情報保護學會論文誌
第11券 第5號, 2001. 10

확장 유클리드 알고리즘을 이용한 파이프라인 구조의 타원곡선 암호용 스칼라 곱셈기 구현

김 종 만*, 김 영 필*, 정 용 진**

Implementation of a pipelined Scalar Multiplier using Extended Euclid Algorithm for Elliptic Curve Cryptography(ECC)

Jong-Man Kim*, Young-Peel Kim*, Yong-Jin Jeong**

요 약

본 논문에서는 타원곡선 암호시스템에 필요한 스칼라 곱셈기를 $GF(2^{163})$의 standard basis상에서 구현하였다. 스칼라 곱셈기는 래딕스-16 유한체 직렬 곱셈기와 유한체 역수기로 구성되어 있다. 스칼라 곱셈을 계산하기 위해서는 유한체 곱셈, 덧셈과 역수의 계산이 필요하지만, 기존의 스칼라 곱셈기는 이러한 스칼라 곱셈을 유한체 곱셈기만으로 계산하였으므로 역수를 계산하는데 많은 시간을 소모하였다. 따라서, 본 논문의 중요한 특징은 가장 많은 연산시간을 필요로 하는 역수 연산을 빠르게 계산하기 위해 유한체 역수기를 추가 사용한 것이다. 유한체 역수기는 기존의 많은 구현 사례 중 두 번의 곱셈 시간이 소요되는 확장 유클리드 알고리즘(Extended Euclid Algorithm)을 이용하였다. 본 논문에서 구현한 유한필드 곱셈기와 역수기는 하드웨어 구조가 규칙적이어서 확장성이 용이하고, 파이프라인 구조와 하드웨어 리소스의 재활용을 이용해 계산과정에서 100%의 효율(throughput)을 발휘할 수 있는 구조를 가지고 있다. 스칼라 곱셈기는 현대전자 0.6㎛ CMOS 공정 라이브러리인 IDEC-C631을 이용하여 예측한 결과 최대 140㎒까지 동작이 가능하며, 이때 데이터 처리속도는 64*Kbps*로 163bit 프레임당 2.53㎳ 걸린다. 이러한 성능의 스칼라 곱셈기는 전자서명(Digital Signature), 암호화 및 복호화(encryption & decryption) 그리고 키 교환(key exchange) 등에 효율적으로 사용될 수 있을 것으로 여겨진다.

ABSTRACT

In this paper, we implemented a scalar multiplier needed at an elliptic curve cryptosystem over standard basis in $GF(2^{163})$. The scalar multiplier consists of a radix-16 finite field serial multiplier and a finite field inverter with some control logics. The main contribution is to develop a new fast finite field inverter, which made it possible to avoid time consuming iterations of finite field multiplication. We used an algorithmic transformation technique to obtain a data-independent computational structure of the Extended Euclid GCD algorithm. The finite field multiplier and inverter shown in this paper have regular structure so that they can be easily extended to larger word size. Moreover they can achieve 100% throughput using the pipelining. Our new scalar multiplier is synthesized using Hyundai Electronics 0.6㎛ CMOS library, and maximum operating frequency is estimated about 140㎒. The resulting data processing performance is 64*Kbps*, that is, it takes 2.53㎳ to process a 163-bit data frame. We assure that this performance is enough to be used for digital signature, encryption & decryption and key exchange in real time embedded-processor environments.

keyword : *ECC scalar multiplier, public key cryptography, euclid algorithm, finite field arithmetic*

* 광운대학교 전자통신공학과 석사과정
** 광운대학교 전자공학부 조교수

Ⅰ. 서 론

최근 전자상거래가 활성화됨에 따라서 전자서명, 인증 및 정보의 암호화에 대한 중요성이 대두되고 있다. 이러한 정보 보안에 필요한 암호 알고리즘으로 공개키와 비밀키 암호 알고리즘이 있으며, 이중 공개키 암호 알고리즘은 전자서명, 인증 및 암호화 모든 분야에 필연적으로 사용되어지고 있다. 현재 가장 많이 사용되고 있는 공개키 암호 알고리즘으로는 RSA(Rivest-Shamir-Adleman)와 ECC (Elliptic Curve Cryptography)가 있다. 이중 ECC는 RSA에 비해 작은 키 사이즈를 가지고도 높은 안전성을 갖는 장점이 있다. 예를 들어, 163bit 사이즈의 ECC 알고리즘이 1024bit 사이즈의 RSA와 동등 레벨의 안전성을 가지는 것으로 알려져 있다[8]. 그러므로 ECC와 RSA의 키 사이즈 비율이 1:7정도이며, 키 사이즈가 커질수록 동등 레벨의 안전성을 유지하기 위한 두 알고리즘의 키 사이즈 비율은 더욱 커진다. ECC는 키 사이즈가 상대적으로 작고 그로 인해 하드웨어 사이즈와 전력소모가 적어 휴대용 보안기기의 보안 코프로세서(Security Coprocessor)로서 사용되기에 적합하다. 휴대용 보안기기에서는 주로 Embedded-Micom을 사용하기 때문에 소프트웨어로써 실시간 조건을 만족시키기 어렵다. 이로 인해 본 논문에서 제시하는 것처럼 앞으로 보안 코프로세서의 필요성이 중요하게 대두될 것이다. 본 논문에서는 휴대용 보안장치에 사용할 수 있도록 ECC 스칼라 곱셈기를 설계하였고, 이를 검증하기 위해 FPGA (Field Programmable Gate Array)로 구현하였다. ECC의 주요 응용분야로는 전자서명, 암호화 및 복호화 그리고 키 교환 등이 있다.

ECC는 RSA와 함께 공개키 암호 알고리즘의 사실상의 표준으로 자리잡고 있는데, 1999년 1월에 ANSI(American National Standards Institute) 표준인 ANSI X9.62로 채택되어졌으며[12], 2000년 2월에는 IEEE(Institute of Electrical and Electronics Engineers)표준인 IEEE P1363으로 받아들여졌다[11]. NIST(National Institute of Standards and Technology)는 2000년 2월에 X9.62에 명시된 ECDSA(Elliptic Curve Digital Signature Algorithm)를 포함시키기 위해서 DSS (Digital Signature Standard)의 범위를 확대한다고 발표했다.

본 논문의 구성은 다음과 같다. Ⅱ에서는 지금까지 알려진 스칼라 곱셈기의 구현 사례를 살펴보고, Ⅲ에서는 ECC에 대해 간단히 소개한다. 스칼라 곱셈기에 필요한 유한필드 곱셈기와 역수기의 하드웨어 구조는 Ⅳ에서 설명하고, Ⅴ에서는 본 논문에서 제시한 스칼라 곱셈기의 성능에 대해 비교 설명한다.

Ⅱ. 구현사례

ECC의 핵심연산인 스칼라 곱셈은 역수연산을 포함한 알고리즘과 역수연산을 포함하지 않은 알고리즘으로 분류할 수 있다. 역수연산을 포함한 알고리즘은 주로 affine coordinate에 기초를 두고 스칼라 곱셈을 표현하고 있다. 이러한 구현사례들을 살펴보면 소프트웨어와 하드웨어의 두 가지 경우가 있는데, 소프트웨어로 구현한 예로는 $GF(2^{155})$에서 Diffie-Hellman key exchange를 연산하는데 걸리는 시간이 25㎒성능의 32비트-Sun SPARC IPC를 탑재한 컴퓨터를 이용할 경우 137ms이고, 175㎒성능의 64비트-DEC Alpha 3000를 탑재한 컴퓨터를 이용할 경우 11.5ms이다[6]. 이러한 성능은 실시간 조건을 만족시키기에 적합하지 못하다. 하드웨어로 구현한 예로는 [2],[4]가 있다. [2]는 standard basis에서 스칼라 곱셈을 수행하기 위해 가변적인 정수 직렬 곱셈기를 사용하였고, 서버 시스템을 목적으로 어떠한 크기의 필드(field)에 대해서 곱셈을 연산할 수 있도록 설계되어졌기 때문에 휴대용 장치로 적합하지 못하다. [4]는 optimal normal basis를 사용하여 스칼라 곱셈기를 구현하였다. 따라서 제곱연산에서 클럭을 소모하지 않는 장점이 있지만, 역수연산을 하기 위해 $GF(2^m)$상에서 $\lfloor \log_2(m-1) \rfloor + HW(m-1) - 1$(여기서, $HW()$는 Humming weight를 나타낸다)개의 곱셈연산이 필요하기 때문에 성능 저하의 요인이 된다[14].

