通信情報保護學會論文誌
第 10 券 第 4 號, 2000. 12

# CRT를 이용한 하이래딕스 RSA 모듈로 멱승 처리기의 구현*

이 석 용**, 김 성 두***, 정 용 진****

# Implementation of High-radix Modular Exponentiator for RSA using CRT

Seok-yong Lee**, Seong-doo Kim***, Yong-jin Jeong****

## 요 약

본 논문에서는 RSA 암호 시스템의 핵심 연산인 모듈로 멱승의 처리속도를 향상시키기 위한 방법으로 하이래딕스(High-Radix) 연산 방식과 CRT(Chinese Remainder Theorem)를 적용한 새로운 하드웨어 구조를 제안한다. 모듈로 멱승의 기본 연산인 모듈로 곱셈은 16진 연산 방법을 사용하여 PE(Processing Element)의 개수를 1/4로 줄임으로써, 기존의 이진 연산 방식에 비해 클럭 수와 파이프라이닝 플립플롭의 지연시간을 1/4로 줄였다. 복호화 시에는 합성수인 계수 N의 인수, p, q를 알고 있는 점을 이용하여 속도를 향상시키는 일반적인 방법인 CRT 알고리즘을 적용하였다. 즉, s비트의 키에 대해, s/2비트 모듈로 곱셈기 두 개를 병렬로 동시 수행함으로써 처리 속도를 CRT를 사용하지 않을 때보다 4배정도 향상시켰다. 암호화의 경우는 두 개의 s/2비트 모듈로 곱셈기를 직렬로 연결하여 s비트에 대한 연산이 가능하도록 하였으며, 공개키 E는 17비트까지의 지수를 허용하여 빠른 속도를 유지하였다. 모듈로 곱셈은 몽고메리 알고리즘을 변형하여 사용하였으며, 그 내부 계산 구조를 보여주는 데이터 종속 그래프(Dependence Graph)를 수평으로 매핑하여 1차원 선형 어레이 구조로 구성하였다. 그 결과 삼성 0.5um CMOS 스탠다드 셀 라이브러리를 근거로 산출할 때, 1024비트 RSA 연산에 대해서 160Mhz의 클럭 주파수로 암호화 시에 15Mbps, 복호화 시에 1.22Mbps의 성능을 가질 것으로 예측되며, 이러한 성능은 지금까지 발표된 국내외의 어느 논문보다도 빠른 RSA 처리 시간이다.

## ABSTRACT

In a methodological approach to improve the processing performance of modulo exponentiation which is the primary arithmetic in RSA crypto algorithm, we present a new RSA hardware architecture based on high-radix modulo multiplication and CRT(Chinese Remainder Theorem). By implementing the modulo multiplier using radix-16 arithmetic, we reduced the number of PE(Processing Element)s by quarter comparing to the binary arithmetic scheme. This leads to having the number of clock cycles and the delay of pipelining flip-flops be reduced by quarter respectively. Because the receiver knows p and q, factors of N, it is possible to apply the CRT to the decryption process. To use CRT, we made two s/2-bit multipliers operating in parallel at decryption, which accomplished 4 times faster performance than when not using the CRT. In encryption phase, the two s/2-bit multipliers can be connected to make a s-bit linear multiplier for the s-bit arithmetic operation. We limited the encryption exponent size up to 17-bit to maintain high speed, We implemented a linear array modulo multiplier by projecting horizontally the DG of Montgomery algorithm. The H/W proposed here performs encryption with 15Mbps bit-rate and decryption with 1.22Mbps, when estimated with reference to Samsung 0.5um CMOS Standard Cell Library, which is the fastest among the publications at present.

**keyword** : *Modular exponentiation, Montgomery algorithm, High-radix, CRT, RSA*

---

* 이 논문은 1999년도 한국학술진흥재단의 연구비 지원(KRF-99-003-E00376)과 1999년도 광운대학교 교내 학술 연구비 지원에 의해 연구되었습니다.

** 광운대학교 전자통신공학과 실시간 구조 연구실 (sylee@explore.gwu.ac.kr)
*** 광운대학교 전자통신공학과 실시간 구조 연구실 (sdkim@explore.gwu.ac.kr)
**** 광운대학교 전자공학부 조교수 (yjjeong@daisy.gwu.ac.kr)

## Ⅰ. 서 론

네트워크의 대규모 확장과 정보 통신 기술의 급격한 발전으로 인터넷이 활성화되었으며, 이를 기반으로 하여 수많은 기업들이 다양한 서비스들을 제공하고 있다. 이 중에서, 상거래의 혁명이라고 일컬어지는 전자 상거래나, 전자문서의 교환 등은 그 특성으로 인하여 인터넷의 전세계적인 개방 환경 속에서도 사용자간의 높은 수준의 보안과 신뢰성이 요구되는 분야이다. 이러한 보안과 신뢰성을 제공하기 위한 다양한 기술들이 연구되고 있지만, 그 중에서도 암호 기술은 경제적인 장점과 효율성으로 인하여 많은 관심을 끌고 있는 기술이다.

