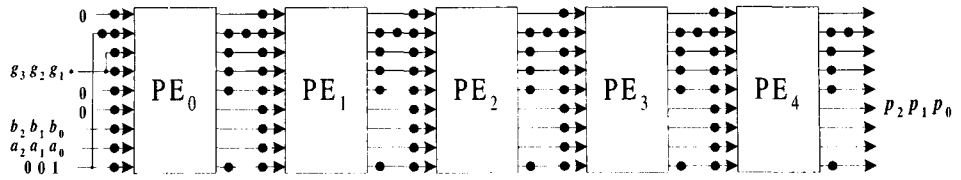

그림 6. $GF(2^3)$의 새로운 비트-시리얼 시스톨릭 나눗셈기

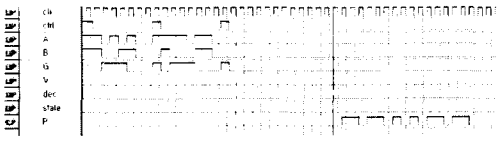

그림 7. $GF(2^8)$상의 타이밍 시뮬레이션 결과

$$A(x)=1+x+x^3+x^5,\ 1+x+x^2+x^3+x^5+x^6 \quad (29)$$

$$B(x)=1+x+x^4+x^5,\ x+x^5+x^6 \quad (30)$$

$$G(x)=x^2+x^3+x^4+x^8,\ x^2+x^3+x^4+x^8 \quad (31)$$

$$P(x)=x+x^2+x^4+x^5+x^7,\ x+x^3+x^4+x^6+x^7 \quad (32)$$

그림 7에 나타나듯이 본 연구에서 제안된 비트-시리얼 시스톨릭 나눗셈기는 $38(5m\text{-}2)$ 클럭 사이클 후에 첫 번째 결과 값인 $(x + x^2 + x^4 + x^5 + x^7)$을, $8(m)$ 클럭 사이클 후에 두 번째 결과 값인 $(x + x^3 + x^4 + x^6 + x^7)$을 각각 출력한다. 또한 본 연구에서는 필드 크기 m을 최대 571까지 확장하여 시뮬레이션을 수행한 결과 모두 정확한 결과를 얻었다. 표 3은 다양한 필드 크기에 대한 FPGA 구현 결과이다. 참고로 표3에서 m = 571은 [1]에서 권고하는 필드 크기 중 가장 큰 값이다. 표 3으로부터 본 연구에서 제안한 나눗셈기는 전역신호 전파가 없기 때문에 필드 크기를 크게 하여도 최대 동작 주파수에는 거의 영향을 미치지 않는다는 사실을 알 수 있다.

표 3. 제안된 나눗셈기의 FPGA 구현 결과

	m=163	m=233	m=571
LE 개수	7477	10697	26421
칩 사용율(%)	19	28	68
최대 동작 주파수 (MHz)	180.96	174.13	176.74

LE(Logic Element): 하나의 4-입력 룩업테이블, 하나의 프로그래머블 레지스터 그리고 캐리 체인 등으로 구성되어 있음[18].

Ⅳ. 성능분석

본 연구에서 제안한 시스톨릭 나눗셈기를 동일한 입출력 형태를 가지는 기존의 나눗셈기들과 비교하였다. 표 4에 성능 비교 결과를 요약하였으며, 조금 더 자세한 비교를 위해 3-입력 XOR 게이트와 4-입력 XOR 게이트는 각각 2개의 2-입력 XOR 게이트 그리고 3개의 2-입력 XOR 게이트로 구성된다고 가정하였다. 또한 2-입력 AND 게이트, 2-입력 XOR 게이트, 2-to-1 MUX, 2-입력 OR 게이트, 그리고 1-비트 래치는 각각 4, 6, 6, 6 그리고 8개의 트랜지스터로 구성된다고 가정하였다[17].

표 1에 기술된바와 같이, Wang 등이 제안한 나눗셈기는 $O(m^2)$의 면적 복잡도를 가지는 반면, 본 연구에서 제안한 나눗셈기는 $O(m)$의 면적 복잡도를 가진다. 또한 제안한 나눗셈기는 Wang 등이 제안한 나눗셈기에 비해 약 $2m$ 사이클만큼 처리지연 시간이 감소하고 처리율은 약 두 배로 증가함을 알 수 있다. 본 연구에서 제안한 나눗셈기를 Guo 등이 제안한 나눗셈기와 비교할 때, 동일한 면적 복잡도를 가지지만, 훨씬 적은 트랜지스터(TR, Transistor)를 사용하면서 각 셀당 $(T_{XOR2} + T_{MUX2})$만큼의 계산 지연시간을 줄이기 때문에, 전체적으로 약 $5m(T_{XOR2} + T_{MUX2})$만큼 계산 지연시간이 감소함을 알 수 있다. 따라서 본 연구에서 제안한 나눗셈기는 기존의 나눗셈기들에 비해 계산지연 시간 및 칩 면적에서 상당한 개선을 보인다.