역수연산을 포함하지 않는 알고리즘은 projective coordinate상에서 구현되어진다. [13]에서는 $GF(p)$인 소수체에서 binary method 방법을 이용해 x 좌표만으로 스칼라 곱셈을 계산하고, 마지막에 y좌표를 얻어낼 수 있는 알고리즘을 제시하였고, 이를 projective coordinate으로의 좌표변환을 통해 역수연산을 제거하였다. [3]에서는 [13]에서 제시한 알고리즘을 유한체에 적용해 좌표변환을 통한 역수를 제거하는 방법과 x 좌표만을 이용한 좌표변환 방법을 이용해 하드웨어로 구현하였다. [3]은 optimal normal basis에서 스칼라 곱셈기를 구현하였기 때문에 제곱연산에서 클럭을 소모하지 않는 이득이 있

지만, 좌표변환 과정에서 3개의 제곱과 16개의 곱셈 연산이 추가적으로 발생하기 때문에 높은 성능의 하드웨어 구현이 어렵다.

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

따라서 본 논문은 normal basis보다 유연성을 가지는 standard basis에서 스칼라 곱셈기의 성능을 향상시키기 위해 전체 성능의 큰 비중을 차지하는 역수 계산을 빠르게 하도록 유한체 역수기를 사용하였다. 이것이 본 논문의 핵심 아이디어이다. 역수기를 구현한 사례로는 [1], [14], [15], [16]이 있다. [1]은 확장 유클리드 알고리즘을 이용하여 standard basis에서 역수를 계산하는데 두 번의 유한체 곱셈시간만 필요하다. [14], [15], [16]은 Fermat's theorom을 이용하였다. [14]는 위에서 언급했듯이 normal basis상에서 역수를 계산하는데 $\lfloor \log_2(m-1) \rfloor + HW(m-1) - 1$개의 유한체 곱셈 시간이 걸리고, [15]는 [14]를 개선하여 $\lfloor \log_2(m-2) \rfloor + 1$개의 유한체 곱셈을 계산해야한다. [16]은 유한체 곱셈기와 유한체 제곱기의 전체 하드웨어를 게이트의 조합으로 구성함으로써 최장지연시간(critical pass)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 수십배 빠르게 역수를 계산할 수 있다.

위의 구현사례들을 고려하여 본 논문에서 제안한 스칼라 곱셈기는 standard basis를 사용하여 설계하였고, 스칼라 곱셈에서 가장 많은 연산 시간을 필요로 하는 역수(inverse)의 계산시간을 줄이기 위해서 두 번의 곱셈 시간만을 필요로 하는 확장 유클리드 알고리즘을 이용한 유한체 역수기를 유한체 곱셈기 외에 추가적으로 사용함으로써 기존의 구현사례들보다 높은 성능의 하드웨어를 구현하였다. 따라서 본 논문에서는 이러한 스칼라 곱셈기를 이용하여 Embedded-Micom 환경에서 빠르게 스칼라 곱셈을 처리하도록 하드웨어를 구현하였다.

Ⅲ. 타원곡선 암호 알고리즘

타원곡선 암호 시스템은 1985년 Neal Koblitz와 Victor Miller에 의해서 개발되어졌으며, 이를 이용한 타원곡선 암호 알고리즘은 타원곡선 이산 로그 문제(ECDLP : Elliptic Curve Discrete Logarithm Problem)에 근간을 두고 있다[11]. ECDLP는 타원곡선상의 임의의 한 점 P에 정수 k를 곱한 값이 $Q = \kappa\ \kappa\ k$P일 때, 점 Q와 P를 알고 있더라도 정수 k를 계산하기 어려움을 나타낸다. $\kappa\ \kappa$ 타원곡선 암호시스템을 구현하는데 필요한 핵심연산은 스칼라 곱셈이다. 스칼라 곱셈은 위에서 언급한 $Q = \kappa\ \kappa\ k$P를 구하는 것이다. 소수체(Prime Field)인 GF(p)와 유한체인 $GF(2^m)$상에서 모두 스칼라 곱셈 연산이 정의되어지며, 대부분 유한체 덧셈연산에서 캐리가 발생하지 않아 소수체보다 하드웨어 구현이 용이하다는 장점 때문에 $GF(2^m)$상에서의 연산을 사용하고 있다. 본 논문에서도 $GF(2^m)$상에서의 스칼라 곱셈을 구현하였다. 스칼라 곱셈이 이루어지는 타원곡선은 $GF(2^m)$상에서 (x, y)인 점들로 구성되어지고, 타원곡선은

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

의 형태를 가진다. 여기서 $a, b \in GF(2^m), b \neq 0$이다. 스칼라 곱셈, Q= kP를 계산하기 위해서는 동일한 두 점의 합(doubling one point)과 서로 다른 두 점의 합(adding two points)을 반복적으로 이용하는 Double and Add 알고리즘을 이용한다. 점 P와 스칼라 값 k에 대해 $P = (x, y) \in (GF(2^m), GF(2^m))$이고, $k = (k_{m-1}, k_{m-2}, \cdots, k_0)$, $k_i \in GF(2)$, $0 \leq i \leq m-1$이라 할 때 스칼라 곱셈의 계산과정은 다음과 같이 표현된다.

```
if k_{m-1} = 1 then Q = P
else Q = 0.
for i = m-2 downto 0 do
    Q = 2Q                  // doubling one point
    if k_i = 1 then Q = Q + P
                            // adding two points
  return Q                                        (2)
```

위 알고리즘에서 보듯이 $k = k_{m-1}^{m-1} + k_{m-2}^{m-2} + \cdots + k_0$

를 Horner's rule을 사용하여 $(\cdots((k_{m-1})2+k_{m-2})2 \cdots+k_0)$의 형태로 나타내면, 스칼라 값 k는 두 배 연산과 덧셈 연산의 반복된 연산과정으로 표현되므로 스칼라 곱셈 kP는 동일한 두 점의 합과 서로 다른 두 점의 합을 번갈아 가며 연산함으로써 스칼라 곱셈을 구할 수 있게 되는 것이다. 이 때 k_i가 '1'일 경우에는 점 P를 이전에 구했던 값에 더하고 난 후 그 점에 대해서 동일한 두 점의 합을 계산하고, k_i가 '0'일 경우에는 서로 다른 두 점의 합의 연산 없이 바로 동일한 점의 합을 구하면 된다. 결국 $GF(2^m)$에 대해 m-1번의 동일한 두 점의 합과 최대 m-1번의 서로 다른 두 점의 합의 계산만으로 Q를 계산할 수 있다.

이러한 ECC의 핵심연산인 스칼라 곱셈을 구하는 과정에서 타원곡선상의 두 점 P, Q를 더한다는 의미는 정수에서의 연산과는 달리 다음과 같은 의미를 갖는다[9].

- 타원곡선상의 두 점 P와 Q를 지나는 일직선을 긋는다.
  만일 P와 Q가 같은 점이라면, 그 점에서 접선을 긋는다.
- 타원곡선은 삼차 방정식으로 P와 Q이외의 또 다른 점과 교차하게 된다.
- 그 점에서 x축에 수직인 직선을 긋는다.
- 그 수직선과 만나는 또 다른 타원곡선상의 점이 P+Q인 점이 된다.

위에서 보듯이 스칼라 곱셈의 핵심연산은 두 점의 합을 구하는 것이며, 타원곡선상의 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 같이 표현된다[5,7].

