새로운 유한체 나눗셈기를 이용한 타원곡선암호(ECC) 스칼라 곱셈기의 설계

준회원 김 의 석*, 정회원 정 용 진**

Design of ECC Scalar Multiplier based on a new Finite Field Division Algorithm

Eui-seok Kim* *Associate Member*, Yong-jin Jeong** *Regular Members*

요 약

본 논문에서는 타원곡선 암호 시스템을 위한 스칼라 곱셈기를 유한체 GF(2^{163})상에서 구현하였다. 스칼라 곱셈기는 stand basis를 기반으로 비트-시리얼 곱셈기와 나눗셈기로 구성되어 있으며 이 가운데 가장 많은 시간을 필요로 하는 나눗셈의 효율적인 연산을 위해 확장 유클리드 알고리즘 기반의 새로운 나눗셈 알고리즘을 제안하였다. 기존의 나눗셈기들이 가변적인 데이터 종속성으로 인해 제어 모듈이 복잡해지며 처리 속도가 느린 것에 비해 새로이 제안하는 나눗셈 알고리즘은 입력신호의 크기에 독립적인 2-bit의 제어 신호만을 필요로 하기 때문에 기존의 나눗셈기에 비하여 하드웨어 사이즈 및 처리 속도면에서 유리하다. 또한 제안하는 나눗셈기의 연산 모듈은 규칙적인 구조를 가지고 있어 입력 신호의 크기에 따라 확장이 용이하다. 새로운 스칼라 곱셈기는 삼성전자 0.18 um CMOS 공정으로 합성하였을 경우 60,000게이트의 하드웨어 사이즈를 가지며 최대 250MHz까지 동작이 가능하다. 이 때 데이터 처리속도는 148kbps로 163-bit 프레임당 1.1ms 걸린다. 이러한 성능은 디지털 서명, 암호화 및 복호화 그리고 키 교환등에 효율적으로 사용될 수 있을 것으로 여겨진다.

Key Words : ECC, Scalar Multiplication, Finite Field

ABSTRACT

In this paper, we proposed a new scalar multiplier structure needed for an elliptic curve cryptosystem(ECC) over the standard basis in GF(2^{163}). It consists of a bit-serial multiplier and a divider with control logics, and the divider consumes most of the processing time. To speed up the division processing, we developed a new division algorithm based on the extended Euclid algorithm. Dynamic data dependency of the Euclid algorithm has been transformed to static and fixed data flow by a localization technique, to make it independent of the input and field polynomial. Compared to other existing scalar multipliers, the new scalar multiplier requires smaller gate counts with improved processor performance. It has been synthesized using Samsung 0.18 um CMOS technology, and the maximum operating frequency is estimated 250 MHz. The resulting performance is 148 kbps, that is, it takes 1.1 msec to process a 163-bit data frame. We assure that this performance is enough to be used for digital signature, encryption/decryption, and key exchanges in real time environments.

* 광운대학교 전자통신공학과 실시간구조연구실(check95@explore.kw.ac.kr), ** 광운대학교 부교수(yjjeong@daisy.kw.ac.kr)
논문번호 : 030476-1103, 접수일자: 2003년 11월 5일
※본 연구는 광운대학교 산학연센터 및 IDEC의 툴 지원으로 이루어졌습니다.

Ⅰ. 서 론

최근 네트워크 기술이 발달하며 전자상거래가 활성화됨에 따라서 전자서명과 인증, 암호화에 대한 중요성이 대두되고 있다. 이러한 정보 보안에 필요한 알고리즘으로는 공개키와 비밀키 알고리즘이 있으며, 이 중 공개키 암호 알고리즘은 전자서명, 인증 및 암호화에 필연적으로 사용되고 있다. 현재 사용되는 대표적인 공개키 암호 알고리즘으로는 소인수 분해의 어려움에 기초한 RSA(Rivest Shamir Adleman)와 이산 대수 문제에 기반을 둔 ECC(Elliptic Curve Cryptography)가 있다. 이중 ECC는 작은 키 사이즈를 가지고도 동일한 안전성을 갖는 장점이 있는데, 예를 들어, 163-bit의 ECC가 1024-bit RSA와 동일한 안전도를 가지는 것으로 알려져 있다.

ECC의 주요 응용 분야로는 전자서명, 암호화 및 복호화(Encrytpion and Decryption), 그리고 키 교환(Key Exchange)등이 있다. 이러한 암호 시스템은 개인용 컴퓨터의 발달로 굳이 하드웨어로 구현하지 않더라도 소프트웨어로 충분히 구현이 가능하다. 하지만 스마트 카드와 같은 메모리나 프로그램 코드의 크기가 제한된 임베디드 프로세서를 사용하는 어플리케이션의 경우 소프트웨어만으로 공개키 암호 시스템을 처리하는데는 속도와 용량에서 한계가 있기 때문에 하드웨어 코프로세서의 사용이 필요하게 된다. 이러한 면에서 ECC는 키 사이즈가 상대적으로 작고 그로 인해 하드웨어 사이즈와 전력소모가 적어 소형 어플리케이션에서의 보안 코프로세서로서 사용되기에 적합하다.

본 논문의 구성은 다음과 같다. 2장에서는 지금까지 알려진 스칼라 곱셈기의 구현 사례를 살펴보고, 3장에서는 ECC에 대해 간단히 소개한다. 4장에서는 전체 스칼라 곱셈기와 스칼라 곱셈기에 필요한 유한필드 곱셈기와 나눗셈기의 구조를 설명하며, 5장과 6장에서는 새로이 제시한 스칼라 곱셈기의 성능 분석과 결론을 맺는다.

Ⅱ. 구현사례

ECC의 핵심연산인 스칼라 곱셈은 역수연산을 포함한 알고리즘과 역수연산을 포함하지 않은 알고리즘으로 분류할 수 있다. 역수연산을 포함한 알고리즘을 소프트웨어로 구현한 예로 $GF(2^{155})$에서 Diffie-Hellman key exchange를 연산하는데 175MHz성능의 64비트-DEC Alpha 3000를 탑재한 컴퓨터를 이용해 11.5ms의 시간이 걸린다. 하드웨어로 구현한 예로 [3]의 경우 standard basis에서 스칼라 곱셈을 수행하기 위해 가변적인 정수 직렬 곱셈기를 사용하였고, 서버 시스템을 목적으로 어떠한 크기의 필드에 대해서도 곱셈을 연산할 수 있도록 설계되어 있으나 하드웨어 사이즈가 너무 커 휴대용 장치로 적합하지 못하다. [4]는 optimal normal basis를 사용하여 스칼라 곱셈기를 구현하여 제곱연산에서 클럭을 소모하지 않는 장점이 있지만, 역수연산을 위해 $GF(2^m)$상에서 연산 할 경우 $\log_2(m-1)+HW(m-1)-2$(여기서, HW( )는 Hamming weight를 나타낸다)개의 유한체 곱셈연산이 필요하기 때문에 성능 저하의 요인이 된다.

