컨텐츠 보호를 위한 DTCP용 타원곡선 암호(ECC) 연산기의 구현

준회원 김 의 석*, 정회원 정 용 진**

Design of a ECC arithmetic engine for Digital Transmission Contents Protection (DTCP)

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

요 약

본 논문에서는 디지털 컨텐츠 보호를 위해 표준으로 제정된 DTCP(Digital Transmission Contents Protection)용 타원 곡선 암호(ECC) 연산기의 구현에 대해 기술한다. 기존의 시스템이 유한체 $GF(2^m)$를 사용하는 것과는 달리 DTCP에서는 소수체인 GF(p)에서 타원 곡선을 정의하여 인증 및 키 교환을 위해 ECC 암호 알고리즘을 사용하고 있다. 본 논문에서는 ECC 알고리즘의 핵심 연산인 GF(p) 상에서의 스칼라 곱셈 연산기를 구현하였으며, 이 중 가장 많은 시간과 자원을 필요로 하는 나눗셈 연산을 제거하기 위하여 투영 좌표 변환 방법을 이용하였다. 또한, 효율적인 모듈러 곱셈 연산을 위하여 몽고메리 알고리즘을 이용하였으며, 곱셈기의 처리 속도를 빠르게 하기 위해 CSA(Carry Save Adder)와 4-레벨의 CLA(Carry Lookahead Adder)를 사용하였다. 본 논문에서 설계한 스칼라 곱셈기는 삼성전자 *0.18* um CMOS 라이브러리를 이용하여 합성하였을 경우 *64,539* 게이트의 크기에 최대 98 MHz까지 동작이 가능하며 이 때 데이터 처리속도는 *29.6* kbps로 *160*-bit 프레임당 *5.4* ms 걸린다. 본 성능은 실시간 환경에서 DTCP를 위한 디지털 서명, 암호화 및 복호화, 그리고 키 교환 등에 효율적으로 적용될 수 있다.

Key Words : ECC, DTCP, Scalar Multiplier, Modular Multiplier, Montgomery Multiplier

ABSTRACT

In this paper, we implemented an Elliptic Curve Cryptography(ECC) processor for Digital Transmission Contents Protection (DTCP), which is a standard for protecting various digital contents in the network. Unlikely to other applications, DTCP uses ECC algorithm which is defined over GF(p), where p is a 160-bit prime integer. The core arithmetic operation of ECC is a scalar multiplication, and it involves large amount of very long integer modular multiplications and additions. In this paper, the modular multiplier was designed using the well-known Montgomery algorithm which was implemented with CSA(Carry-save Adder) and 4-level CLA(Carry-lookahead Adder). Our new ECC processor has been synthesized using Samsung *0.18* um CMOS standard cell library, and the maximum operation frequency was estimated 98 MHz, with the size about 65,000 gates. The resulting performance was *29.6* kbps, that is, it took *5.4* msec to process a *160*-bit data frame. We assure that this performance is enough to be used for digital signature, encryption and decryption, and key exchanges in real time environments.

* 세인시스템(check95@explore.kw.ac.kr), ** 광운대학교 부교수(yjjeong@daisy.kw.ac.kr)
논문번호 : KICS2004-10-252 , 접수일자 : 2004년 10월 29일
※본 연구는 광운대학교 산학연컨소시엄 및 IDEC/SIPAC의 지원으로 이루어졌습니다.

Ⅰ. 서 론

정보통신 기술의 급격한 발전은 전자상거래 및 전자 문서 교환과 같은 사용자간의 높은 수준의 보안과 신뢰성을 요구하는 다양한 서비스를 제공하고 있다. 그러나 인터넷 환경에서는 디지털 컨텐츠의 공유, 복사 및 유포가 가능하기 때문에 저작권 침해와 같은 많은 문제가 발생하고 있다. 이를 해결하기 위해 Hitachi, Intel, Matsushita, Sony, Toshiba 5개 회사가 모여 디지털 컨텐츠 보호를 위한 표준으로 IEEE1394 인터페이스를 기반으로 DTCP(Digital Transmission Content Protection)를 제정하였다. DTCP는 음성/영상 컨텐츠 보호를 위하여 공개키 기반의 인증 시스템을 이용하여 불법 복제 및 위조를 방지하기 위함을 목적으로 하고 있다.

DTCP는 그 기능이 AKE(Authentication and Key exchange), CE(Content Encryption), CCI (Copy Control Information), SR(System renewability)의 4가지 주요 구성으로 이루어져 있다. 이 가운데 AKE는 Full Authentication과 Restrict Authentication으로 나누어진다. Full Authentication은 DTCP를 지원하는 대부분의 디지털 컨텐츠 장비에서 사용되며, 인증 및 키 교환 알고리즘으로 소수체 GF(p) 기반의 EC-DSA와 EC-DH를 사용한다[1]. 시스템 자원이 제한된 시스템에서는 연산이 복잡한 EC-DSA와 EC-DH가 아닌 미리 정의된 키 벡터를 이용하는 Restrict Authentication을 사용한다.