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

$$\begin{cases} x_3=\lambda^2+\lambda+a \\ y_3=x_1^2+\lambda x_3+x_3 \end{cases},\ \lambda=x_1+\frac{y_1}{x_1} \qquad (3)$$

  else $P_3=0$;
else // adding two points

$$\begin{cases} x_3=\lambda^2+\lambda+x_1+x_2+a \\ y_3=\lambda(x_1+x_3)+x_3+y_1 \end{cases},\ \lambda=\frac{y_1+y_2}{x_1+x_2} \qquad (4)$$

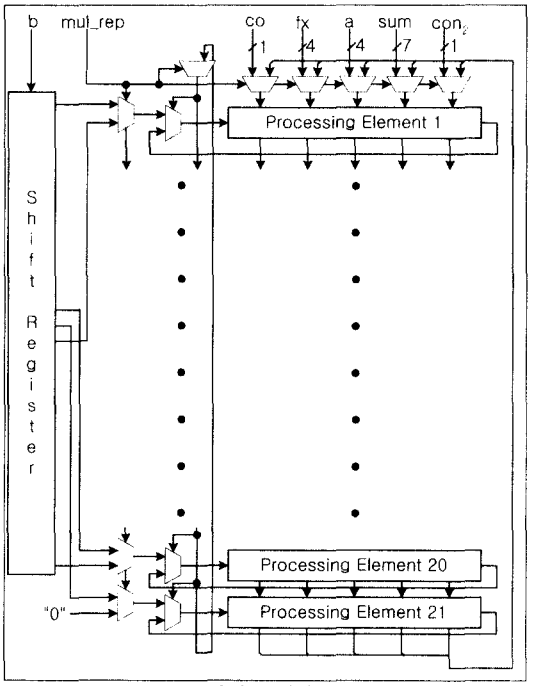
윗 식을 보면 식 (3)은 2번의 제곱과 곱셈 그리고 한번의 역수 계산이 필요하고, 식 (4)는 한번의 제곱과 2번의 곱셈, 그리고 한번의 역수 계산이 필요하다. 여기서 식 (3)의 x_1^2의 제곱 연산은 역수 계산시 유한체 곱셈기에서 미리 연산이 가능하므로 식 (3), (4)는 각각 한번의 제곱과 두 번의 곱셈 그리고 한번의 역수 계산 시간이 필요하다. 식 (2)에서 보듯이 곱셈과 역수의 계산이 반복루프에서 m-1번 반복되므로 가장 많은 계산시간을 필요로 하는 역수 연산이 전체 스칼라 곱셈기의 성능을 좌우하기 때문에 역수기의 구현이 가장 중요한 요인이 된다.

Ⅳ. 하드웨어 구조

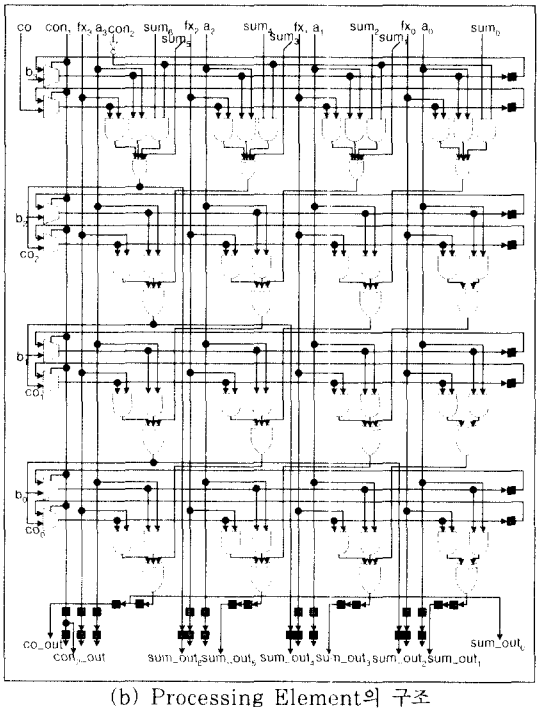
Ⅳ앞에서 설명했듯이 스칼라 곱셈을 계산하기 위해서는 m-1번의 동일한 두 점의 합과 최대 m-1번의 서로 다른 두 점의 합의 계산이 필요하며, 각 계산을 나타내는 식 (3), (4)를 계산하기 위해서 2번의 제곱과 4번의 곱셈 그리고 2번의 역수 계산이 필요하게 된다. 그러나 덧셈은 단지 exclusive-OR 연산이므로 전체 연산시간에 거의 영향을 미치지 않기 때문에 무시한다. 제곱 연산은 곱셈기를 사용하여 계산하므로 곱셈연산으로 간주하고, 곱셈기는 동작 클럭수를 줄이기 위해 래딕스-16 곱셈기를 사용하였다. 곱셈과 역수를 계산하기 위해 필요한 클럭수가 $GF(2^m)$상에서 각각 2m과 4m이고, 최장지연시간을 각각 T_M과 T_I라 하면, 한번의 스칼라 곱셈을 계산하는데 걸리는 시간은 $\{6\lceil m/2\rceil(m-1)\}T_M+\{8m(m-1)\}T_I$이다. 이러한 스칼라 곱셈기는 크게 유한체 곱셈기, 덧셈기, 역수기 그리고 변수와 결과 값을 저장하기 위한 레지스터와 이들을 제어하기 위한 컨트롤 박스로 구성되어진다.

1.1 유한체 곱셈기

유한체 곱셈기는 직렬 곱셈기와 병렬 곱셈기의 형태로 구현이 가능하지만, 병렬 곱셈기를 사용할 경우 레지스터와 곱셈기, 덧셈기와 역수기를 사이를 연결하는 데이터 패스가 많아지고 중간에 생겨나는 값들을 저장하는데 부수적인 하드웨어가 더 필요하다. 따라서 본 논문에서는 직렬곱셈기를유한체 역수기와의 최장지연시간을 맞추기 위해 $GF(2^{163})$상에서 래딕스-16 직렬곱셈기로 새롭게 디자인하였다. [그림 1](a)의 직렬 곱셈기를 살펴보면, 곱하여질 두 polynomial인



(a) 유한체 곱셈기의 구조



(b) Processing Element의 구조

〔그림 1〕 유한체 곱셈기

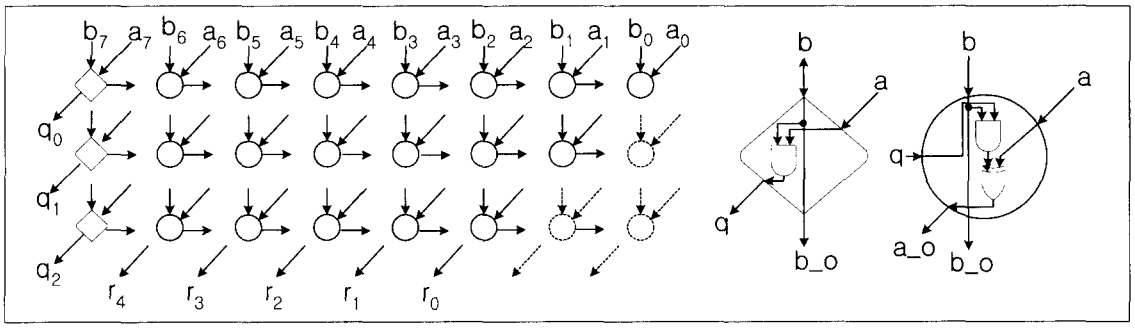
$a(x)$와 $b(x)$ 그리고 모듈러스 값인 $f(x)$가 4비트씩 직렬로 입력되어지고, 초기 입력 값으로 co와 sum 값을 '0'으로 입력하도록 설계되어 있다. $b(x)$의 값은 직렬로 입력되어진 후 쉬프트 레지스터를 거치면서 병렬로 각각의 PE(Processing Element)에 전달되어진다. 이러한 구조에서 래딕스-16 직렬 곱셈기를 구현한다면 전체 $\lceil m/4 \rceil$ 개의 PE가 필요하게된다. 이때, 멱승 계산에서처럼 곱셈한 결과를 재 사용하거나 여러 개의 곱셈 연산을 연속적으로 계산할 때에는 하드웨어 사용효율이 100%이지만, 곱셈한 결과를 또 다른 입력의 곱셈 값으로 사용할 경우에 입력한 결과가 출력되기까지 기다려야 한다. 따라서 절반 정도의 PE가 곱셈을 수행하는 동안 사용되지 않기 때문에 하드웨어 사용효율이 50%정도 밖에 되지 않는다. 본 논문에서는 이를 보완하기 위해서 곱셈 연산동안 연산과정에 관여하지 않는 PE를 다시 사용하는 방법으로 $\lceil m/8 \rceil$ 개의 PE만으로 곱셈기를 구현하였다. 〔그림 1〕(b)는 각 PE의 내부구조를 나타낸 것으로, 4비트에 해당하는 연산이 하나의 PE에서 한 클럭에 동작하게 되어 있다. 이는 래딕스-2 구조에 비해 PE내에서의 하드웨어 오버헤드는 커지지만 파이프라인 레지스터가 줄어들기 때문에 전체적으로 볼 때 하드웨어 리소스는 비슷하게 된다.