역수 연산을 포함하지 않는 알고리즘으로서 [6]에서는 $GF(p)$인 소수체에서 binary method 방법을 이용해 x 좌표만으로 스칼라 곱셈을 계산하고, 마지막에 y좌표를 얻어낼 수 있는 알고리즘을 제시하였고, 이를 projective 좌표로의 좌표변환을 통해 역수연산을 제거하였다. [7]에서는 [6]에서 제시한 알고리즘을 유한체에 적용해 하드웨어로 구현하였다. [7]은 optimal normal basis에서 스칼라 곱셈기를 구현하였기 때문에 제곱연산에서 클럭을 소모하지 않는 이득이 있지만, 좌표변환 과정에서 3개의 제곱과 16개의 곱셈연산이 추가적으로 발생하기 때문에 높은 성능의 하드웨어 구현이 어렵다.

이들 하드웨어 구현사례 중에서 [3]는 서버 시스템을 목적으로 설계되어졌고, [4][7]는 optimal normal basis에서만 사용되어질 수 있다. 하지만 임베디드 프로세서를 사용하는 환경의 휴대용 보안장치에 필요한 보안코프로세서는 하드웨어 사이즈가 작고 처리속도가 빨라야 하며 임의의 m에 대해 $GF(2^m)$상에서 구현 가능해야 한다.

본 논문에서는 유한체 연산을 위해 normal basis보다 유연성을 가지는 standard basis을 사용하였으며 역수연산을 이용한 방법을 택하였다. 스칼라 곱셈의 전체 성능 향상을 위해 전체 성능의 큰 비중을 차지하는 나눗셈 연산을 빠르게 하기 위해 새로운 유한체 나눗셈 알고리즘을 개발하였다. 유한체 나눗셈 알고리즘을 구현한 사례로는 [2][5][8][9][10]이 있다. 확장 유클리드 알고리즘을 이용하여 유한체 나눗셈을 구현한 [8][9]의 경우 2m번의 반복적인 덧셈연산을 필요로 하며 나눗셈기를 제어하기 위해 $\log_2(m+1)$-bit의 up/down counter와 zero check를 필요로 한다. [5]에서는 유클리드 알고리즘 내에 존

재하는 동적인 데이터 종속관계를 입력 데이터에 무관한 정적인 연산구조로 변환하여 하드웨어 구조를 제안하였다. 그러나 각 연산단계마다 필요한 컨트롤 노드의 복잡도 때문에 정적인 데이터 종속관계를 하드웨어 구현에 충분히 활용하지 못하고 있다. Fermat's theorem을 이용한 [10]의 경우 $\log_2(m-2)+1$개의 곱셈 연산을 하여야 한다. [2]은 유한체 곱셈기와 유한체 제곱기의 전체 하드웨어를 게이트의 조합으로 구성함으로써 최장지연시간(critical path)이 작지만, 하드웨어 사이즈가 커져 휴대용 장치로 적용하기에는 적합하지 않으며, 역수를 계산하는데 m-1개의 유한체 곱셈과 m-2개의 유한체 제곱연산이 필요하다.

따라서 확장 유클리드 알고리즘을 이용한 역수기가 다른 하드웨어보다 수배에서 수십 배 빠르게 역수를 계산할 수 있다. 위의 구현사례들을 고려하여 본 논문에서는 standard basis를 사용하여 새로운 알고리즘의 나눗셈기를 사용한 스칼라 곱셈기를 구현하였다.

Ⅲ. 타원곡선 암호 알고리즘

타원 곡선 암호 시스템은 이산 로그 문제(ECDLP : Elliptic Curve Discrete Lograithm Problem)에 근거하여 1985년 Neal Koblitz와 Victor Miller에 의해서 개발되었다[1]. ECDLP는 타원 곡선에서 임의의 한점 P에 정수 k를 곱한 값이 $Q=kP$일 때, Q와 P를 이용하여 k를 계산하기 어려움을 나타낸다. 타원 곡선 암호 알고리즘의 핵심 연산은 $Q=kP$의 연산을 수행하는 스칼라 곱셈이며, 소수체(Prime Field)인 $GF(p)$와 유한체인 $GF(2^m)$에서 정의된다. 하지만 유한체의 경우 덧셈 연산에서 캐리가 발생하지 않기 때문에 하드웨어로 설계시 소수체보다 구현이 용이하다.

스칼라 곱셈이 이루어지는 유한체 기반의 타원 곡선은 $GF(2^m)$에서 (x,y)인 점들로 구성되어지고, 타원곡선은

$$y^2+xy=x^3+ax^2+b \qquad (1)$$

의 형태를 가진다. 여기서 $a,b\in GF(2^m)$, $b\neq 0$이다. 스칼라 곱셈을 연산하기 위해서는 동일한 두 점의 합을 구하는 두배점 연산(doubling one point)과 서로 다른 두 점의 합을 구하는 점덧셈 연산(adding two point)을 반복적으로 이용하는 Double and Add 알고리즘을 이용할 수 있으며, 알고리즘은 다음과 같다.

```
k = (b_{m-1}b_{m-2} ··· b_1b_0)_2
P = P(x, y) ;
Q = P ;
for i = m-1 downto 0 do                    (2)
   Q = 2Q ;          //doubling one point
   if  b_i = 1 then
      Q = P + Q ; //adding two point
end(Q = kP)
```

알고리즘 (2)에서와 같이 스칼라 곱셈은 스칼라 k 값의 i번째 bit가 1일 경우에는 두배점 연산 후 점 P와 두배점 연산의 결과 값을 점 덧셈 연산하며, 0일 경우 두배점 연산 후 점 덧셈 연산을 거치지 않고 다음 연산을 하게 된다. 따라서 $GF(2^m)$에서 스칼라 곱셈, $Q=kP$를 계산하기 위해서는 최소 m-1번의 두배점 연산에 더하여 k의 Hamming weight만큼의 점 덧셈 연산이 필요하다.