Ⅴ. 결 론

본 연구에서는 ECC를 위한 비트-시리얼 시스톨릭 나눗셈기를 제안하였다. 제안된 나눗셈기는 새로운 바이너리 확장 GCD 알고리즘으로부터 설계되었으며, FPGA 구현을 통하여 정확히 동작함을 확인하였다.

제안된 나눗셈기는 $O(m)$의 시간 및 면적 복잡도를 가지며, 연속적인 데이터 입력에 대하여 초기 $5m\text{-}2$ 사이클의 지연 후, m 사이클마다 나눗셈의 결

표 4. $GF(2^m)$상의 비트-시리얼 시스톨릭 나눗셈기들의 특성 비교

	Wang *et al.* [9]	Guo *et al.* [12]	제안된 나눗셈기
처리율 (1/cycles)	$1/(2m-1)$	$1/m$	$1/m$
지연시간 (cycles)	$7m-3$	$5m-4$	$5m-2$
최대 처리기 지연시간	$T_{AND2}+T_{XOR2}+T_{MUX2}$	$T_{AND2}+3T_{XOR2}+T_{MUX2}$	$T_{AND2}+2T_{XOR2}$
셀의 구성요소	AND_2 : $3m^2+3m-2$ XOR_2 : $1.5m^2+1.5m-1$ Latch : $6m^2+8m-4$ MUX_2 : $3m^2+m-2$	AND_2 : $16m-16$ XOR_2 : $10m-10$ Latch : $44m-43$ MUX_2 : $22m-22$	AND_2 : $24m-12$ XOR_2 : $8m-4$ OR2 : $2m-1$ Latch : $40m-19$ MUX_2 : $14m-7$
TR 갯수	$87m^2+91m-58$	$608m-432$	$560m-272$
면적 복잡도	$O(m^2)$	$O(m)$	$O(m)$
시간-면적 복잡도	$O(m^3)$	$O(m^2)$	$O(m^2)$
제어신호 수	2	1	1

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 ANDi gate
T_{XORi} : the propagation delay through one XORi gate
T_{MUXi} : the propagation delay through one MUXi gate

과들을 출력한다. 제안된 나눗셈기를 기존의 동일한 입출력 형태를 갖는 비트-시리얼 시스톨릭 나눗셈기들과 비교 분석한 결과 칩 면적 및 계산 지연시간 모두가 상당히 개선되었다. 서론에서 기술한 바와 같이 $GF(2^m)$상에서 표준기저 표기법을 사용할 때, 나눗셈 연산을 고속으로 수행 할 수 있다면, Affine 좌표계를 이용하여 ECC를 고속으로 구현 할 수 있다. 따라서 본 연구에서 제안한 나눗셈기는 Affine 좌표계를 이용하는 ECC의 구현에 적합하다고 할 수 있다. 또한 제안된 나눗셈기는 기약 다항식 선택에 있어 어떤 제약도 받지 않을 뿐 아니라, 단 방향의 신호흐름 및 매우 규칙적인 구조를 가지기 때문에 필드 크기 m에 대하여 높은 유연성 및 확장성을 제공한다.

참 고 문 헌

[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] 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.

[10] 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.

[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.Y. Kung, *VLSI Array Processors*. Englewood Cliffs, N.J.: Prentice Hall, 1988.

[15] D. E. Knuth, *The art of computer programming: Seminumerical algorithms*, 3rd edn. 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.

[18] Altera, APEX™II Programable Logic Device Family Data Sheet, Aug. 2000. http://www.altera.com/literature/lit-ap2.html.

김 창 훈(Chang Hoon Kim) 정회원

2000년 2월 : 대구대학교 컴퓨터정보공학부 학사
2000년 3월~현재 : 대구대학교 컴퓨터정보공학과 석사과정

<주관심 분야> 암호 시스템, 내장형 시스템, 재구성형 컴퓨팅

홍 춘 표(Chun Pyo Hong) 정회원

1978년 2월 : 경북대학교 전자공학과 학사
1986년 12월 : Georgia Institute of Technology, Electrical and Computer Engineering 석사
1991년 12월 : Georgia Institute of Technology, Electrical and Computer Engineering 박사
1992년 9월~현재 : 대구대학교 정보통신공학부 교수
<주관심 분야> DSP 하드웨어 및 소프트웨어, 컴퓨터 구조, VLSI 신호처리, 내장형 시스템

김 남 식(Nam Shik Kim) 정회원

1996년 2월 : 성균관대학교 수학과 (학사)
2000년 2월 : 포항공과대학교 수학과 (석사)
<주관심 분야> 암호이론, 부호이론, 유한체 연산기

권 순 학(Soonhack Kwon) 정회원

1990년 2월 : KAIST 수학과 (학사)
1992년 2월 : 서울대학교 수학과 (석사)
1997년 5월 Johns Hopkins University (박사)

1998년 3월~현재 : 성균관대학교 수학과 조교수
<주관심 분야> 정수론, 암호이론, 회로이론