1.2 유한체 역수기

역수를 구하는 대표적인 방법으로는 Fermat theorem[9]과 확장 유클리드 알고리즘을 이용하는 방법[1]이 있다. Fermat theorem은 임의의 수 a와 p가 서로 소의 관계를 가질 때,

$$\begin{cases} \cdot a^{p} = a \mod p, \ a \in GF(p) \\ \cdot a^{-1} = a^{p-2} = a^{2^m} - 2 \mod 2^m, \end{cases} \quad p = 2^m \quad (5)$$

과 같이 표현되며, Fermat theorem으로 역수를 구할 경우에 $GF(2^m)$상에서 m-1번의 유한체 곱셈과 m-1번의 유한체 제곱의 연산이 필요하다. 이를 앞의 구현사례에서 살펴보았듯이 여러 알고리즘을 이용하여 연산의 수를 줄였다. 하지만 확장 유클리드 알고리즘을 이용할 경우 두 번의 곱셈 시간만 소요되며, 이는 다른 알고리즘에 비하여 가장 빠르며, 식 (10)에서 수식으로 설명하였다. 따라서 본 논문에서는 〔1〕에서 제안하고 기능적으로 검증된 알고리즘



〔그림 2〕 Polynomial 나눗셈과 Processing Element : $b(x)=q(x)a(x)+r(x)$

을 최적화하여 유한체 역수기를 구현하였다. 역수기의 구현을 위해 유클리드 알고리즘(Euclid Algorithm)과 확장 유클리드 알고리즘에 대해 설명한다.

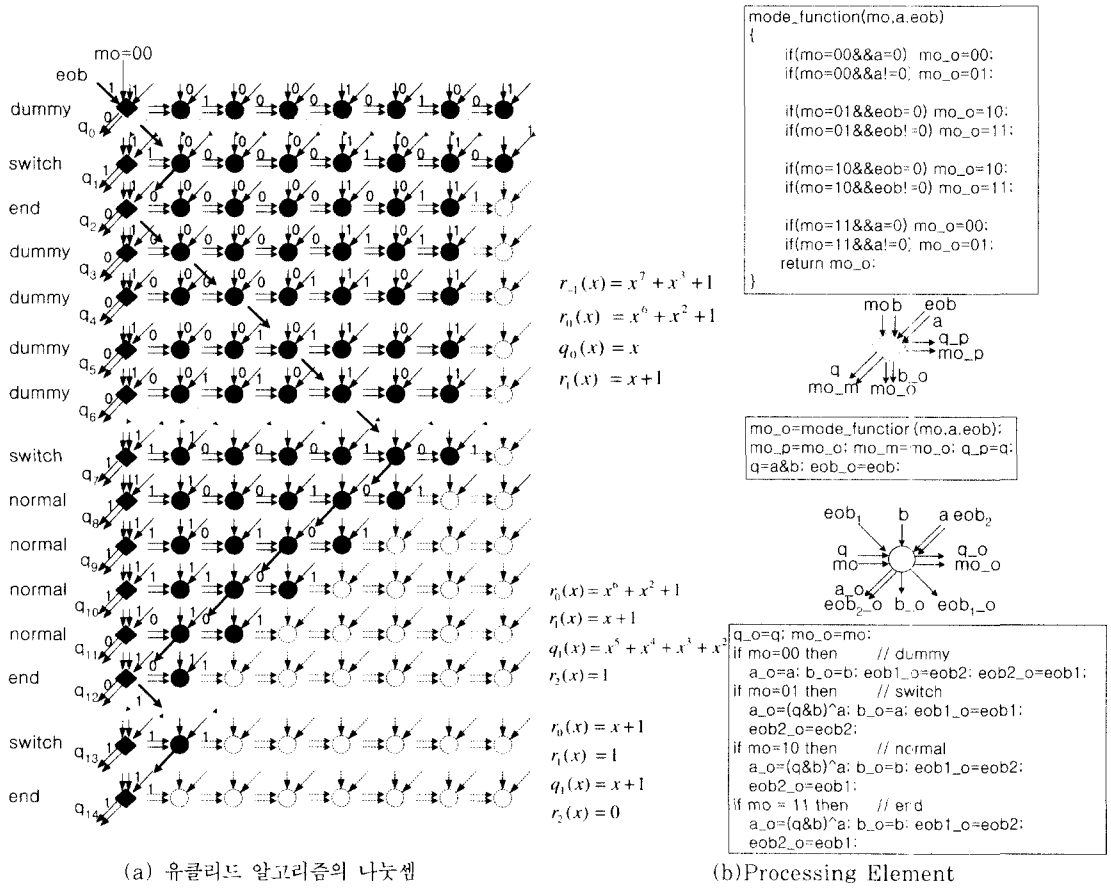
두 polynomial, $a(x)$와 $b(x)$, $\{\deg(a(x)) < \deg(b(x))\}$의 GCD(greatest common divisor)를 유클리드 알고리즘을 이용하여 계산하는 과정은

$$\begin{aligned} r_{-1}(x) &= b(x), \quad r_0(x) = a(x) \\ r_{-1}(x) &= q_0(x)\ r_0(x) + r_1(x) \\ r_0(x) &= q_1(x)\ r_1(x) + r_2(x) \\ &\vdots \quad \vdots \\ r_{k-1}(x) &= q_k(x)\ r_k(x) + r_{k+1}(x) \end{aligned} \tag{6}$$

이다. 위 식은 $r_{k+1}=0$일 때까지 반복적으로 계산하며 $b(x)=a(x)q(x)+r(x)$이면 $GCD(a(x),b(x))=GCD(a(x), r(x))$인 관계를 이용한 것이다. $r_{k+1}(x)=0$일 때, $r_k(x)$가 $a(x)$와 $b(x)$의 최대공약수, 즉 $r_k(x)=GCD(a(x), b(x))$가 된다. 〔그림 2〕는 〔1〕에서 제안된 알고리즘을 간단히 소개하는 것으로 식 (6)의 각 단계에서 필요한 유한체 나눗셈을 계산하는 과정과 PE를 나타내며, $b(x)=q(x)\times a(x)+r(x)$를 계산하는 과정을 보여주고 있다. 여기서 $a(x)$의 차수는 m이고 $b(x)$의 차수가 n일 때, $q(x)$의 차수는 $n-m+1$이고 $r(x)$의 차수는 $m-1$보다 작거나 같게된다. 따라서 각 단계에서 $a(x)$와 $b(x)$의 차수의 차이에 따라 $q(x)$와 $r(x)$의 크기가 달라진다. 그리고 마름모 모양의 PE는 식 (6)에서 몫인 $q(x)$를 출력하고, 원 모양의 PE는 나머지 $r(x)$를 계산하며, 점선으로 그려진 원은 사용되지 않는 PE이다. 〔그림 3〕은 〔그림 2〕의 유한체 나눗셈을 이용하여 $a(x)=x^6+x^{2+1}$이고, $b(x)=x^7+x^3+1$ 일 경우에 $GCD(a(x),b(x))$를 구하는 과정을 나타내고 있으며, 식 (6)의 세 단계만으로 계산이 이루어지므로, 〔그림 3〕에서 보듯이 세 개의 큰 유한체 나눗셈의 블록으로 나누어진다. 각 블록은 유한체 나눗셈이 시작할 수 있는 조건인 피제수 다항식과 제수 다항식 $(r_{i-1}(x), r_i(x))$의 자릿수를 맞춘 후, 유한체 나눗셈을 시작한다. 그리고 유한체 나눗셈이 어느 시점에서 끝나야 하는지, 그리고 어느 시점에서 피제수다항식과 제수 다항식 그리고 나머지의 값들을 서로 바꾸어야 하는지를 결정해야 한다. 다음의 4가지 경우가 확장 유클리드 알고리즘의 각 스테이지의 상태를 나타내며 다음과 같은 의미를 갖는다.