위에서 보듯이 스칼라 곱셈의 핵심연산은 두 점의 합을 구하는 것이며, 타원 곡선상의 임의의 두 점을 $P_1(x_1,y_1)$, $P_2(x_2,y_2)$라하고 출력을 $P_3(x_3,y_3)$라 하면 연산과정은 다음과 같다[1].

```
if P_1 == 0 then P_3 = P_2 and stop
else if P_2 == 0 then P_3 = P_1 and stop
else if x_1 == x_2 then
   if y_1 == y_2 then     //doubling one point
      λ = x_1 + y_1/x_1
      x_3 = λ^2 + λ + a                    (3)
      y_3 = x_1^2 + λx_3 + x_3
   else P_3 = 0
else                          //adding two points
λ = (y_1 + y_2)/(x_1 + x_2)
x_3 = λ^2 + λ + x_1 + x_2 + a
y_3 = λ(x_1 + x_3) + x_2 + y_1
```

식 (3)의 점 덧셈 연산에는 b의 값이 포함되어 있지 않다. 이는 원래 식을 변형하여 b를 소거함으로서 계산상의 편의를 위한 것이다. 또한 스칼라 곱셈에서 두 점의 합을 계산하기 위한 식이 두배점 연산과 점덧셈 연산으로 나누어지는 이유는 유한체에서 덧셈은 Exclusive-OR 연산과 같기 때문에 두배점 연산에서 divide-by-zero의 경우가 발생하기 때문이다. 위 식을 살펴보면 타원 곡선에서 두 점의 합을 계산하기 위해서는 유한체 곱셈, 나눗셈 그리고 덧셈이 필요함을 알 수 있다. 이 가운데 나

눗셈 연산이 가장 많은 시간과 자원을 필요로 하며, 스칼라 곱셈기의 성능을 좌우하게 된다. 본 논문에서는 스칼라 곱셈에서의 유한체 나눗셈 연산을 위하여 새로운 나눗셈 알고리즘을 제안하였으며 자세한 알고리즘에 대해서는 다음 장에서 설명한다.

Ⅳ. 하드웨어 구조

앞장에서 언급한 바와 같이 스칼라 곱셈을 계산하기 위해서는 두배점 연산과 점덧셈 연산이 필요하다. 이 때 두배점 연산을 위해서는 한 번의 나눗셈 연산과 곱셈연산 그리고 두 번의 제곱연산이 필요하다. 또한 점덧셈 연산을 위해서는 한 번의 나눗셈 연산과 곱셈, 제곱연산이 필요하다. 제곱 연산은 곱셈 연산으로 계산이 가능하며 덧셈 연산은 Exclusive-OR 연산이므로 스칼라 곱셈기는 유한체 나눗셈기와 곱셈기 그리고 변수와 결과 값을 저장하기 위한 레지스터와 이들을 제어하기 위한 컨트롤 박스로 구성되어 진다.

1. 유한체 나눗셈기

유한체 나눗셈을 구현하기 위한 대표적인 알고리즘으로는 확장 유클리드 알고리즘과 페르마 이론에 근거한 방법이 있으며 하드웨어로 구현할 경우 확장 유클리드 알고리즘이 페르마 이론보다 빠른 성능을 보인다. 예를 들면 페르마 이론의 경우 $GF(2^m)$상에서 m-1번의 유한체 곱셈과 m-1번의 유한체 제곱연산이 필요하다. 그러나 확장 유클리드 알고리즘의 경우 나눗셈 연산을 위해 2m번의 반복적인 덧셈 연산만이 필요하기 때문에 페르마 방법의 경우보다 성능이 훨씬 빠르다. 따라서 본 논문에서는 $GF(2^m)$에서 확장 유클리드 알고리즘을 기반으로 하여 새로운 나눗셈 알고리즘을 제안하였다.

1.1 확장 유클리드 알고리즘

나눗셈기의 구현을 위한 확장 유클리드 알고리즘은 다음과 같다[5].

```
〈 Equation : B-1 = 1/B mod F 〉
R = F; D = B; U = 1; V = 0;
while (R != 0) begin
  // Polynomial Division                    (4)
  R = R - Q·D ;
  R ↔ D ;
  // Polynomial Multiplication
  V=Q·U+V ;
  V ↔ U ;
end
return V(=B-1)
```

여기에서 ↔는 두 변수의 값을 교환하는 것을 나타낸다. 알고리즘 (4)에서와 같이 확장 유클리드 알고리즘은 역수 연산 알고리즘이다. 하지만 U의 초기값으로 1이 아닌 나눗셈을 위한 피제수 다항식을 대입할 경우 나눗셈 연산이 가능하다. 연산을 위해서는 다항식 나눗셈과 곱셈 후에 두 다항식의 값을 서로 교환하는 과정이 필요하며 각 다항식 나눗셈과 곱셈은 몇 개의 다항식 덧셈 연산으로 이루어져 있다. 하지만 입력 데이터에 따라 필요한 다항식의 덧셈의 수가 달라지기 때문에 동적인 데이터 종속성이 나타나며 이는 하드웨어 구현을 어렵게 만드는 요인이 된다.

1.2 제안하는 새로운 나눗셈 알고리즘

앞에서 설명한 것처럼 확장 유클리드 알고리즘은 입력 데이터에 종속적인 구조를 가지고 있어 연산을 위해 deg(R)-deg(D)(deg( )는 다항식의 차수)번의 가변적인 덧셈 연산수를 필요로 함으로서 하드웨어로 설계하기에 어려움이 있다. 하지만 확장 유클리드 알고리즘을 수행하기 위해 필요한 전체 다항식 덧셈의 연산수는 2m+1로 고정시킬 수 있으며, 이는 하드웨어로 구현이 용이하게 될 수 있음을 의미한다[5]. 따라서 2m+1의 반복적인 연산 동안 나눗셈의 시작과 끝 그리고 다항식의 교환 시점을 결정할 수 있다면 하드웨어로 설계가 가능하다. 본 논문에서는 나눗셈 연산을 제어하기 위해 유한체 나눗셈이 계산 단계인지 아니면 계산을 위한 준비 단계인지를 나타내는 mode신호와 두 다항식의 값이 언제 교환될지를 나타내는 swap신호, 그리고 mode와 swap을 결정하기 위한 추가적인 다항식 G와 H를 이용하여 해결하였다. 나눗셈을 위해 본 논문에서 제안하는 알고리즘은 다음과 같다.

```
〈 Eqaution : A / B mod F 〉
D = F; R = B; U = A; V = 0;
swap = 0;  mode = 1;
G = (0...0)2;   H = (10...0)2;
for i = 1 to 2m-1 begin
  q = rm ;
  R_p = R ;
  R = R⊕(rm ·D);
  swap_p = (hm + ~mode)rm-1 ;
  mode = rm-1 + mode · ~hm ;
  U = {(q·V) ⊕ (hm·T)⊕U} mod F ;
```

```
if(gm == 0)
  U = U·x ; V = V ; T = T ;
  else                                    (5)
  U = 0 ; V = U ; T = V ;
end
gm ↔ hm ;
if(swap == 1) begin
  g(m-1:0) ↔ h(m-1:0) ;
  D = R_p ;
end
R = R·x ; G = G/x ; H = H·x ;
swap = swap_p ;
end.
return V(=A/B)
```

알고리즘 (5)는 확장 유클리드 알고리즘을 비트 단위로 계산하도록 표현하여 하드웨어 구현이 용이하도록 한 것이다. 여기서 ~는 비트의 반전, ⊕는 XOR 연산, 그리고 $A_{[x:y]}$는 A 다항식의 x번째 항에서 y번째 항까지의 비트 열을 나타낸다. 위의 알고리즘은 제어 부분과 유한체 나눗셈 그리고 곱셈의 세 부분으로 나눌 수 있으며 나눗셈을 위하여 $GF(2^m)$에서 2m-1번의 반복적인 계산이 필요하다.