암호 기술이란, 전송되는 데이터를 합법적인 사용자가 아닌 제 3자가 읽을 수 없도록 암호화하는 기술과 사용자의 신원을 확인하기 위한 디지털 서명 기술을 말한다. 이 두 가지 기술을 모두 제공하는 메카니즘은 공개키 암호 알고리즘에서 발견할 수 있으며, 이러한 개념은 1976년 Diffie와 hellman[1]에 의해 제안 된 이후 1978년 MIT 대학의 R. Rivest, A. Shamir, L. Adleman에 의해서 실제적인 공개키 암호 시스템으로 등장하였으며, 창안자 이름의 머릿글자를 따서 RSA라 명명되었다.[2]

암호 알고리즘은 암호화와 복호화시에 사용되는 키의 공개 여부 및 특성에 따라 비밀키 암호 방식(대칭키)과 공개키 암호 방식(비대칭키)으로 나뉜다. 비밀키 암호 방식은 암호화할 때의 키와 복호화할 때의 키가 동일한 시스템으로 공개키 시스템보다 암호화 키의 크기가 상대적으로 작기 때문에 훨씬 빠른 속도의 암호 시스템 구축이 가능하지만, 여러 사람과 정보 교환시 한 사람이 많은 수의 키를 관리, 유지해야하는 어려움이 있다. 공개키 암호 방식은 암호화할 때의 키와 복호화할 때의 키가 다른 시스템으로, 키 생성 알고리즘을 이용하여 2 개의 키를 만들어 낸 후, 하나는 전화번호부처럼 공개하고 나머지 하나를 개인키로 자신이 보관하여 사용하는 것이다. 따라서, 키의 관리 측면에서는 비밀키 암호 시스템보다 관리해야 할 키의 수가 적고, 안전한 키 교환의 장점이 있는데 반해, 암호화를 위한 키 크기는 상대적으로 크기 때문에 속도면에서 효과적인 암호 시스템 구축이 어렵다.

공개키 암호 방식의 대표적인 RSA알고리즘은 보안성을 높이기 위해 1024비트 이상의 큰 정수를 키로 사용하며, 암호화 및 복호화가 모듈로 멱승 연산으로 수행된다. 이러한 점이 하드웨어로의 구현시 비밀키 방식에 비해 상대적으로 속도를 느리게 하는 주요 요인이 된다. 따라서 보안성을 높이기 위한 키의 길이는 줄일 수 없더라도 모듈로 멱승 연산을 더 효율적으로 함으로써 전체적인 속도를 높이는 방법이 여러 모로 연구되고 있다.[3~12]

본 논문에서는 이러한 속도 향상을 위한 연구로서, 새로운 하드웨어 구조를 제안한다. 복호화할 때에는 수신자가 합성수 N의 인수를 알고 있다는 특징을 살려 속도를 향상시키기 위해 일반적으로 사용되는 CRT(Chinese Remainder Theorem)방식을 이용함으로써 복호화의 속도를 CRT를 사용하지 않았을 경우보다 4배정도 향상시켰다. 또, 모듈로 멱승을 위해 반복적으로 수행되는 모듈로 곱셈은 16진 연산 방식을 사용하였는데, 이진 방식에 비해 PE의 개수가 1/4로 줄어들어 파이프라이닝 플립플롭 때문에 발생되는 지연시간과 전체적인 클럭 수를 1/4로 줄였다. 또한, RSA의 암호화 지수 E값은 작은 크기를 사용하여도 안전도에 문제가 없으며, 현재의 응용분야에서도 한번 만들어 두면 오래 동안 사용 가능한 디지털 서명은 오래 걸리더라도, 매번 수행하게 되는 서명 검증 과정은 빠르게 할 수 있도록 공개키를 작은 크기의 것으로 사용하는 것이 일반적이다. 따라서 본 논문에서도 CRT를 사용한 복호화에 비해 상대적으로 속도가 느린 암호화에 대해서는 암호화 지수 E의 크기를 17비트로 제한함으로써 전체적으로 빠른 속도를 유지하였다. 그리고, 내부 계산 구조에 있어서는, 모듈로 멱승의 구현시 가장 적합하다고 알려진 몽고메리 알고리즘을 사용하였다. 이의 데이터 종속 그래프의 매핑시, 본 논문에서는 지금까지의 다른 논문들이 사용한 수직 매핑과는 달리, 수평으로의 매핑을 통하여 1차원 선형 어레이 구조로 구현하였다. 이로 인한 장점은 수직 매핑에 비해 단순한 입력 패턴과 출력 패턴으로 쉬운 컨트롤과 100%의 처리율을 가지도록 하였다는 점이다.