- dummy : 유한체 나눗 셈을 하기 위한 자릿수를 맞추는 상태.(부호화 : 00)
- switch : 제수와 피제수의 값을 바꾸는 상태. (부호화 : 01)
- normal : 나눗셈의 정상적인 연산을 하는 상태. (부호화 : 10)
- end : 식 (6)의 한 단계가 끝남을 나타내는 상태.(부호화 : 11)

〔그림 3〕에서 보듯이 초기 열의 상태는 유한체 나눗셈을 시작하기 위해 피제수와 제수의 자릿수를 맞추어야 하므로 dummy상태이다. 그 다음 열부터 각 열의 상태 전이는 〔그림 3〕(b)의 마름모 모양의 PE의 입력인 a와 eob의 값에 따라 이루어진다. 그리고 switch상태에서는 데이터의 진행방향이 바뀌게 된다. 이러한 상태 전이를 위해 따로 컨트롤 로직을 두지 않고 〔그림 3〕의 마름모 모양의 PE에서 제어하도록 했다. 그리고 식 (10)에 설명했듯이 하드웨어 구조가 공정되어 있기 때문에 필요하지 않는 sob신호를 제거하였다. 여기에서는 상태 전이가 데이터 의존적이므로 여러 개의 PE를 하나의 PE로 묶어서 데이터를 처리하는 하이래딕스 방식을 사용하기가 어렵다.



(a) 유클리드 알고리즘의 나눗셈 (b)Processing Element

〔그림 3〕 유클리드 알고리즘의 데이터 흐름

식 (6)에서의 GCD를 $g(x)$라 하면, 확장 유클리드 알고리즘에 의해서 다음과 같은 관계가 성립한다[7].

$$s(x)b(x)+t(x)a(x)=g(x),$$
$$g(x)=GCD(a(x),b(x)) \quad (7)$$

아울러 식 (7)의 $s(x)$와 $t(x)$는 다음과 같은 반복적인 계산으로 구할 수 있다.

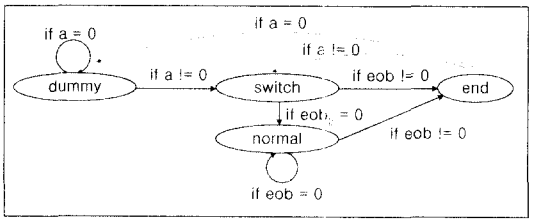
$$s_{-1}(x)=1, \quad s_0(x)=0$$
$$t_{-1}(x)=0, \quad t_0(x)=1$$
$$s_i(x)=q_{i-1}(x)\ s_{i-1}(x)+s_{i-2}(x), \ (-1\le i\le k+1)$$
$$t_i(x)=q_{i-1}(x)\ t_{i-1}(x)+t_{i-2}(x) \quad (8)$$

식 (8)에서 $q_i(x)$는 식 (6)에서 계산되어지며, 식 (6)의 $r_{k+1}=0$일 때까지 $t(x)$를 반복적으로 계산하는 $t_k(x)$ $b(x)$가 되고, $t_{k-1}(x)$는 $a^{-1}(x)$가 된다.

여기서 식 (7)을 다시 쓰면

$$t(x)\,a(x)=g(x) \mod b(x) \quad (9)$$

이 된다. 이때 $b(x)$가 irreducible polynomial이므로 항상 $g(x)=1$이다. 따라서 $t(x)a(x)=1 \mod b(x)$이고



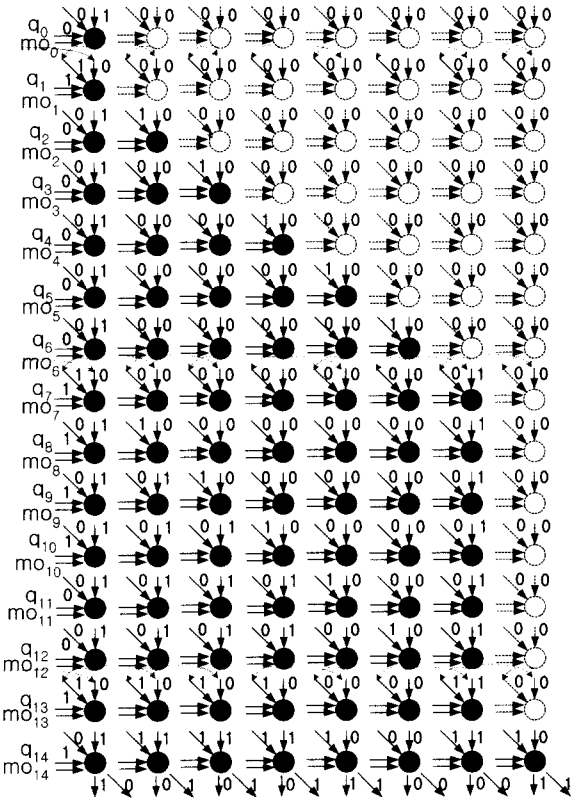
〔그림 4〕 확장 유클리드 알고리즘에서 각 열의 상태 전이도

$t(x)=\frac{1}{a(x)} \bmod b(x)$가 된다.

[그림 5]는 식 (8)에서의 반복적인 유한체 곱셈과 덧셈의 데이터 흐름을 보여주고 있다. 좌측에서는 그림 3에서 계산되어져 전달된 $q_i(x)$가 입력되어지고, 위쪽에서는 식 (8)의 초기 값인 $t_{-1}(x)$, $t_0(x)$가 입력된다. 그리고 [그림 3]과 데이터 패스는 다르지만, [그림 5]는 [그림 3]에 맞추어서 계산이 이루어지는 구조를 가지고 있으므로 똑같은 열의 PE가 필요하다. [그림 3]을 보면 좌측 대각선 방향으로 데이터가 움직이지만, [그림 5]는 우측 대각선 방향으로 데이터가 움직이며, 같은 배열의 PE구조를 가진다. 이들 두 가지 하드웨어 구조가 병렬적으로 데이터를 처리하기 때문에 역수기는 두 하드웨어가 겹쳐진 구조를 가져야한다. 유한체에서 확장 유클리드 알고리즘을 계산하는데 필요 한 열의 수는 $GF(2^m)$ 상에서 $2m+1$개이다. [그림 4]에서dummy에서 end로 전이되는 과정을 하나의 블록이라 하고, i번째 블럭에서 피제수와 제수의 차수를 m_i과 n_i이라 할 때, 첫 번째 블록의 dummy 열의 수는 m_0-n_0이고, 그 이후 i번째 블럭에서 dummy 열의 수는 m_i-n_i-1 이며, switch, normal 과 end를 합한 전체 열의 수는 m_i-n_i+1 이다. 따라서 전체 열의 수는

$$\sum_{i=0}^{j}(m_i-n_i+1)+\sum_{i=0}^{j}(m_i-n_i-1)+1$$
$$=\sum_{i=0}^{j}2(m_i-n_i)+1$$
$$=2\,m_0-2\,n_j+1 \qquad (10)$$

가 된다. 마지막 열의 제수 다항식의 차수는 초기값의 제수 다항식과 피제수 다항식이 서로 소의 관계를 가지므로 항상 1이다. 따라서 마지막 열인 n_j의 값은 1이고, 제수인 $b(x)$의 차수(m_0)는 $m+1$이므로 전체 열의 수는 $2m+1$개가 필요하다. 그러므로 역



(a) 확장 유클리드 알고리즘의 곱셈 (b) Processing Element

[그림 5] 확장 유클리드 알고리즘에서의 곱셈과 덧셈

수기가 데이터 종속적인 구조를 가지고 있으나 PE의 갯수가 $2m+1$개로 고정되어 있으므로 [그림 6]에서 보듯이 파이프라인 구조를 만들어 낼 수 있다.