나눗셈을 위한 몫은 R의 최상위 항의 값에 의해 결정된다. R의 최상위 항의 값이 1일 경우에만 R과 D의 교환 동작이 일어나기 때문에 D의 최상위 항의 값은 항상 1이며 D의 값은 연산 과정 중에 변하지 않는다. 따라서 유한체 나눗셈과 곱셈을 위한 몫은 R의 최상위 항의 값이 된다.

연산을 위한 두 다항식이 교환되는 시점은 두개의 컨트롤 신호 mode와 swap 그리고 새롭게 도입한 G와 H의 다항식에 의해 결정되며, mode와 swap은 1과 0으로, G와 H는 $(0...0)_2$와 $(10...0)_2$로 초기화 된다. 이 후 반복적인 연산 동안 g_m와 h_m이 교환되며 swap 신호가 1인 경우에만 $g_{(m-1:0)}$과 $h_{(m-1:0)}$이 교환된다. 연산 과정에서 H의 최상위 항의 값은 G의 최상위 항의 값과 바뀐 후 나눗셈 연산을 위한 준비 단계에서 R의 최상위의 값이 1이 될 때 까지 G의 차수를 낮춤으로 유한체 나눗셈을 위해 필요한 연산의 횟수를 기록하게 된다. R의 최상위 항의 값이 1이 되었을 경우 swap이 1이 되어 G와 H의 값을 교환한 후 H의 차수를 높이면서 H의 최상위 항의 값이 1이 될 때 까지 나눗셈 연산을 하게 된다.

제어신호 mode는 유한체 나눗셈이 계산되고 있는지 아니면 유한체 나눗셈을 위한 준비단계 인지를 나타낸다. 유한체 나눗셈이 끝났을 때 H의 최상위 항의 값이 1이 되며 이때 R의 최상위 항의 값이 1일 경우 다음 연산이 수행되지만 R의 최상위 항의 값이 0일 경우 다음 나눗셈을 위한 준비단계로서 R의 차수를 높이게 된다. 제어신호 swap는 R과 D 다항식의 교환 동작이 일어나는 시점을 결정한다. 유한체 나눗셈이 끝났을 때, R의 최상위 항의 값이 1일 경우에는 그 다음 단계에서 교환 동작이 일어나지만 R의 최상위 항이 1이 아닌 경우에는 R의 차수를 높인 후, 즉 mode 신호가 0인 동안에 R 다항식의 최상위 항이 1이 되면 교환 동작이 일어나야 한다.

앞에서 살펴본 알고리즘 (4)에서 다항식 R의 값이 0이 될 때 A/B mod F는 유한체 곱셈의 우측 변에 있는 V값이 된다. V는 이전 반복 루프에서 U 항등식의 값이므로 마지막 유한체 나눗셈을 계산할 필요없이 U의 값이 결과 값이 된다. 따라서 마지막 유한체 나눗셈은 비트 연산으로 계산하였을 때 최소 2번의 반복 루프를 필요로 하므로, [5]에서는 2m+1번의 반복적인 연산이 필요하지만 본 논문에서는 나눗셈, 즉 A/B mod F를 계산하기 위해 2m-1번의 반복적인 연산이 필요하다.

표 1은 위에서 설명한 알고리즘을 $GF(2^4)$에서 나타난 예로 $x^2+x+1/x^3+x^2+1 \bmod x^4+x+1$을 계산한 것

표 1. 제안한 나눗셈 알고리즘의 예제($x^2+x+1/x^3+x^2+1 \ mod \ x^4+x+1$)

	mode	swap	H	G	R	D	V	U	T
1	1	0	x^4	0	x^3+x^2+1	x^4+x+1	0	x^2+x+1	0
2	1	1	0	x^3	x^4+x^3+x	x^4+x+1	x^2+x+1	0	0
3	1	0	x^4	0	x^4+x	x^4+x^3+x	x^2+x+1	x^3+x^2+x	0
4	1	1	0	x^3	x^4	x^4+x^3+x	x^3+1	0	x^2+x+1
5	1	0	x^4	0	x^4+x^2	x^4	x^3+1	1	x^2+x+1
6	0	0	0	x^3	x^3	x^4	x^3+x^2+x+1	0	x^3+1
7	1	1	0	x^2	x^4	x^4	x^3+x^2+x+1	0	x^3+1
8	1	0	x^3	0	0	x^4	x^3+x^2+x+1	x^3+x^2+1	x^3+1

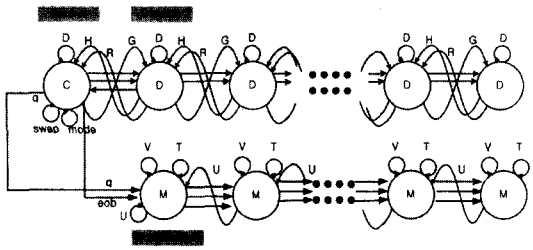


그림 1. 제안하는 나눗셈기의 구조

이다. D의 최상위 항의 값이 항상 1이며 7번째 항에서 V에 나눗셈의 결과 값으로 x^3+x^2+x+1이 나타나는 것을 볼 수 있다. 이 결과는 다음과 같이 나눗셈의 결과 다항식과 제수 다항식을 곱하여 나눗셈에서의 피제수 다항식이 나오는 것을 확인함으로 검증할 수 있다.