DTCP에서 전자 인증 및 키 교환에 사용되는 타원곡선 암호(ECC) 알고리즘은 RSA(Rivest-Shamir-Adelman)와 함께 현재 사용되고 있는 대표적인 공개키 암호 알고리즘이다. 타원 곡선 암호 알고리즘은 RSA에 비해 작은 키 사이즈를 가지고도 높은 안정성을 갖는 장점이 있다. 예를 들어 1024비트의 키 사이즈를 가지는 RSA와 대등한 안정성을 가지기 위해 ECC는 160 비트의 키 사이즈만을 필요로 한다. 그러므로 ECC와 RSA의 키 사이즈 비율이 1:7 정도이며 키 사이즈가 커질수록 동등 레벨의 안정성을 유지하기 위한 두 알고리즘의 키 사이즈 비율은 더욱 커진다. 암호학적으로 안전하게 요구되는 복잡도는 시대를 거듭할수록 높아지기 때문에, 매우 큰 정수 유한체에서 연산이 수행되는 RSA 암호 알고리즘보다는 비교적 작은 타원 곡선 상에서 연산이 수행되는 타원 곡선 암호 알고리즘이 확장성에서 유리하고 미래 지향적이라 할 수 있다.
타원 곡선 암호 알고리즘은 유한체인 $GF(2^m)$상에서의 스칼라 곱셈 알고리즘이 구현상으로 용이하기 때문에 많이 쓰인다. 유한체에 기초한 알고리즘의 경우 하드웨어로 설계 시 덧셈 연산이 XOR 연산과 같기 때문에 캐리가 발생하지 않아 설계가 쉬우며 빠르게 연산 할 수 있기 때문이다. 하지만 DTCP의 경우 소수체인 GF(p)에서의 스칼라 곱셈 알고리즘을 사용하며[1], 유한체에서와는 달리 덧셈 연산에서 캐리가 발생하여 모듈러 곱셈 및 나눗셈 연산에 많은 시간을 필요로 한다. 그러므로 소수체 GF(p)에서의 스칼라 곱셈 연산은 효율적인 모듈러 곱셈기와 나눗셈기의 설계가 전체 성능을 좌우하게 된다.

본 논문에서는 DTCP에서 AKE 과정 가운데 Full Authentication을 위한 GF(p)에서의 스칼라 곱셈 연산기를 설계하였다. 타원곡선에서의 연산 과정 가운데 나눗셈 연산을 제거하기 위하여 투영 좌표 변환을 이용하였으며, 빠른 스칼라 곱셈 연산을 위하여 몽고메리 모듈러 곱셈 알고리즘을 이용하였다. 효율적인 몽고메리 모듈러 곱셈 연산을 위하여 전처리 변환 및 후처리 변환을 이용하였으며, Affine 좌표 변환을 위한 역수 연산은 모듈러 곱셈기를 재사용할 수 있는 페르마 이론을 이용하였다. 몽고메리 모듈러 곱셈기에 사용되는 덧셈기는 CSA(Carry-save Adder)와 4-level CLA(Carry-lookahead Adder)를 사용하였다.

본 논문의 구성은 다음과 같다. 2장에서는 타원곡선 암호 알고리즘에 대하여 살펴보며, 3장에서는 몽고메리 모듈러 곱셈 알고리즘에 대하여 살펴본다. 4장에서는 스칼라 곱셈을 위한 세부 연산을 살펴보며, 5장에서는 하드웨어 구현을 위한 구조를 살펴본다. 6장에서는 설계한 스칼라 곱셈기의 성능 분석을 하며 7장에서 결론을 맺는다.

Ⅱ. 타원 곡선 암호 알고리즘

타원곡선 암호 알고리즘은 타원곡선 이산로그 문제(ECDLP : Elliptic Curve Discrete Logarithm Problem)에 근간을 두고 있다. ECDLP는 타원 곡선 상의 임의의 한 점 P에 정수 k를 곱한 값이 $Q=kP$일 때, 점 Q와 P를 알고 있더라도 정수 k를 계산하기 어려움을 의미한다. 또한 모든 ECC 응용 시스템에서 필요로 하는 핵심 연산은 스칼라 곱셈, 즉 $Q=kP$를 구하는 것이다.

소수체 GF(p)상에서의 타원곡선은 (x, y)인 점들로 구성되어지고, $y^2+xy=x^3+ax+b$ 의 형태를 가진다[3]. 여기서 a, $b \in GF(p)$이며 $4a^3+27b^2 \neq 0$ (mod

p) 이다. 효율적인 스칼라 곱셈을 연산하기 위하여 Double and Add 알고리즘이 사용되며, 알고리즘은 다음과 같다[2].

Algorithm 1 : Double and Add

intput : $k=b_{m-1}2^{m-1}+b_{m-2}2^{m-2}+\cdots+b_1 2+b_0,\ b_{m-1}=1$
$P(x, y)$, where $b_i \in \{0,1\}$, $x,y \in GF(p)$
output : Q = kP

```
Q = P ;
for i = m-2 downto 0 do
    Q = 2Q ;            //doubling one point
  if  bi = 1 then
    Q = P + Q ;         //adding two point
  end if
end for
```

알고리즘 1에서와 같이 Double and Add 알고리즘을 이용 할 경우 스칼라 곱셈은 동일한 두 점의 합을 구하는 두배점 연산(doubling one point)과 서로 다른 두 점의 합을 구하는 점덧셈 연산(adding two points)을 반복적으로 연산한다. 타원 곡선상의 임의의 두 점을 $P_1(x_1, y_1)$, $P_2(x_2, y_2)$라 하였을 때 두배점 연산과 점덧셈 연산 과정은 다음과 같이 정의된다[3][4].