본 논문의 구성은 다음과 같다. II장에서는 RSA 알고리즘에 대한 기존의 구현 사례들에 대해 분석하고, III장에서는 RSA 알고리즘에 대해 소개하며, 이를 구현하기 위한 기본 연산 구조인 모듈로 곱셈기를 하이래딕스 몽고메리 모듈로 곱셈 알고리즘을 이용하여 설계하는 과정을 설명한다. 또, 복호화할 때 CRT방식을 이용하기 위한 구조로 각 곱셈기, 레지스터부, 컨트롤 부를 구현하여 전체적인 모듈로 멱승을 수행하는 새로운 하드웨어를 제안한다. IV장에서는

제안된 하드웨어 구조에 대한 성능을 분석하고, 다른 논문과 비교하며, 마지막으로 V장에서 결론을 맺는다.

## II. RSA의 구현 사례

RSA모듈 구현에 대한 기존의 방법들을 살펴보면, 논문 [6]은 모듈로 곱셈을 수행할 때 중간 결과 값에 대한 모듈로 감소 과정에서, 몽고메리 알고리즘이 LSB값에 따라 계수를 더한 것과는 달리, MSB 우선 방식을 사용하였다. 여기서는 미리 계수의 보수들을 계산해서 레지스터에 저장해 놓고 look-up하는 단순한 방식을 사용하였는데, 이 때문에 PE (Processing Element)가 아주 간단해져서 빠른 클럭 주기를 가능하게 한다. 미리 계산된 계수의 보수들을 저장해 놓기위한 추가의 레지스터가 들어가지만, 후처리(post processing)가 필요하지 않기 때문에 적은 양의 계산시에는 몽고메리 알고리즘보다 유리할 수 있다. 그러나, 멱승 계산처럼 연속된 곱셈을 요하는 경우에는 그 장점이 몽고메리 방법에 비해 크게 드러나지는 않는다.

논문 [7]은 몽고메리 알고리즘을 이용하여 모듈로 곱셈을 2차원 평면상에 시스톨릭 어레이(systolic array)로 구현하였다. s비트의 입력에 대하여 처음으로 출력이 나오기 시작할때까지 2s+2의 클럭 수가 소요되지만, 2차원 평면상에 구현된 것이라 그 크기 때문에 실제로 하드웨어로 구현하기는 힘들다.

논문 [8]은 몽고메리 알고리즘을 이용한 멱승구조를 하나의 FPGA (Field Programmable Gate Array)에 들어갈 수 있도록 PE들의 한 줄만으로 매핑하여 선형 어레이 구조로 구현하였다. 입력값 A, B, N에 대한 모듈로 곱셈식 $(R_i+a_iB+q_iN)/2$의 계산을 매번 반복하기 보다는 미리 B + N값을 계산해 놓고 $a_i$와 $q_i$값에 따라서 0, N, B, B+N값을 멀티플렉서를 이용하여 선택하는 방식을 취함으로써 덧셈기를 하나로 줄였다. 그러나, 이를 멱승에 적용시에는 매 곱셈이 끝날 때마다 B+N을 다시 계산해야 한다. 수직매핑으로 인한 50%의 처리율을 극복하기 위해 제곱과 곱셈의 인자를 인터리빙하기 때문에 추가의 레지스터와 콘트롤이 복잡해지는 단점이 있다. s비트의 입력에 대해서 출력이 나오기까지는 2(s+2)(s+4)의 클럭 사이클이 소요된다.

논문 [9]는 속도를 높이기 위한 다양한 방법으로 CRT(Chinese Remainder Therem), Carry Completion Adder, Quotient pipelining을 사용하여 그들이 새로이 제안한 프로그래머블 능동 메모리(PAM: Programmable Active Memory) 구조에 구현하였다. 그러나, RSA 암호화와 복호화시 서로 다른 PAM디자인을 사용하였고, 계수가 곱셈기안에 Hard-wired되어 있어서 계수가 바뀔 때마다 아키텍쳐를 다시 구성해야 하는 단점을 가지고 있다. Carry save형태의 redundant binary representation을 사용하였기 때문에 한번의 곱셈이 끝난 후 다음의 곱셈과정으로 이동하기 위해 non-redundant 형태로 바꾸어야 하는데, 이 과정에서의 딜레이를 줄이기 위해 asynchronous carry completion detection 회로를 사용하였다. CRT를 사용하여 복호화시 1024비트의 키에 대해서 약 165kbps의 속도를 이루어 최근까지 발표된 결과중 가장 좋은 성능을 갖는 것으로 보인다.