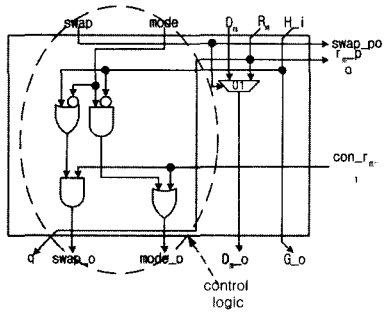
$$
\begin{aligned}
&(x^3+x^2+x+1)(x^3+x^2+1) \bmod x^4+x+1\\
&= x^6+x^5+x^4+x^3+x^5+x^4+x^3+x^2+x^3+x^2+x+1 \bmod x^4+x+1\\
&= x^6+x^3+x+1 \bmod x^4+x+1\\
&= x^2(x+1)+x^3+x+1\\
&= x^3+x^2+x^3+x+1\\
&= x^2+x+1
\end{aligned}
$$

1.3 나눗셈기 하드웨어 구조

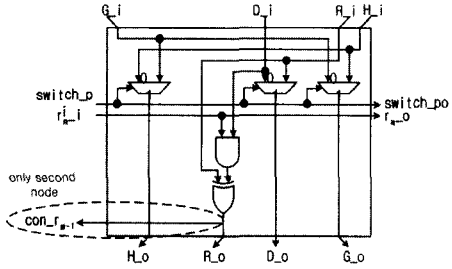
제안하는 알고리즘은 나눗셈과 곱셈 그리고 제어를 위한 세 부분으로 나눌 수 있다. 그림 1은 제안하는 나눗셈 알고리즘의 전체 구조를 보여주고 있다. 그림 1에서 C는 제어 노드이며 D는 나눗셈을 위한 노드, 그리고 M은 곱셈을 위한 노드이다. 제어 로직(C)에서는 연산 과정 중에 다항식을 교환하는데 필요한 제어신호(swap, mode)와 나눗셈과 곱셈을 위한 몫(q)을 결정하여 D와 M에전달한다. D와 M은 제어신호와 몫을 받아 연산을 하며 다항식의 swap 신호에 의해 교환 과정을 거친다.

그림 2는 각각의 세부 로직을 나타내고 있다. 그림 2의 (a)는 C 노드의 세부 로직으로 점선으로 나타낸 부분은 다음 연산을 위한 제어 신호를 결정하는 부분이며 나머지는 나눗셈과 곱셈의 몫을 결정하는 부분이다. 그림 2의 (b)는 D 노드의 세부 로직을 나타내며 다항식 나눗셈 연산을 한다. 그림 2의 (c)는 M 노드의 내부이며 다항식 곱셈 연산을 한다.

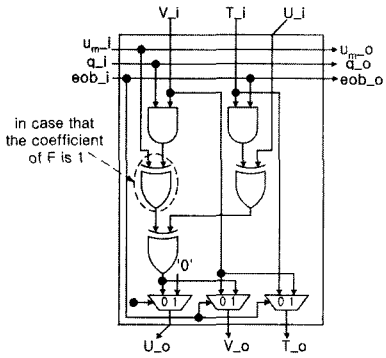
그림 2의 (c)에서 점선으로 나타낸 부분은 모듈러 연산을 할 때 사용되며, irreducible polynomial, F(x)의 계수가 '1'인 PE에만 필요하다. 그림 2의 (a)에서는 식 (2)의 R의 최상위 항의 그림 2. 각 노드의 세부 로직값을 몫으로 결정하며 다음 연산을 위한 제어 신호를 결정한다. 또한 제어 신호 swap이 1일 경우 D와 R의 값을 교환하는 과정 가운데 D와 R의 msb를 교환하게 되며 G와 H의 최상위 항의 값을 교환 한다. 그림 2의 (b)에서



(a)



(b)



(c)

(a)control logic(C-node)
(b)division logic(D-node)
(c)multiplication logic(M-node)

그림 2. 각 노드의 세부 로직

는 식 (2)에서 유한체 나눗셈을 위한 과정 가운데 msb를 제외한 나머지 bit들의 나눗셈 연산을 하게 된다. 이 때 swap의 값이 1일 경우에는 D와 R 그리고, G와 H의 msb를 제외한 나머지 bit들의 값을 mux를 통해 교환하게 된다. 또한 유한체 나눗셈의 준비단계와 계산단계를 구별하기 위해 G의 차수는 낮추며 H의 차수는 높여 주며 다음 연산을 위하여 R의 차수를 높여준다. 유한체 곱셈을 위한 PE는 q와 h_m에 의해 연산되며 모듈러 연산을 한다. 또한 g_m에 의해 U, V 그리고 T의 값을 결정하게 된다.

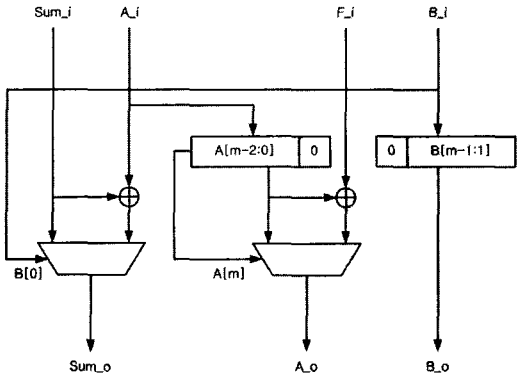


그림 3. 곱셈기 구조

2. 유한체 곱셈기

곱셈기는 A×B의 경우 B의 상수항 계수, 즉 LSB(Last Significand Bit)부터 최고차항 계수, 즉 MSB(Most Significand Bit)까지 순차적으로 검색하여 검색한 비트값이 1일 경우에만 곱셈의 결과와 인수 값을 더하는 일반적인 비트 시리얼 곱셈기를 이용하였다. 곱셈기의 구조는 그림 3과 같다. sum_o은 곱셈의 결과 값을 나타내며 A_o와 B_o는 다음 곱셈을 위한 인수를 타나낸다. B의 LSB값이 1일 경우에는 sum_i 과 A_i의 값을 더한 값이 결과 값이 되며 B의 LSB값이 0일 경우 입력된 sum_i값이 결과 값이 된다. 또한 다음 연산에서 B의 LSB를 검색하기 위하여 B의 값을 오른쪽으로 1-bit Shift하며 A의 경우 차수를 높이기 위하여 왼쪽으로 1-bit Shift한다. 이 때 A의 MSB 값이 1일 경우 F와 XOR 연산을 함으로 모듈러 연산을 하게 된다. GF(2^m)의 곱셈의 경우 m번 반복적으로 연산하게 되면 결과 값을 얻을 수 있다.