Algorithm 2 : Point Addition (Affine Coordinate)

input : $P_1(x_1, y_1)$, $P_2(x_2, y_2)$
output : $P_3(x_3, y_3) = P_1 + P_2$

```
if P1 = 0 then P3 = P2 and stop
if P2 = 0 then P3 = P1 and stop
if x1 == x2 then
  if y1 == y2 then          //doubling one point
     λ=(y1-y2)/(x1-x2) mod p
  else
     P3 = 0
else                        //adding two points
     λ=(3x2^2+a)/(2y2) mod p

x3 = λ^2-x1-x2 mod p
y3 = (x2-x3)λ-y2 mod p
```

알고리즘 2와 같이 타원곡선($y^2+xy=x^3+ax^2+b$)상의 두 점의 합을 연산하는 과정에는 b 값이 포함되어 있지 않은데 이는 계산상의 편의를 위하여 식을 변형하여 b를 소거한 것이다. 위 알고리즘을 살펴보면 타원 곡선에서 두 점의 합을 계산하기 위해서는 곱셈, 나눗셈 그리고 덧셈 연산이 필요함을 알 수 있다. 이 가운데 나눗셈 연산이 가장 많은 시간과 자원을 필요로 하며, 스칼라 곱셈기의 성능을 좌우하게 된다. 그러나 투영 좌표 변환(Projective Coordinate Conversion)을 이용하면 나눗셈 연산 제거가 가능하며, 이를 이용한 연산 과정이 IEEE P1363a 표준 문서에 다음과 같이 정의되어 있다[3].

Algorithm 3 : Point Addition (Projective Coordinate)

intput : $P_1(X_1,Y_1,Z_1)$, $P_2(X_2,Y_2,Z_2)$
output : $P_3(X_3,Y_3,Z_3) = P_1+P_2$

```
if X1 == X2 then
  if Y1 == Y2 then          //doubling one point
    M = 3X1^2 + aZ1^4 ;
    Z2 = 2Y1Z1 ;
    S = 4X1Y1^2 ;
    X2 = M^2 - 2S ;
    T = 8Y1^4 ;
    Y2 = M (S - X2) - T ;
  else
    P3 = 0 ;

else                        //adding two points
    U0 = X0Z1^2 ;
    U1 = X1Z0^2 ;
    S0 = Y0Z1^3 ;
    S1 = Y1Z0^3 ;
    W = U0 - U1 ;
    R = S0 - S1 ;
    T = U0 + U1 ;
    M = S0 + S1 ;
    Z2 = Z0Z1W ;
    X2 = R^2 - TW^2 ;
    V = TW^2 - 2X2 ;
    2Y2 = VR - MW^3 ;
```

위 알고리즘의 전체 연산은 알고리즘 2에서와 같이 모듈러 p로 계산된다. 알고리즘 3은 알고리즘 2와 비교하여 나눗셈 연산이 제거되었지만 보다 많은 곱셈 연산이 사용된다. 따라서 모듈러 곱셈 연산이 스칼라 곱셈의 많은 부분을 차지하게 되며 성능 향상을 위하여 효율적인 모듈러 곱셈기의 설계를 필요로 한다. 본 논문에서는 이를 위해 몽고메리 모듈러 곱셈 알고리즘을 사용하였으며 자세한 알고리즘은 다음 장에서 설명한다.

Ⅲ. 몽고메리 모듈러 곱셈 알고리즘

모듈러 곱셈 연산 방법에는 곱셈을 먼저 수행한 후 모듈러 연산을 하는 방법과 곱셈 중에 확장되는 비트를 제한시키는 방법이 있다. 하드웨어 구현 시 전자는 곱셈의 중간 결과 값의 비트 사이즈가 승수

와 피승수의 비트 사이즈 합의 크기로 커지기 때문에 모듈러 연산을 위한 연산기의 사이즈가 커지게 된다. 따라서 승수와 피승수의 비트 사이즈의 연산기만을 필요로 하는 후자의 방법이 주로 사용되고 있으며 본 논문에서도 이 방법을 적용하였다.

곱셈 연산 중 확장되는 비트를 검색하여 제거하기 위한 방법에는, 곱셈 연산의 중간 값(partial product)에 대해 MSB(Most Significant Bit) 방향으로 커지는 비트를 보고 적당한 계수의 배수를 뺌으로써 새로 늘어난 비트를 제거하는 방법으로 전체적인 비트수를 감소, 유지시키는 MSB우선 방식(e.g. JB 알고리즘)과 LSB(Least Significant Bit)부분을 '*0*'으로 만드는 적당한 계수의 배수를 더한 다음 오른쪽으로 쉬프트하는 LSB 우선 방식 (e.g. 몽고메리 알고리즘)이 있다[2]. 몽고메리 알고리즘의 경우는 원하는 결과값을 위해서는 전처리 및 후처리 과정이 필요하다는 단점이 있지만 반복된 곱셈을 실행하기 위해서는 이러한 전처리 및 후처리 과정을 제거시킬 수 있다는 특징이 있다. 본 논문에서는 하드웨어 구현상의 특징을 고려하여 LSB 우선 방식인 몽고메리 알고리즘을 사용하였으며, 연산과정은 다음과 같이 정의된다.[5]

Algorithm 4 : Montgomery Product Algorithm

```
input : x, y, P
output : S = Monpro(x, y) = xy2^-k mod P
         ,where k is the bit length of x, y, P

S = 0 ;
for i from 0 to k-1 do
   m_i = S_0 ^ x_i y_0 ;
   S = (S + x_i y + m_i P) / 2 ;
end
```

위에서 알 수 있듯이 몽고메리 알고리즘은 y의 모든 비트에 대해 LSB에서부터 MSB까지 검색하여 덧셈 및 쉬프트 연산을 행하는 것이다. 알고리즘의 결과는 결과 값 xy에 대해 2^{-k}이 곱해진 값이다. 즉, x와 y가 k-bit 일 경우 k번의 연산을 하며, k번 오른쪽 쉬프트 연산이 이루어지기 때문에 결과 값 xy에 2^{-k} 만큼 곱해진 값이 출력된다. 따라서 연산 후, 올바른 값을 얻기 위하여 다음과 같은 후처리 과정이 필요하다.