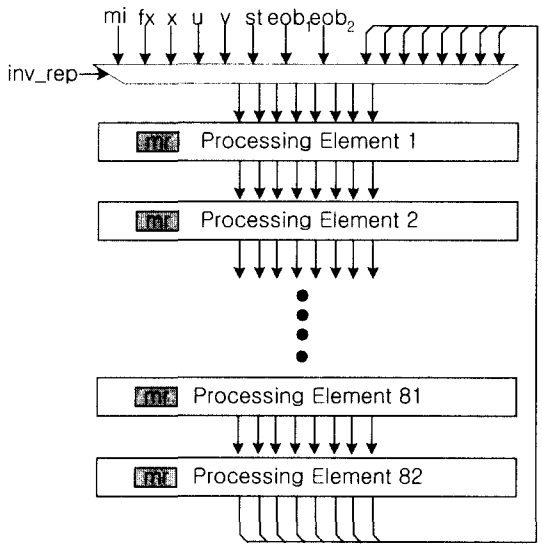
이전 예제에서 [그림 3]과 [그림 5]에서 역수를 구하기 위해 병렬적으로 계산되는 과정의 예를 보여주고 있다. [그림 3]과 [그림 5]에서처럼 3개의 블럭을 거치면서 계산되어진다. 첫 번째 열과 두 번째 열은 [그림 3]에서의 계산과정을 나타내고, 세 번째 열은 [그림 5]에서의 계산과정을 나타낸다. 두 번째 열은 첫 번째 열의 $r_{i-1}(x)$와 $r_i(x)$를 이용하여 몫과 나머지인 $q_i(x)$와 $r_{i+1}(x)$를 구하고, 세 번째 열은 $q_i(x)$와 $t_{i-1}(x)$를 이용하여 $t_{i+1}(x) = q_i(x)\ t_i(x) + t_{i-1}(x)$를 계산한다. 각 단계를 거칠 때마다 피제수와 제수의 값이 이전 단계의 제수와 나머지로 대치되고, 세 번째 열의 $t_i(x)$와 $t_{i-1}(x)$는 이전 행의 $t_{i+1}(x)$와 $t_i(x)$로 대치된다. 이러한 과정을 $r_{i+1}(x)$값이 zero가 될 때까지 반복한다.

$r_{i+1}(x)$가 zero일 때 $t_2(x)$의 값이 $a^{-1}(x) \bmod b(x)$가 된다.

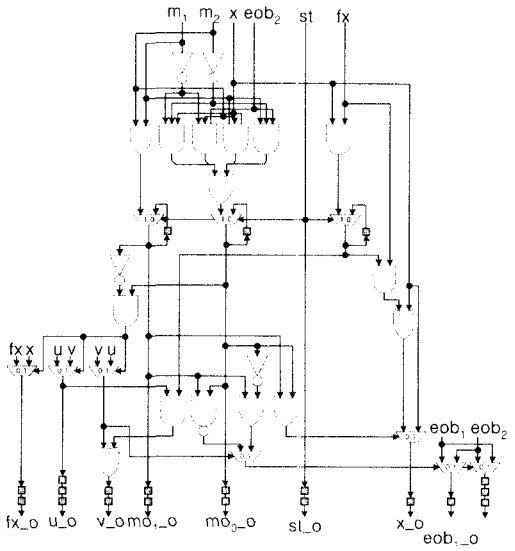
[그림 6]은 역수기의 하드웨어 전체 구조를 나타낸 것이다. 역수기의 각각의 PE는 [그림 3]과 [그림 5]의 하드웨어 구조를 겹쳐놓은 구조이며, [1]에서 기능적으로 검증한 알고리즘을 본 논문에서 최적화하여 하드웨어로 구현하였다. 곱셈기는 래딕스-16로 구현되었지만, 역수기는 직렬로 1비트씩 계산이 이루어지도록 설계되었다. 그 이유는 역수기의 하드웨어 구조가 앞 절에서 설명한 바와 같이 모드 신호에 따라서 데이터의 진행방향이 결정되는 데이터 종속적인 구조를 가지고 있으며, 이에 따른 PE의 최장지연시간을 곱셈기에도 적용하기 위함이다. 역수

example

단계	$r_{i-1}(x)$ $r_i(x)$	$q_i(x)$ $r_{i+1}(x)$	$t_{i-1}(x),\ t_i(x)$ $t_{i+1}(x)$
1	$r_{-1}(x) = x^7 + x^3 + 1$ $r_0(x) = x^6 + x^2 + 1$	$q_0(x) = x$ $r_1(x) = x + 1$	$t_{-1}(x) = 0,\ t_0(x) = 1,$ $t_1(x) = x$
2	$r_0(x) = x^6 + x^2 + 1$ $r_1(x) = x + 1$	$q_1(x) = x^5 + x^4 + x^3 + x^2$ $r_2(x) = 1$	$t_0(x) = 1,\ t_1(x) = x$ $t_2(x) = x^6 + x^5 + x^4 + x^3 + 1$
3	$r_1(x) = x + 1$ $r_2(x) = 1$	$q_2(x) = x + 1$ $r_3(x) = 0$	$t_1(x) = x,\ t_2(x) = x^6 + x^5 + x^4 + x^3 + 1$ $t_3(x) = x^7 + x^3 + 1$



(a) 역수기의 구조



(b) Processing Element

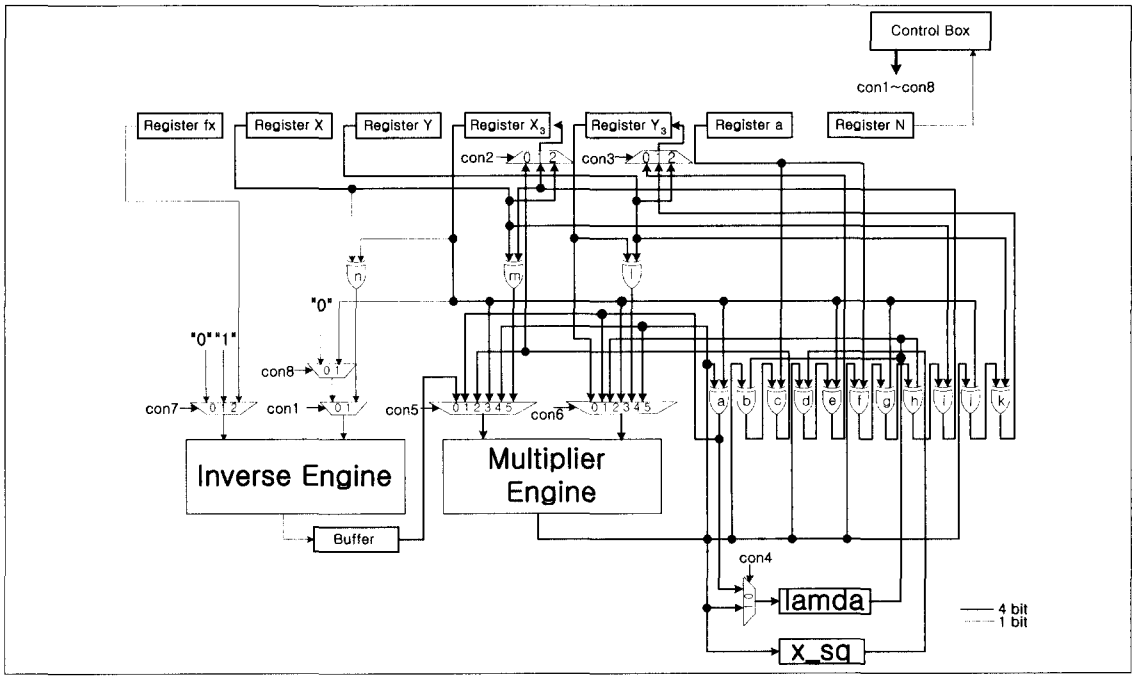
[그림 6] 역수기

계산이 수행되는데 걸리는 시간은 $GF(2^{163})$에서 652 클럭 소요된다. 즉 필드의 크기가 m비트일 경우 $4m$ 클럭이 걸리는 것이다. 역수기 또한 곱셈기처럼 $GF(2^m)$에서 전체 하드웨어를 구성하기 위해서 $2m+1$개의 PE가 필요하다. 하지만 식 (3), (4)를 보면 알 수 있듯이 역수를 연속해서 구할 필요가 없기 때문에 불필요한 하드웨어 리소스를 줄이기 위해 전체 PE의 1/4수준인 $\lceil (2m+1)/4 \rceil$개의 PE와 추가적인 rep_inv 신호를 사용하였다.