3. 스칼라 곱셈기 하드웨어 구조

그림 4는 스칼라 곱셈기의 전체 구조이다. 곱셈기는 유한체 나눗셈과 곱셈을 위한 모듈과 연산을 위한 제어신호를 결정하는 컨트롤 블록으로 구성되어 있다. 컨트롤 블록에서는 k를 m-1bit부터 순차적으로 검색하여 Double and Add알고리즘을 수행하며

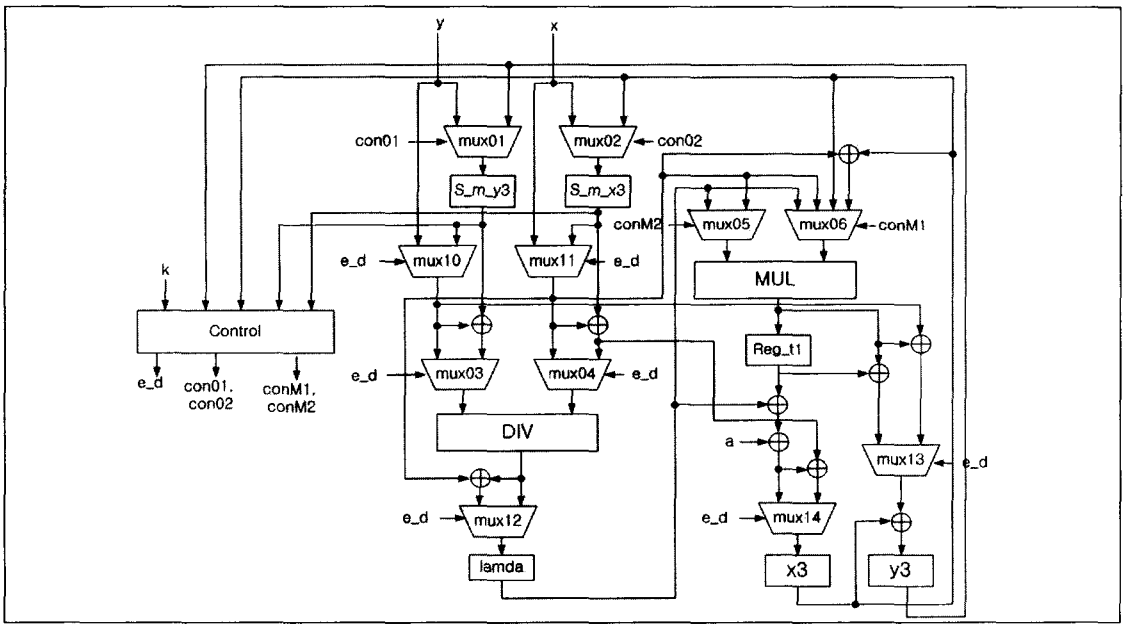


그림 4. 스칼라 곱셈기 전체구조

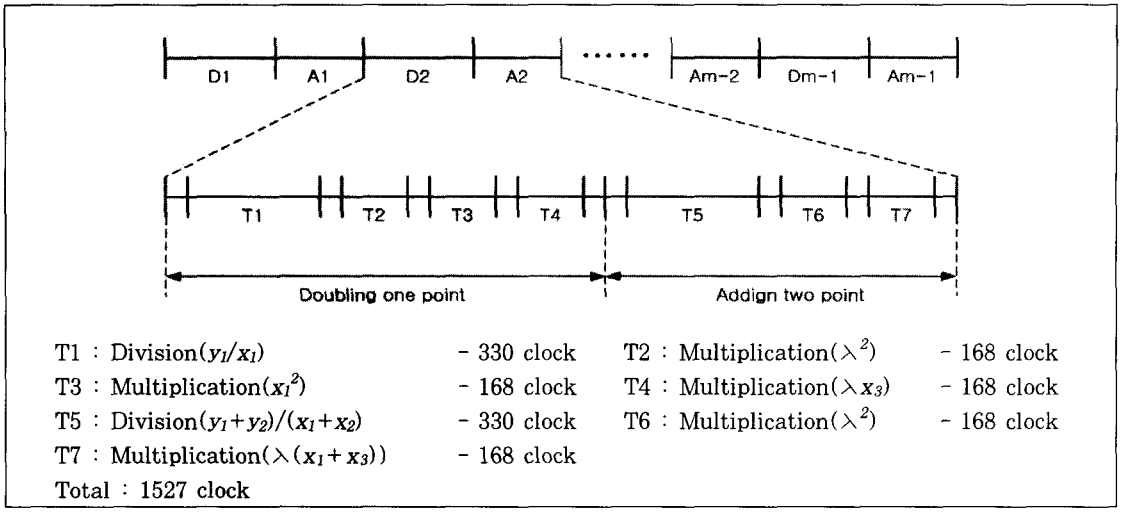


그림 5. 타이밍도

연산에 필요한 제어 신호를 결정한다.

제어신호 e_d는 두배점 연산의 경우 0이며 점덧셈 연산의 경우 1이다. 또한 mux05과 mux06는 동일한 점의 덧셈연산과 다른 점의 덧셈 연산에서의 곱셈 연산에서 x_1^2과 λx_3 그리고 $\lambda(x_1+x_3)$을 구별하기 위해 사용된다. 연산이 끝난 후에 결과값은 x3와 y3 레지스터에 저장되어 다음 연산에서 사용된다.

그림 5는 스칼라 곱셈이 이루어지는 전체 과정을 나타내는 타이밍도를 보여주고 있다. 타이밍도에서 T1, T2, T3, T4는 두배점 연산하는 부분을 나타내며 T5, T6, T7은 점덧셈 연산하는 부분이다. 타이밍도를 살펴보면 두배점 연산을 위하여 한번의 나눗셈 연산과 3번의 곱셈 연산이 필요하며 점덧셈 연산을 위해서는 한번의 나눗셈과 두 번의 곱셈 연산이 필요함을 알 수 있다. 나눗셈을 위해서는 330 clock이 필요하며 곱셈을 위해서는 168 clock이 필요하다. 또한 T1-T7을 제외한 나머지 빈 칸들에서 각 연산의 중간 과정으로 데이터를 저장하고 필요한 데이터를 선택하기 위한 클럭을 소모하고 있는 것을 볼 수 있다.

식 2에서와 같이 스칼라 곱셈에서의 나눗셈 연산은 λ를 계산하기 위해 사용되며, 이 때 두배점 연산과 점덧셈 연산에서의 나눗셈을 위한 연산자와 피연산자가 서로 다르다. mux01와 mux02에서 각각의 나눗셈 연산을 위해 요구되는 연산자와 피연산자를 출력하게 되며 mux03와 mux04에서 제어신호 e_d에 의해 필요한 연산자와 피연산자를 선택하여 나눗셈 연산을 하게 된다.