$$S = S \times 2^{k} \bmod P \quad (1)$$

그러나 스칼라 곱셈 연산 가운데 모든 곱셈 연산 후에 위 식과 같은 후처리 과정을 해준다면 비효율적이다. 하지만 ECC의 스칼라 곱셈은 기본 좌표 값과 기본 좌표 값에 의해 연산되는 값을 이용하여 다음 연산이 이루어지기 때문에 ECC의 기본 좌표 값에 2^{2k}으로 미리 몽고메리 알고리즘을 이용하여 곱하는 전처리 과정을 행한 후 계산함으로 각각의 곱셈연산에 대한 후처리 과정을 생략할 수 있다. 모든 연산 후 스칼라 곱셈의 올바른 값을 도출하기 위한 후처리 과정은 전처리 과정에서 2^{2k}을 곱하였기 때문에 식 (1)에서와 같이 2^{k}이 아닌 1을 곱하면 된다. $S=A\times B \bmod P$를 연산할 때 몽고메리 전처리 과정 및 후처리 과정을 포함하는 전체 모듈러 곱셈 연산 과정을 나타내면 다음과 같다.

Pre-Processing

$$A^* = \text{Monpro}(A,\ 2^{2k}) = A\times 2^{2k}\times 2^{-k} \bmod P = A\times 2^{k} \bmod P$$

$$B^* = \text{Monpro}(B,\ 2^{2k}) = B\times 2^{2k}\times 2^{-k} \bmod P = B\times 2^{k} \bmod P$$

Main -Processing (2)

$$S^* = \text{Monpro}(A^*,\ B^*) = A\times 2^{k}\times B\times 2^{k}\times 2^{-k} \bmod P = A\times B\times 2^{k} \bmod P$$

Post -Processing

$$S = \text{Monpro}(S^*,\ 1) = A\times B\times 2^{k}\times 2^{-k} = A\times B \bmod P$$

Ⅳ. 스칼라 곱셈 알고리즘

알고리즘 3의 투영 좌표 변환과 알고리즘 4의 몽고메리 알고리즘을 이용한 스칼라 곱셈 연산을 위해서는 다음과 같은 과정이 필요하다.

(a) 투영 좌표 변환
(Affine to Projective Coordinates Converseion)
(b) 몽고메리 모듈러 곱셈 전처리 변환
(c) 타원 곡선 스칼라 곱셈 연산
(d) Affine 좌표 변환
(Projective to Affine Coordinates Conversion)
(e) 몽고메리 모듈러 곱셈 후처리 변환

또한, 위의 과정들을 수행하기 위해서는 다음과 같은 연산들이 필요하다.

- 모듈러 덧셈/뺄셈 연산
- 모듈러 역수 연산

• 몽고메리 모듈러 곱셈 연산

위 연산 과정 가운데 (a)와 (d)는 스칼라 곱셈 연산에 나눗셈 연산을 제거하기 위하여 사용되는데, 그 중 (d) 연산과정에서 모듈러 역수 연산이 사용된다. 본 논문에서는 역수 연산이 이 경우 한 번만 필요하기 때문에 별도의 역수기를 설계하지 않고 페르마 이론에 근거하여 곱셈기를 반복 이용함으로써 계산하였다. 스칼라 곱셈의 핵심 연산은 (c)이며, 전체 연산 가운데 가장 많은 시간을 필요로 한다.

스칼라 곱셈 연산은 (a)에서 (e)의 순서대로 연산이 전개된다. 위에서와 같이 Affine 좌표 변환이 몽고메리 후처리 변환 이전에 연산된다. Affine 좌표 변환은 모듈러 곱셈으로 연산이 가능하기 때문에 몽고메리 모듈러 곱셈기를 사용하기 위하여 Affine 좌표 변환 이후에 몽고메리 후처리 변환을 하였다. Affine 좌표 변환의 결과 값은 식(2)에서와 같이 정확한 좌표 값에 2^k 곱해진 값이 출력된다. 따라서 Affine 좌표 변환 이후에 몽고메리 후처리 변환을 해주어도 정확한 좌표 값을 계산할 수 있다.

4.1 투영 좌표 변환 및 Affine 좌표 변환

알고리즘 3에서 살펴본 바와 같이 타원 곡선에서 스칼라 곱셈 연산 가운데 나눗셈 연산을 제거하기 위하여 투영 좌표 변환을 이용한다. 전체 스칼라 곱셈 연산은 투영 좌표 변환 된 상태에서 이루어지며, 올바른 좌표 값을 찾아내기 위하여 스칼라 곱셈 연산 후 Affine 좌표 변환을 한다. 좌표 변환 과정은 다음과 같이 정의된다[3].

Affine 좌표(x, y) → 투영 좌표(X, Y, Z)

$$X = x,\ Y = y,\ Z = 1 \tag{3}$$

투영 좌표(X, Y, Z) → Affine 좌표(x, y)

$$x = XZ^{-2} \text{ and } y = TZ^{-3} \tag{4}$$

위 식에서 T는 알고리즘 3의 투영 좌표를 이용한 스칼라 곱셈 연산에서 T를 의미한다. 식 (4)에서와 같이 Affine 좌표 변환 과정은 역수 연산(Z^{-1})을 필요로 한다. 역수 연산을 위한 알고리즘으로는 페르마 이론과 확장 유클르드 알고리즘이 있다. 역수 연산의 경우 일반적으로 확장 유클리드 알고리즘이 페르마 이론보다 빠르다. 그러나 역수 연산을 곱셈 연산으로 변환하여 연산하는 페르마 이론을 이용할 경우 몽고메리 모듈러 곱셈기를 이용할 수 있으며, 전체 스칼라 곱셈 연산에서 역수 연산은 한 번만 필요로 하기 때문에 본 논문에서는 페르마 이론에 의한 방법을 사용하였다. 페르마 이론을 이용한 역수 연산 과정은 다음과 같이 정의된다.