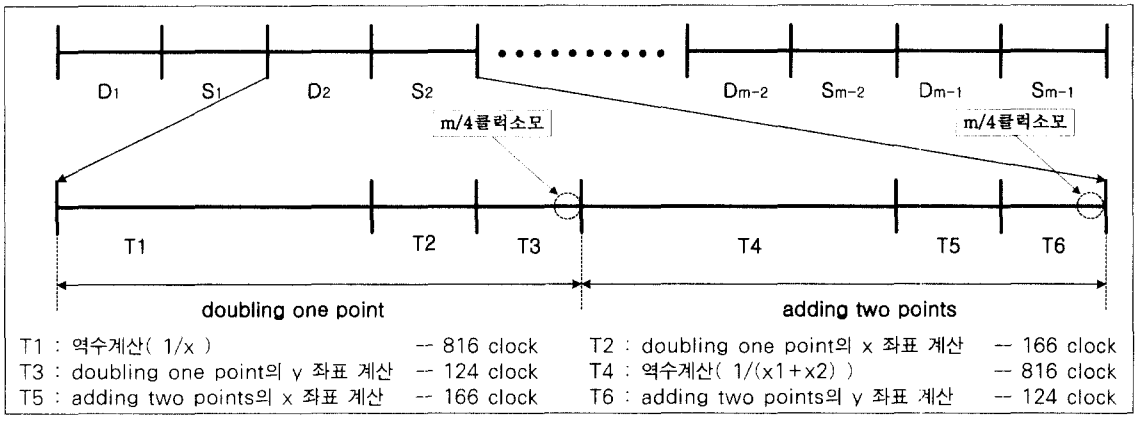
1.3 스칼라 곱셈기 전체 구조

〔그림 7〕은 전체 스칼라 곱셈기의 구조를 보여주고 있다. 스칼라 곱셈기의 전체 하드웨어 구조는 유한체 곱셈기, 역수기와 덧셈기 그리고 각각의 변수 값, 결과 값과 중간 값을 저장하기 위한 레지스터들 그리고 이들의 데이터 흐름을 제어하기 위한 컨트롤박스로 구성되어 있으며, 전체 데이터 흐름의 손실을 없애기 위해 전체 시스템을 새롭게 설계하였다. 앞에서 언급했듯이 역수기는 1비트씩 연산이 이루어지도록 설계되었고, 곱셈기는 4비트씩 계산이 이루어지도록 설계되어졌다. 그래서 각각의 변수 값을 저장하고 있는 레지스터가 유한체 역수기와 곱셈기의 경우 모두 사용할 수 있도록 역수기를 이용할 경우에는 1비트씩 데이터가 쉬프트되도록 레지스터가 설계되어졌고, 곱셈기를 이용할 경우는 4비트씩 데이터가 쉬프트되도록 설계되어졌다. 또한 PC로부터 PCI버스를 통해서 데이터를 받기 위해 32비트씩 데이터가 입출력되는 구조를 가지고 있다. 곱셈기와 역수기의 데이터 처리량이 다른 이유는 역수기의 데이터 패스의 길이가 곱셈기보다 4배정도 길기 때문이다. 스칼라 곱셈을 처리하기 위해 8개의 컨트롤 신호가 필요한데, con2와 con3는 식 (3), (4)에서 계산되어진 결과 값(x,y)을 저장하기 위해 필요한 신호이고 con4는 식 (3), (4)의 각 λ값을 저장하기 위해 필요한 신호이다. 그리고 나머지 컨트롤 신호들은 유한체 곱셈기와 역수 기의 입력 값을 결정하기 위해 요구되어지는 신호들이다.

스칼라 곱셈이 이루어지는 과정은 크게 서로 다른 두 점의 합과 동일한 두 점의 합을 계산하는 부분으로 나뉘어진다. 〔그림 8〕은 스칼라 곱셈이 이루어지는 전체 과정을 나타내는 타이밍도를 보여주고 있다. 타이밍도에서 D_i(T1,T2 그리고 T3) 부분이 동일한 두 점의 합을 계산하는 부분으로 식 (2)를 계산하는 과정을 나타내며, S_i(T4, T5 그리고 T6) 부분이 서로 다른 두 점의 합을 계산하는 부분으로 식 (3)을 계산



〔그림 7〕 스칼라 곱셈기 전체구조



[그림 8] 타이밍도

하는 과정을 나타낸 것이다. 각각의 타이밍도를 살펴보면 가장 많은 연산 시간을 필요로 하는 역수를 연산하는 과정이 한번씩 존재하고, 각각 4번과 3번씩 곱셈이 존재하며 클럭을 소모하지 않는 연산인 덧셈연산이 여러 번씩 이루어진다. 맨 마지막 부분에 중간결과를 저장하기 위해 클럭을 소모하고 있는데, 이것은 스칼라 곱셈인 kP를 연산할 때 동일한 두 점의 합과 서로 다른 두 점의 합을 연산하는 도중 k의 비트 값이 '0'이면 서로 다른 두 점의 합을 계산하지 않기 위함이다. 다음은 스칼라 곱셈의 테스트 벡터이다.

P_x :1 3029482d 3632dacb 35723dca 6a84bef1 358ac365
P_y : 2 479bc536 a5927c38 4795bde0 735ac91f 598cfe32
a : 2 5c4beac8 074b8c2d 9df63af9 1263eb82 29b3c967
f : 8 00000000 00000000 00000000 00000000 00000107
k : 6 a05cd513 a05cd513 a05cd513 a05cd513 00000001
Q_x : 4 efd142fa dea0286f 7325fd91 5d0f4617 3c976490
Q_y : 3 14166c4 e448b445 e007da5a 2b4ead5f 88666808

V. 성능분석

[표 1]은 지금까지 발표된 대표적인 하드웨어 구현 예들과 본 논문에서 제시한 스칼라 곱셈기에 대해 필요한 연산의 수를 나타내고 있다. [3]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. [2],[4] 그리고 본 논문에서는 반복 루프에서 역수 계산을 포함하고 있

[표 1] 각 구현사례에 대한 연산의 수

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

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

으며, 이를 계산하는데 필요한 시간비용은 곱셈기만을 이용하여 계산할 경우에 전체 시스템에 큰 부하를 가져온다. 왜냐하면 [표 2]를 보면 알 수 있듯이 [2]는 $2\lceil \frac{m}{82} \rceil \lceil \frac{m}{4} \rceil (m-1)$번의 곱셈연산이 필요하고, [3]은 24번의 곱셈연산이 필요하며, [4]는 $\lfloor \log_2(m-1) \rfloor + HW(m-1) - 1$의 곱셈연산이 필요하기 때문이다. 따라서 역수를 두 번의 곱셈연산 시간에 계산할 수 있는 확장 유클리드 알고리즘을 이용한 역수기를 사용하여 다른 하드웨어보다 역수계산을 수배에서 수십배 빠르게 계산할 수 있다. 따라서 역수계산에 필요한 부하를 줄임으로써 다른 하드웨어보다 빠른 성능을 발휘할 수 있다. 이는 표 [1]에서의 각 연산을 유한체 곱셈의 수로 환산한 표 [2]에서의 스칼라 곱셈에 필요한 전체 클럭수를 보면 알 수 있다.

[표 3]은 각 구현사례들의 성능을 비교한 것으로 [2]가 가장 좋은 성능을 나타내고 있지만, 서버시스템을 목적으로 어느 크기의 field의 스칼라 곱셈을 할 수 있도록 설계하였기 때문에 하드웨어 리소스가 커며 본 논문 역시 [2]와 같은 ASIC Library를 이용한다면 더욱 좋은 성능을 발휘할 수 있다. 이는

앞에서 설명했듯이 [표 2]에서 비교한 전체 스칼라 곱셈을 계산하는데 필요한 클럭수를 보면 알 수 있다. 또한 [3], [4]는 optimal normal basis에서 스칼라 곱셈기를 구현하였다. [3]은 subfield가 존재하는 field를 선택함으로써 성능과 사이즈에서의 장점이 있지만 특정 field에만 국한된다는 단점이 있으며, [4]는 normal basis의 계산상의 장점을 이용하였지만 normal basis는 standard basis보다 하드웨어 구현시 optimal normal basis가 아닐 경우 어려운 단점이 있다.