두배점 연산에서 x_3는 $\lambda^2+\lambda+a$이며 점덧셈 연산에서 x_3는 $\lambda^2+\lambda+x_1+x_2+a$이다. 두 연산 과정 가운데 $\lambda^2+\lambda+a$는 공통으로 포함되기 때문에 $\lambda^2+\lambda+a$을 연산 한 후 점덧셈 연산일 경우에만 x_1+x_2값을 더하여 x_3의 레지스터에 저장한다. y_3의 경우 점덧셈 연산에서는 한번의 곱셈 연산 ($\lambda(x_1+x_3)$)이후에 x_3와 y_1을 더한 후 y_3의 레지스터에 저장하지만 두배점 덧셈 연산의 경우 x_1^2와 λx_3의 두 번의 곱셈 연산 후에 x_3의 값을 더하여 y_3의 레지스터에 저장한다. 이 때 x_1^2의 값은 reg_t1에 저장되며 λx_3의 연산 후에는 결과 값을 바로 덧셈 연산하여 y_3의 값을 계산한다.

V. 검증 및 성능 분석

표 2는 지금까지 발표된 대표적인 하드웨어 구현 예들과 본 논문에서 제시한 스칼라 곱셈기에 대해 필요한 연산의 수를 나타내고 있으며 표 3에서는 스칼라 곱셈에 필요한 전체 클럭 수를 나타내고 있다. 표 2에서와 같이 [7]은 식(2)의 반복 루프에서 역수계산을 제거하기 위해 좌표 변환함으로써 더 많은 곱셈과 제곱연산이 발생되어졌고, 또한 반복 루프를 끝낸 후 추가적인 좌표 역 변환 과정이 필요하다. 본 논문과 [3], [4]에서는 반복 루프에서 역수 계산을 포함하고 있으며, 이를 계산하는데 필요한 시간비용은 곱셈기만을 이용하여 계산할 경우에 전체 시스템에 큰 부하를 가져온다. 왜냐하면

표 3을 보면 알 수 있듯이 역수 연산을 위하여 [7]은 $24m$번의 연산이 필요하며, [4]는 $\log_2(m-1)+HW(m-1)-2$의 연산이 필요하다. [3]의 경우 역수 연산에 대한 언급이 없어 페르마 이론을 이용하였을 경우로 가정하였으며, 이 경우 $2\lceil m/288\rceil \lceil m/8\rceil (m-1)$의 연산이 필요하다. 따라서 나눗셈을 $2m-1$의 반복적인 연산 시간에 계산할 수 있는 본 논문에서 제안한 알고리즘을 이용한 나눗셈기를 사용할 경우 다른 구조에 비해 빠른 연산을 수행할 수 있다. 이는 표 3에서의 스칼라 곱셈에 필요한 전체 클럭 수를 보면 알 수 있다.

표 2. 각 구현사례에 대한 연산의 수

구현사례	Basis	반복루프에서의 연산수			추가연산	
		곱셈	제곱	역수	곱셈	역수
[7]	ONB	20	6	0	6	1
[4]	ONB	4	3	2	0	0
[3]	SB	4	3	2	0	0
본논문	SB	2	3	2	0	0

(ONB : optimal normal basis,
SB : standard basis)

표 4는 각 구현사례들의 성능을 비교한 것으로 [3]의 경우 표 3에서 가장 많은 클럭 수를 필요로 했던 것과는 달리 본 논문과 동등한 성능을 가진다. 표 3에서는 [3]의 역수 연산을 페르마 이론을 사용하였을 경우로 가정하였기 때문에 [3]에서 제시한 성능과 표 3에서의 성능은 역수 연산 알고리즘에 의해 다르게 측정되었다. [3]은 서버 시스템을 목적으로 어느 크기의 필드에서도 스칼라 곱셈을 할 수 있도록 설계하였기 때문에 하드웨어 리소스가 본 논문에 비하여 크다. 또한 [7] [4]는 optimal normal basis에서 스칼라 곱셈기를 구현하였다. [7]은 서브필드가 존재하는 필드를 선택함으로써 성능과 사이즈에서의 장점이 있지만 특정 필드에만 국한된다는 단점이 있으며, [4]는 normal basis의 계산상의 장점을 이용하였지만 이는 standard basis가 아닐 경우 어려운 단점이 있다.

본 논문에서 제안한 나눗셈기는 $GF(2^m)$에서 $2m-1$의 반복적인 연산 과정이 필요하며 곱셈기는 일반적인 비트 시리얼 곱셈기를 사용하였기 때문에 m번의 반복적인 연산과정이 필요하다. 식 (2)의 Dobule-and-Add알고리즘을 사용하였을 경우 $Q=kP$에서 k의 모든 bit 값이 1이라고 한다면 $m-1$번의 두배점 연산과 점 덧셈 연산이 필요하다. 식(2)에서와 같이 점 덧셈 연산에서 두배점 연산의

표 3. 각 구현사례의 스칼라 곱셈에서 필요한 전체 클럭수

구현사례	스칼라 곱셈에 필요한 전체 클럭수	하드웨어 구성	$GF(2^m)$상에서 연산(클럭수)
[7]	$26m^2+2m$	곱셈기	곱셈(m) 역수($24m$)
[4]	$(7m+2(\log_2(m-1)+HW(m-1)-2)(m-1))(m-1)$	곱셈기	곱셈(m) 역수($\log_2(m-1)+HW(m-1)-2$)
[3]	$(4m^2-m-3)\lceil m/288\rceil \lceil m/8\rceil$ *	82×4 곱셈기	곱셈($\lceil m/288\rceil \lceil m/8\rceil$) 역수($2\lceil m/288\rceil \lceil m/8\rceil (m-1)$)*
본 논문	(m-1)(5(m)+2(2m-1))	곱셈기 나눗셈기	곱셈(m) 나눗셈(2m-1)

*[3]의 경우 역수 연산을 Fermat 이론을 이용하였다고 가정한다.
$\lceil x\rceil$는 x보다 크거나 같은 정수 가운데 가장 작은 값을 의미한다.

표 4. 각 구현사례의 성능 분석

구현사례	field	device	gate count	performance
[7]	$GF((2^5)^{31})$	GF155MIC	11k gates	19kbps
[4]	$GF(2^{53})$	Xilinx FPGA XC4044XL	CLB usage 84%	23kbps
[3]	$GF(2^{163})$	0.25um ASIC	165k gates	148kbps
본 논문	$GF(2^{163})$	0.18um ASIC	60k gates	148kps

경우 1번의 나눗셈과 3번의 곱셈 연산이 필요하며 점덧셈 연산의 경우 1번의 나눗셈과 2번의 곱셈 연산이 필요하다. 최악의 경우로서 연산의 모든 경우가 두배점 연산이라고 가정하였을 경우 총 곱셈과 나눗셈만을 위해 $(m-1)(5(m)+2(2m-1))$의 클럭이 요구된다.