$$Z^{-1} \bmod p = Z^{p-2} \bmod p \tag{5}$$

투영 좌표 변환을 이용한 두배점 연산에서는 알고리즘 3에서 알 수 있듯이 결과 값이 Y_2가 아닌 $2Y_2$값이 계산된다. Y_2를 계산하기 위하여 2로 나누는 연산은 오른쪽 쉬프트 연산과 같다. 그러나 LSB의 값이 '1'일 경우에는 오른쪽 쉬프트 연산을 할 수 없기 때문에 소수인 모듈러 연산을 위한 p의 값을 Y_2에 더함으로 '1'을 제거하여 쉬프트 연산을 한다. $2Y_2$를 T라 하면 $T/2 \bmod p$의 연산과정은 다음과 같다.

$$\begin{aligned} 2Y_2 &= T = t_{m-1}2^{m-1}+t_{m-2}2^{m-2}+\cdots+t_1 2+t_0 \\ Y_2 &= (T + (t_0 P)) / 2 \end{aligned} \tag{6}$$

4.2 몽고메리 곱셈 전처리 변환 및 후처리 변환

앞에서 살펴본 바와 같이 $A \times B \bmod P = S$일 경우 B의 값이 k-bit이면 몽고메리 알고리즘의 결과 값은 $S \times 2^{-k}$값이다. 즉 결과 값 S에 2^{-k} 곱해진 값이 출력된다. 이를 해결하기 위해 3장에서 살펴본 바와 같이 전처리 과정과 후처리 과정을 필요로 한다. 전처리 과정으로 스칼라 곱셈 연산을 위한 기본 좌표 X, Y와 투영 좌표 변환을 통하여 생성된 Z값을 다음과 같은 연산을 해주어야 한다.

$$\begin{aligned} X &= \text{Monpro}(X,\ 2^{2(k+2)}) \\ Y &= \text{Monpro}(Y,\ 2^{2(k+2)}) \\ Z &= \text{Monpro}(Z,\ 2^{2(k+2)}) \end{aligned} \tag{7}$$

모든 연산을 마친 후 올바른 좌표 값을 도출하기 위하여 후처리 과정을 해주어야 하며 다음과 같이 정의된다.

$$\begin{aligned} x &= \text{Monpro}(X,\ 1) \\ y &= \text{Monpro}(Y,\ 1) \end{aligned} \tag{8}$$

식(2) 에서는 전처리 과정을 위하여 2^{2k}을 곱하였지만 본 논문에서는 $2^{2(k+2)}$를 곱하여 전처리 과정을 하였다. 모듈러 곱셈 연산에서 초기 입력 조건이 A, $B < P$일 경우 몽고메리 모듈러 곱셈 연산은 알고리즘 4에서와 같이 덧셈 연산에 필요한 인수 값이

3개이기 때문에 k 번의 반복적인 연산 후 결과 값은 $S_{(k)} < 2P$가 되며 결과 값은 다음 연산의 입력으로 사용된다. 따라서 모듈러 곱셈 연산의 입력 조건을 $A,\ B < 2P$ 로 할 경우, $(k+1)$ 번의 반복적인 연산이 필요하며, 결과 값은 $S_{(k+1)} = (S_{(k)}+A+P)/2) < 3P$이 되는데, 이렇게 되면 다음 곱셈을 위한 입력 값으로는 적합하지가 않게 된다. 이를 해결하기 위하여 여분의 '0'을 입력에 패딩하는 방법을 사용하여 $(k+2)$-bit를 입력으로 넣어주게 되면 $S_{(k+2)} = (S_{(k+1)}+P)/2 < 2P$ 으로 항상 입력 조건에 부합하게 되어 곱셈 연산에서 중간 결과 값에 대한 추가 모듈러 감소 과정을 거칠 필요가 없이 다음 연산의 입력 값으로 사용될 수 있다. 그러므로 2^{2k}가 아닌 $2^{2(k+2)}$를 전처리 변환에 사용한다.

4.3 모듈러 덧셈 및 뺄셈

스칼라 곱셈에서의 점덧셈 연산과 두배점 연산에서는 모듈러 덧셈과 뺄셈연산을 필요로 한다. 본 논문에서는 다음과 같은 방법을 이용하였다.

```
Algorithm 5 : Modular Addition/Subtraction
intput : A, B, P
output : C = A + B mod P

if B > 0 then
    C1 = A + B ;
    C2 = C1 - P ;
else
    C2 = A + B ;
    C1 = C2 + P ;

if C2 < 0 then
    C = C1 ;
else
    C = C2 ;
```

V. 하드웨어 구현

소수체인 GF(p)에서의 스칼라 곱셈기는 몽고메리 알고리즘을 이용한 모듈러 곱셈기와 모듈러 덧셈/뺄셈기 그리고 연산의 결과 값을 저장하기 위한 레지스터와 컨트롤러로 구성되어 있다. 본 논문에서 구현한 스칼라 곱셈기의 전체 구조는 그림 1과 같다. 연산을 위하여 타원 곡선의 변수로서 P, a가 입력되며, 초기 좌표 x, y, 스칼라 곱셈 연산을 위한 r값이 입력된다. 또한 몽고메리 알고리즘에서 전처리 과정에 필요한 값으로 mon_2k값이 입력되며, 스칼라 곱셈의 결과 값으로 X_2, Y_2값이 출력 된다. 이

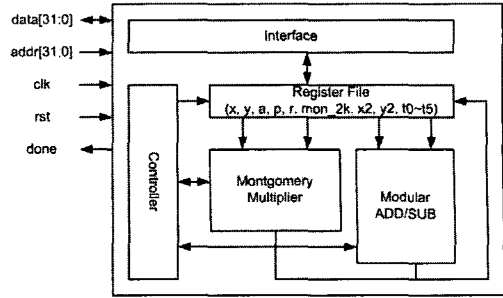


그림 1. 스칼라 곱셈기의 구조

때 mon_2k값은 식 (7)에서와 같이 추가적인 모듈러 연산을 제거하기 위하여 $2^{2(k+2)}$ mod P값을 입력한다.