이러한 성능분석을 통해서 본 논문에서는 standard basis에서 유한체 곱셈기와 역수기를 사용하여 스칼라 곱셈기를 구현하였다. [표 4]는 $GF(2^{163})$에서 IDEC-C631 3.3V Cell Based Library를 사용하여 구현한 스칼라 곱셈기의 성능에 대해서 요약하고 있다. 스칼라 곱셈기를 구현하기 위해 필요한 하드웨어 리소스는 약 6만 gates이고, 최대 140㎒까지 동작할 수 있으며, 이 경우의 성능은 약 64 *kbps*이다. 본 하드웨어의 특징으로 유한체 곱셈기와 유한체 역수기는 규칙적인 구조를 가지고 있으므로 $GF(2^m)$의 m의 크기가 변하더라도 쉽게 확장 혹은 축소할 수 있다. 따라서 본 논문에서 제시하는 하드웨어 구조는 성능이나 사이즈면에서 휴대용 보안장치에 적합하다고 여겨진다.

Ⅵ. 결 론

스칼라 곱셈기는 확장 유클리드 알고리즘을 이용

[표 2] 각 구현사례의 스칼라 곱셈에 필요한 클럭수

구현사례	스칼라곱셈에 필요한 전체 클럭수	하드웨어 구성	$GF(2^m)$상에서 연산(클럭수)
[3]	$20m^2+6m$	곱셈기	곱셈(m) 역수($24m$)
[4]	$4m^2-4m+2T(m-1)$	곱셈기	곱셈(m) 역수($\lfloor \log_2(m-1) \rfloor + HW(m-1)-1=T$)
[2]	$(4m^2-2m-2)\lceil \frac{m}{82} \rceil \lceil \frac{m}{4} \rceil$	82×4 곱셈기	곱셈($\lceil \frac{m}{82} \rceil \lceil \frac{m}{4} \rceil$) 역수($2\lceil \frac{m}{82} \rceil \lceil \frac{m}{4} \rceil (m-1)$)
본 논문	$(m-1)(6\lceil \frac{m}{2} \rceil +8m)$	래딕스-4 곱셈기 역수기	곱셈($\lceil \frac{m}{2} \rceil$) 역수($4m$)

[표 3] 각 구현사례들의 성능분석

구현사례	field	device	gate count	performance
[3]	$GF((2^5)^{31})$	GF155MIC	11k gates	19kbps
[4]	$GF(2^{53})$	Xilinx FPGA XC4044XL	CLB usage 80%	23kbps
[2]	$GF(2^{163})$	0.25㎛ ASIC	165k gates	148kbps
본 논문	$GF(2^{163})$	0.6㎛ ASIC	60k gates	60kbps

[표 4] 하드웨어 리소스

클럭수	• doubling one point, adding two points - 1106 clock • 성능 : $\frac{163}{2212\times162\times11.5\times10^{-9}}=64kbps$		
하드웨어 리소스		곱셈기	역수기
	각 PE	513gates	244gates
전체 리소스	513×21+244×82+레지스터+컨트롤 박스=6만 gates		
비 고	IDEC-C631 0.6㎛, 3.3V Cell Based Library Release에 근거		

한 유한체 역수기와 래딕스-16 곱셈기로 구성되어 있다. 이러한 본 논문의 가장 큰 특징은 기존의 하드웨어는 유한체 곱셈기만을 사용하여 유한체 곱셈, 역수를 계산한 반면, 스칼라 곱셈의 성능에 가장 큰 비중을 차지하는 역수 계산을 유한체 역수기를 사용하여 두 번의 곱셈시간에 계산한 것이다. 유한체 역수기와 곱셈기는 모두 파이프라인 구조를 가지고 있으며 100%의 하드웨어 사용효율을 발휘하기 위해 하드웨어 리소스의 재활용을 통해 절반정도의 하드웨어 리소스로 구현하였다.

이들을 현대전자 0.6㎛ CMOS 공정 라이브러리인 IDEC-C631을 이용하여 구현한 스칼라 곱셈기의 성능은 $GF(2^{163})$에서 64*Kbps*로 163bit 프레임당 2.53㎳ 걸리며 최대 141㎒까지 동작한다. 이러한 성능의 스칼라 곱셈기는 휴대용 보안장치의 보안 코프로세서로 사용하기에 적합하다고 여겨진다. 현재 휴대용 보안장치의 코프로세서로 사용하기 위해 동작속도를 낮추고 유한체 곱셈이나 역수의 계산 수를 더 줄이는 방법에 대한 추가연구가 필요하다. 현재 본 논문의 하드웨어를 최적화시키고, binary method와 좌표변환을 이용하여 더욱 빠른 스칼라 곱셈기를 연구 중에 있다.

참 고 문 헌

[1] Y. Jeong and W. Burleson, "VLSI Array Syn thesis for Polynomial GCD Computation and Application to Finite Field Division", *IEEE Transactions on Circuits and Systems*, pp. 891~897, Dec. 1994.

[2] 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.

[3] G. B. Agnew, R. C. Mullin, and S. A. Vanstone, "Implementation of Elliptic Curve Cryptosystems over F_2^{155}," *IEEE Journal on Selected Areas in Communications*, Vol. 11, No. 5, pp. 804~813, 1993.

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

[5] D. Johnson and A. Memezes, "The Elliptic Curve Digital Signature Algorithm(ECDSA)", Technical Report CORR 99-31, Dept. of C&O, Univ. of Waterloo, Canada, August 1999.

[6] R. Schroeppel, H. Orman, S. O'Malley and O. Spatscheck, "Fast Key Exchange with Elliptic Curve Systems", CRYPTO 1995, pp. 43~56, 1995.

[7] William Stallings, Cryptography and Network Security, Prentice Hall, 1999.

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

[9] Michael Rosing, Implementing Elliptic Curve Cryptography, Manning, 1999.

[10] IDEC-c631 0.6㎛, 3.3V Cell Base Library Release.

[11] IEEE P1363a/D5(Draft Version 5), Standard Specifications for Public key Cryptography: Additional Techniques, August 16 2000.

[12] ANSI X9.62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm(ECDSA), 1998.

[13] P. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," Mathematics of Computation, Vol. 48, pp. 243~264, 1987.

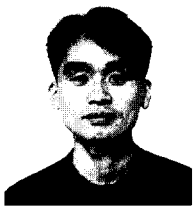
[14] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in GF(2^m) using normal basis", J. Society for Electronic Communication, pp. 31~36, 1986.

[15] S. Morioka, Y. Katayama, "O(log2^m) iterative algorithm for multiplicative inversion in GF(2^m), International Symposium on Information Theory, 2000 Proceedings, IEEE, pp. 449, 2000.

[16] A. V. Dinh, R. J. Palmer, R. J. Bolton, R. Mason, "A low latency architecture for computing multiplicative inverses

and divisions in GF(2^m), Electrical and Computer Engineering, IEEE 2000 Canadian Conference, Vol. 1, pp. 43~47, 2000.

〈著者紹介〉



김 종 만 (Jong-Man Kim)
2000년 2월 : 광운대학교 전자공학부 졸업
2000년 3월~현재 : 광운대학교 전자통신공학과 석사과정
〈관심분야〉 통신용 칩 설계, 무선 통신, 정보보호

김 영 필 (Young-Peel Kim)
1997년 2월 : 유한전문대학교 전자과 졸업
2000년 8월 : 광운대학교 전자공학부 졸업
2000년 8월~현재 : 광운대학교 전자통신공학과 석사과정
〈관심분야〉 통신용 칩 설계, 무선 통신, 정보보호

정 용 진 (Yong-Jin Jeong)
1983년 2월 : 서울대학교 제어계측공학과 졸업
1991년 5월 : 미국 UMASS 전자전산공학과 석사
1995년 2월 : 미국 UMASS 전자전산공학과 박사
1995년 4월~1999년 2월 : 삼성전자 반도체 수석 연구원
1999년 3월~현재 : 광운대학교 전자공학부 조교수
〈관심분야〉 컴퓨터 연산 알고리즘, ASIC 설계, 무선 통신, 정보보호