스칼라 곱셈기의 설계는 Verilog을 사용하여 설계하였으며 Altera의 SoC FPGA인 Excalliber에서 검증하였다. 또한 ASIC을 위하여 삼성전자의 0.18um 공정의 std130 표준 셀 라이브러리를 사용하여 Synopsys Design Complie 에서 합성하였다. 이 때 60,000 게이트의 하드웨어 사이즈와 250MHz의 최대 동작 주파수를 나타내었다. 이 경우의 성능은 약 148kbps로 163-bit 프레임당 1.1ms 시간이 걸린다. 따라서 본 논문에서 제안한 나눗셈기를 적용한 스칼라 곱셈기는 기존의 스칼라 곱셈기에 비하여 성능 및 하드웨어 사이즈에서 유리하다. 또한 제안한 스칼라 곱셈기의 중요한 특징으로는 유한체 나눗셈기가 규칙적인 구조를 가지고 있어 GF(2^m)의 m의 크기가 변하더라도 쉽게 확장 혹은 축소할 수 있으며, m의 크기에 독립적인 2-bit의 제어신호만을 필요로 하기 때문에 입력신호의 필드가 커질수록 본 논문의 유한체 나눗셈기가 기존의 나눗셈기에 비하여 매우 유리하다.

VI. 결 론

스마트카드처럼 저가의 임베디드 프로세서를 사용하는 휴대용 보안기기에는 보안 코프로세서가 필수적이다. 본 논문에서는 타원곡선 암호알고리즘을 이용한 보안 코프로세서를 위해 새로운 구조의 스칼라 곱셈기를 구현하였다. 본 논문의 스칼라 곱셈기는 확장 유클리드 알고리즘에 기초한 새로운 유한체 나눗셈기와 비트-시리얼 유한체 곱셈기로 구성되어 있다. 스칼라 곱셈 연산에서 가장 많은 시간과 자원을 필요로 하는 나눗셈 연산을 위해 새로운 알고리즘을 제안하였으며 이를 이용하여 일차원 어레이 형태의 나눗셈기를 구현하였다. 제안한 알고리즘은 나눗셈 연산을 위해 2m-1의 반복적인 덧셈 연산을 필요로 한다. 또한 유한체 나눗셈과 곱셈의 제어를 위해 도입한 G와 H의 새로운 유한체 원소를 사용함으로써 입력 데이터에 독립적이며 2bit의 고정된 제어 신호만으로 제어할 수 있다.

구현한 스칼라 곱셈기를 삼성전자 0.18um std130을 이용하여 예측한 성능은 최대 250MHz 동작이며 이 경우 성능은 약 148kbps로 163bit 프레임당 1.1ms걸린다. 이러한 성능의 스칼라 곱셈기는 휴대용 보안장치의 보안 코프로세서로 사용하기에 적합하다. 또한 본 논문에서 설계한 나눗셈기는 ECC를 위한 스칼라 곱셈기 뿐만 아니라 RS(Reed-Solomon) Code와 같은 에러 정정 코드의 복호화 과정에서도 효과적으로 사용될 수 있으며, 하드웨어로 설계시 규칙적인 구조를 가지고 있어 GF(2^m)의 m의 크기가 변하더라도 쉽게 확장 혹은 축소할 수 있다. 현재 휴대용 보안장치의 코프로세서로 사용하기 위해 동작속도를 낮추고 유한체 곱셈의 계산 수를 더 줄이는 방법에 대한 추가 연구를 진행중이다.

참 고 문 헌

[1] Certicom Whitepaper, "The Elliptic Curve Cryptosystem for Smart Cards," http://www.certicom.com, May, 1998

[2] A. V. Dinh, R. J. Palmer, R. J. Bolton and R. Mason, "A low latency architecture for computing multiplicative inverses and division in GF(2^m)," Electrical and Computer Engineering, IEEE 2000 Canadian Conference, Vol. 1, pp. 43-47, 2000

[3] S. Okada, N. Torii, K. Itoh, and M. Takenaka, "Implementation of Elliptic Curve Cryptographic Coprocessor over GF(2^m) on an FPGA," Workshop on Cryptographic Hardware and Embedded Systems(CHES), pp. 25-40, August 2000.

[4] L. Gao, S. Shrivastava, and G. E. Sovelman, "Elliptic Curve Scalar Multiplier Design Using FPGAs," Workshop on Cryptographic Hardware and Embedded Systems(CHES), pp. 257-268, August 1999.

[5] Y. Jeong and W. Burleson, "VLSI Array Synthesis for Polynomial GCD Computation and Application to Finite Field Division," IEEE Transaction on Circuits and Systems, pp. 891-897, Dec. 1994.

[6] P. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," Mathematics of Computation, Vol. 48, pp. 243-264, 1987.

[7] G, B. Agnew, R. C. Mullin and S. A. Vanstone, "Implementation of Elliptic Curve Cryptosystems over GF(2^{155})," IEEE Journal on Selected Areas in Communication, Vol. 11, No. 5, pp. 804-813, 1993.

[8] H. Brunner, A. Curiger and M. Hofstetter, "On computing multiplicative inverses in GF(2m)," IEEE Transactins on computers, Vol. 42, No. 8, pp. 1010-1015, Aug. 1993.

[9] J. H. Guo and C. L. Wang, "Hardware-efficient systolic architecture for inversion and division in GF(2^m)," IEE Proceedings computers and digital techniques, Vol. 145, No. 4, July 1998.

[10] S. Morioka andY. Katayama, "O($\log_2{}^m$) Iterative Algorithm for Multiplicative inversion in GF(2^m)," IEEE proc. of International Symposium on Information Theory, 2000 Proceedings, IEEE. pp. 449, 2000.

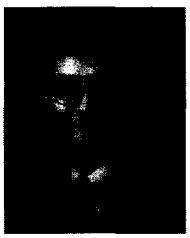
김 의 석(Eui-seok Kim) 준회원

2003년 2월 : 광운대학교 전자공학부 졸업

2003년 3월~현재 : 광운대학교 전자통신공학과 석사과정

<관심분야> 보안, 통신 시스템, ASIC 설계

정 용 진(Yong-jin Jeong) 정회원



1983년 2월 : 서울대학교 제어 계측공학과 졸업

1983년 3월~1989 9월 : 한국전자통신연구원

1991년 5월 : 미국 UMASS 전자전산공학과 석사

1995년 2월 : 미국 UMASS 전자전산공학과 박사

1995년 4월~1999년 2월 : 삼성전자 반도체 수석연구원

1999년 3월~현재 : 광운대학교 전자공학부 부교수

<관심분야> 컴퓨터 연산 알고리즘, ASIC 설계, 무선 통신, 정보보호