본 논문에서 구현한 타원 곡선에서 점 덧셈 연산 과정은 표 1과 같다. 알고리즘 3에서와 같이 전체 연산은 두배점 연산과 점덧셈 연산으로 나누어지며, 곱셈 연산과 덧셈/뺄셈 연산이 동시에 이루어진다. 본 논문에서 설계한 스칼라 곱셈기는 표 1에서와 같이 각 연산을 위한 중간 결과값을 저장하기 위하여 6개(T_0-T_5)의 레지스터를 필요로 한다. 점 덧셈 연산에서 $T_1/2$ 은 식 (6)에서 살펴본 바와 같이 덧셈과 오른쪽 쉬프트 연산으로 계산이 가능하기 때문에 덧셈 연산으로 분류하였다.

표 1. 타원 곡선 점 덧셈 연산

<table>
<tr><th colspan="2">Elliptic Doubling</th><th colspan="2">Elliptic Addition</th></tr>
<tr><th>Mul.</th><th>Add./Sub.</th><th>Mul.</th><th>Add./Sub.</th></tr>
<tr><td>$T_1=X_1X_1$</td><td></td><td>$T_1=Z_1Z_1$</td><td></td></tr>
<tr><td rowspan="2">$T_2=Z_1Z_1$</td><td rowspan="2">$T_3=T_1+T_1$
$T_3=T_3+T_1$</td><td>$T_2=Z_1T_1$</td><td></td></tr>
<tr><td>$T_3=X_0T_1$</td><td></td></tr>
<tr><td>$T_2=T_2T_2$</td><td></td><td>$T_4=Y_0T_2$</td><td></td></tr>
<tr><td>$T_2=aT_2$</td><td></td><td>$T_5=Z_0Z_0$</td><td></td></tr>
<tr><td>$T_3=Z_1Y_1$</td><td>$T_4=T_3+T_2$</td><td>$T_1=T_5X_1$</td><td></td></tr>
<tr><td>$T_5=Y_1Y_1$</td><td>$\mathbf{Z_2=T_3+T_3}$</td><td>$T_5=T_5Z_0$</td><td></td></tr>
<tr><td>$T_3=T_5X_1$</td><td></td><td rowspan="2">$T_2=T_5Y_1$</td><td rowspan="2">$T_0=T_3-T_1$
$T_3=T_3+T_1$</td></tr>
<tr><td rowspan="2">$T_3=T_4T_4$</td><td rowspan="2">$T_2=T_3+T_3$
$T_2=T_2+T_2$
$T_1=T_2+T_2$</td></tr>
<tr><td>$T_4=Z_0Z_1$</td><td>$T_0=T_3-T_1$
$T_3=T_3+T_1$</td></tr>
<tr><td></td><td>$\mathbf{X_2=T_3-T_1}$</td><td>$\mathbf{Z_2=T_4T_0}$</td><td></td></tr>
<tr><td></td><td>$T_0=T_2-X_2$</td><td>$T_5=T_1T_1$</td><td></td></tr>
<tr><td>$T_1=T_5T_5$</td><td></td><td>$T_4=T_0T_0$</td><td></td></tr>
<tr><td rowspan="3">$T_3=T_4T_0$</td><td rowspan="3">$T_1=T_1+T_1$
$T_1=T_1+T_1$
$T_1=T_1+T_1$</td><td>$T_0=T_4T_0$</td><td></td></tr>
<tr><td>$T_4=T_4T_3$</td><td></td></tr>
<tr><td rowspan="3">$T_2=T_2T_0$</td><td rowspan="3">$\mathbf{X_2=T_5-T_4}$
$T_2=X_2+X_2$
$T_5=T_4-T_2$</td></tr>
<tr><td></td><td>$\mathbf{Y_2=T_3-T_1}$</td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td><td>$T_3=T_5T_1$</td><td></td></tr>
<tr><td></td><td></td><td></td><td>$T_1=T_3-T_2$</td></tr>
<tr><td></td><td></td><td></td><td>$\mathbf{Y_2=T_1/2}$</td></tr>
</table>

모듈러 곱셈기는 몽고메리 알고리즘을 이용하였으며 전체 구조는 그림 2와 같다. 알고리즘 4에서와 같이 몽고메리 모듈러 곱셈연산에서 덧셈 연산은 중간 결과 값 *Sum*, Multiplier *B*, 그리고 모듈러 연산을 위한 *P*, 3개의 입력 값을 필요로 하기 때문에 연산 시간을 단축하기 위하여 CSA와 CLA를 사용하였다. 전체 과정이 모듈러 A, $B < 2P$에서 연산되기 때문에 160-bit가 아닌 161-bit CSA와 162-bit CLA를 사용하였으며, CLA의 결과 값 163-bit 가운데 상위 162-bit 값이 출력되어 다음 연산을 위해 사용된다. 알고리즘 4에서와 같이 곱셈 연산은 LSB부터 MSB까지 검색하여 연산되며, 식 (7)에서 살펴본 바와 같이 입력 조건을 만족시키기 위하여 162번 반복하여 연산한다.

모듈러 덧셈/뺄셈기의 구조는 그림 3과 같다. 곱셈기와 마찬가지로 모듈러 덧셈/뺄셈기 입력 및 출력 값은 161-bit에서 연산된다. 하지만 뺄셈 연산을 하기 위하여 부호 비트가 필요하며 덧셈 연산에서의 캐리로 인해 Sign bit 가 수정되는 것을 피하기 위하여 여기에서는 163-bit CLA를 사용한다. 연산을 위한 A와 B값은 연산 초기에 입력되어 레지스터에 저장되지만 $-P$ 값은 스칼라 곱셈 연산 가운데 변하지 않는 값으로 계속 사용되기 때문에 스칼라 곱셈 연산 초기에 계산하여 레지스터에 저장하고 계속 사용한다.

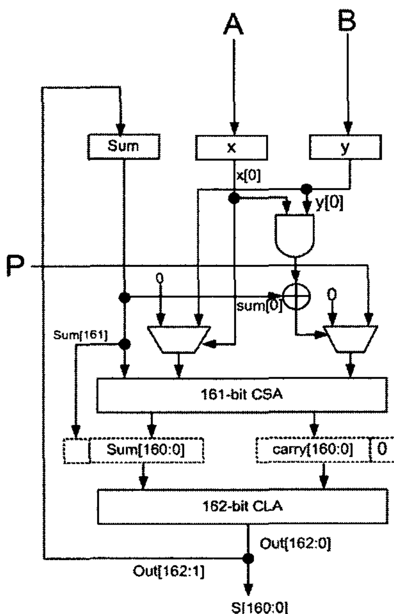


그림 2. 몽고메리 곱셈기 구조

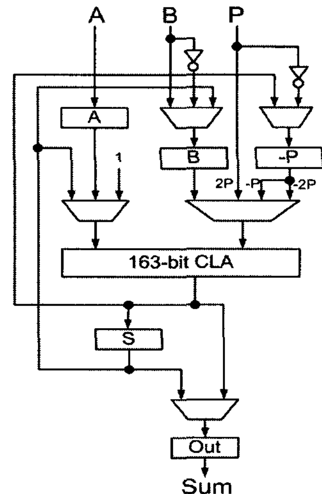


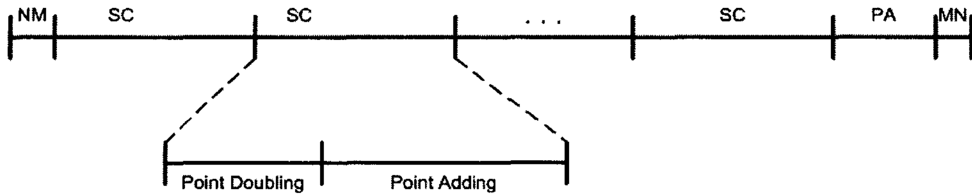
그림 3. 모듈러 덧셈/뺄셈기 구조

Ⅵ. 검증 및 성능 분석

본 논문에서 구현한 스칼라 곱셈기는 Verilog HDL을 이용하여 설계하였고, ALTERA(사)의 Excalibur PLD 칩에 합성하여 검증하였다. 검증은 IEEE P1363a 표준 문서에 제시되어 있는 테스트 벡터를 이용하고, 전체 시스템 검증을 위해 EC-DSA 및 EC-DH를 수행하여 획득한 키 값으로 텍스트 파일을 암호화 한 후 복호화하여 비교하는 방법으로 검증하였다.

그림 4는 스칼라 곱셈 연산의 타이밍도이다. 투영 좌표 변환 과정은 초기값을 입력하는 연산으로 클럭을 소모하지 않기 때문에 표현하지 않았다. 그림 4에서 보는 바와 같이 스칼라 곱셈에서 타원 곡선에서의 좌표 연산(SC)이 가장 많은 시간을 필요로 한다. 또한 역수 연산을 위하여 페르마 이론을 이용한 Affine 좌표 변환(PA)에서도 많은 곱셈 연산으로 인하여 많은 연산 시간을 필요로 함을 알 수 있다.

본 논문에서 설계한 스칼라 곱셈기의 성능 분석을 위해 Siddika의 논문[5]과 성능 및 사이즈를 비교하여 표 2에 나타내었다. 논문 [5]는 대부분의 ECC 관련 연구가 GF(2^m) 상에서 이루어진 관계로 GF(p) 에서의 구현 예로 찾은 유일한 논문이다. Siddika의 스칼라 곱셈기는 본 논문과 동일하게 투영 좌표 변환 및 몽고메리 알고리즘을 이용하였지만, 모듈러 곱셈기는 파이프라인 구조를 사용하였다. 파이프라인 구조의 곱셈기는 삽입되는 레지스터에 의해 최대 지연 시간이 짧아지는 장점이 있지만 단



NM(Normal to Montgomery) - 648 clock
SC(EC Scalar Multiplication) - 468480 clock
PA(Projective to Affine) - 39366 clock
MN(Montgoemry to Normal) - 324 clock

Point Doubling - 1629 clock
Point Adding - 2598 clock

그림 4. 점덧셈 연산과 두배점 연산의 타이밍도

표 2. 제안한 스칼라 곱셈기의 성능 비교

	Sıddıka[5]	본 논문
Normal to Montgomery	$12n+16$	$4(n+2)$
EC doubling	$36n + 38$	$10n + 29$
EC adding	$42n + 56$	$16n + 38$
m(EC point double) + m/2(EC point adding)	$m(57n+66)$	$m(18n+48)$
Projective to Affine	$3n^2 + 16n + 16$	$HW(p\text{-}2)\ (n+2) + (n\text{-}1)(n+2) + 4(n+2)$
Montgomery to normal	$6n + 8$	$2(n+2)$
Montgomery Multiplication	$3n + 4$	$n+2$
Modular Add/Sub	$2n+1$	3
Critical Path	$2T_{FA}+T_{HA} = 10T_g$	$T_{CLA} + T_{FA} + 4T_g = 25T_g$
Total number of clock	$60n^2+105n+40$ $= 1{,}552{,}840$ clock	$19.5n2+60n+18$ ($HW(p\text{-}2) = n/2$) $= 508{,}818$ clock
Total Area(gate)	115,520	64,539 (Samsung STD-130)

일 곱셈 연산을 위해 많은 클럭 수를 필요로 한다. 반면에 본 논문에서 설계한 스칼라 곱셈기는 모듈러 곱셈기에서 중간 결과 값의 덧셈 연산을 위한 CSA와 162-bit CLA를 한 클럭에 계산하므로 최대 지연 시간은 상대적으로 길지만, 전체 연산을 위해 필요로 하는 총 클럭 수의 감소로 인해 전체적인 성능의 향상을 가져온다.

표 2에서 보이는 바와 같이 [5]의 스칼라 곱셈기는 1,552,840개의 전체 클럭 수와 10개 게이트의 최대 지연 시간을 가지지만, 본 논문의 스칼라 곱셈기는 508,818개의 전체 클럭 수와 25개 게이트의 최대 지연 시간을 가진다. 표 2에서 TCLA는 162-bit 4-level CLA의 최대 지연 시간으로 17개 게이트의 지연 시간을 갖는 것으로 추정하였다.

본 논문에서 제안한 스칼라 곱셈기는 하드웨어 사이즈 면에서도 [5]에 비해 휠씬 우수함을 볼 수 있다. 제안한 곱셈기를 삼성 STD 130 셀 라이브러리를 이용해 ASIC 칩으로 합성하였을 경우 [5]의 115,520 게이트에 비해 64,539 게이트를 필요로 한다. 이 때 동작 속도는 98 MHz이며, 데이터 처리 속도는 29.6 kbps로 160-bit 프레임 당 5.4 ms 걸린다. 이러한 성능은 메모리와 하드웨어 사이즈가 제한적인 휴대용 보안 장치에서 디지털 서명, 암호화 및 복호화, 그리고 키 교환 등에 효율적으로 사용될 수 있을 것이다.

Ⅶ. 결 론

본 논문에서는 몽고메리 모듈러 곱셈 알고리즘을 이용하여 DTCP를 위한 타원 곡선 스칼라 곱셈기를 설계하였다. 제안한 스칼라 곱셈기는 모듈러 곱셈기와 모듈러 덧셈/뺄셈기로 구성되어 있으며, 스칼라 곱셈 연산에서 가장 많은 자원과 시간을 필요로 하는 모듈러 곱셈 연산을 효율적으로 처리하기 위하

여 몽고메리 알고리즘을 이용하였다.

본 논문에서 제안한 스칼라 곱셈기는 소수체 GF(p)에서의 타원 곡선 알고리즘을 이용하는 DTCP에서 EC-DSA와 EC-DH를 필요로 하는 Full Authentication에서 사용될 수 있으며, 그 밖에도 모듈러 곱셈 연산 및 GF(p)에서의 스칼라 곱셈을 필요로 하는 다양한 응용처에 사용될 수 있다. 제안한 구조는 성능 대비 하드웨어 사이즈가 작아 특히 휴대용 보안장치와 같은 임베디드 시스템에서 보안 코프로세서로 사용하기에 적합하다.

참 고 문 헌

〔1〕 DTCP Specification, "5C Digital Transmission Content Protection Specification Volume 1(Information Version)," July, 2000.

〔2〕 J. Kim, Y. Kim and Y. Jeong, "Implementation of a pipelined Scalar Multiplier using Extended Euclid Algorithm for Elliptic Curve Cryptography(ECC)," 한국정보보호학회, pp. 17~30. Oct. 2001.

〔3〕 IEEE P1363a/D5(Draft Version 5). Standard Specifications for Public Key Cryptography : Additional Techniques, August 16 2000

〔4〕 Certicom Whitepaper, "The Elliptic Curve Cryptosystem for Smart Cards," http://www.certicom.com, May, 1998

〔5〕 S. Berna Örs, L. Batina, B. Preneel and J. Vandewalle, "Hardware Implementation of an Elliptic Curve Processor over GF(p)," Proceedings of the Application-Specific Systems, Architectures, and Processor, IEEE 2003.

〔6〕 E.Savaş, A. F. Tenca and C. K. Koç, "A Scalable and Unified Multiplier Architecture for Finite Fields GF(p) and GF(2^m)," Workshop on Cryptographic Hardware and Embedded Systems(CHES), pp. 281~282. Aug. 2000.

〔7〕 Israel Koren, Computer Arithmetic Algorithms (2nd edition), 2002.

김 의 석(Eui seok Kim) 준회원

2003년 2월 광운대학교 전자공학부 졸업

2005년 2월 광운대학교 전자통신공학과 석사

2005년 2월 세인시스템 연구원

<관심분야> 보안, 제어 시스템 설계, SoC 설계

정 용 진(Yong jin Jeong) 정회원

1983년 2월 서울대학교 제어계측공학과 졸업

1983년 3월~1989년 8월 한국전자통신연구원

1995년 2월 미국 UMASS 전자전산공학과 박사

1995년 4월~1999년 2월 삼성전자 반도체 수석 연구원

1999년 3월~현재 광운대학교 전자공학부 부교수

<관심분야> 무선 통신, 정보보호, SoC 설계