## III. 본 론

### 3.1 RSA 암호 알고리즘

RSA 암호 알고리즘은 s비트의 크기를 갖는 키를 기반으로 한 모듈로 멱승 연산에 의해 수행된다. 암호화와 복호화를 위한 키의 생성과정은 다음과 같다.

1) 두 개의 서로 다른 큰 소수 p와 q를 임의로 생성한다.
2) $N=p \cdot q$를 계산한다.
3) N에 대한 Euler's totient function $\Phi(N)=(p-1)(q-1)$일때, $\gcd(E, \Phi(N))=1$인 관계를 갖는 E 값을 $1<E<\Phi(N)$의 수 중에서 임의로 선택한다.
4) $D \cdot E \bmod \Phi(N)=1$인 관계를 갖는 D를 확장된 유클리드 알고리즘을 이용하여 계산한다.
5) E와 N을 공개키로서 공개하며, D와 p, q는 개인키로서 공개하지 않는다.

이 키들을 이용하는 RSA 암호 알고리즘의 암호화 및 복호화 연산은 식 1과 같다. 식 1에서 보는 바와 같이 RSA암호시스템은 공개키 E(암호화 지수)나 개인키 D(복호화 지수)에 대해 모듈로 멱승을 취함으로써 암호화 (C) 및 복호화 (M)가 이루어 지며, 두 과정이 인자만 바뀌고 같은 연산으로 이루어진다. 또한, 디지털 서명을 할 때는 사용자의 개인키 D로 암호화를 하며, 서명 검증을 할 때는 송신자의 공개키 E로 복호화를 수행하게 된다.

$$C = M^E \pmod{N}, \quad where\ E = \sum_{i=0}^{\lfloor \log_2 E \rfloor} e_i \times 2^i$$
$$M = C^D \pmod{N}, \quad where\ D = \sum_{i=0}^{\lfloor \log_2 D \rfloor} d_i \times 2^i \qquad (1)$$

공개키 암호 알고리즘은 두 개의 서로 다른 키를 사용하게 된다. 따라서, 공개키를 알고 있더라도 비밀키를 쉽게 찾아 낼 수 없는 구조를 가지고 있어야 한다. RSA 알고리즘의 경우는 공개키 E, N과 비밀키 D의 관계가 $D \cdot E \bmod \Phi(N) = 1$ 이며, 따라서 $\Phi(N)$을 알게 되면 비밀키는 쉽게 드러나게 된다. $\Phi(N)$을 계산하기 위해서는 N을 p와 q로 정확히 인수분해 해야 하는데, RSA알고리즘의 안전도는 이러한 인수분해의 난이도에 근거를 두고 있다. 이러한 난이도는 N의 비트수가 크면 클수록 증가하게 되는데, 512비트 이상이면 거의 확실한 안전도를 제공하며 1024비트 이상이면 현재까지는 그 안전도에 위협이 될 만한 것이 없다고 알려져 있다.

안전도를 위한 키의 길이가 크기 때문에 RSA의 하드웨어로의 구현은 그 속도 문제가 항상 연구의 초점이 되어 왔으며, 핵심 연산 과정인 모듈로 멱승을 효율적으로 하기 위한 방법 또한 많은 연구가 진행되고 있는 상태이다.

RSA암호 시스템의 처리 속도를 향상시키기 위한 한 가지 방법으로, 복호화할 때에 수신자는 개인키 p, q를 알고 있다는 사실을 이용하여 CRT를 사용하는 방법이 있다.

### 3.2 CRT(Chinese Remainder Theorem)

RSA의 복호화 과정인

$$M = C^D \pmod{N}$$

은 CRT를 이용하여 더 빠르게 수행할 수 있는데, 이 방법은 Quisquater와 Couvreur에 의해 제안된 방법이다.[5]

i=1, 2, ..., k이고, $p_i$ 가 각각 서로 소인 관계를 갖는 정수라고 할 때, 즉,

$$\gcd(p_i, p_j) = 1, \quad i \neq j,$$

$u_i \in \{0, p_i - 1\}$가 주어지면, $P = p_1 p_2 \ldots p_k$인 $\{0, P-1\}$ 범위 내에 $u \equiv u_i \pmod{p_i})$ 인 유일한 u가 존재한다.

또한, 이 유일한 u를 구하는 방법은

$$u = \sum_{i=1}^{k} u_i c_i P_i \pmod{P} \qquad (2)$$

이다. 여기서, $P_i = p_1 p_2 \ldots p_{i-1} p_{i+1} \ldots p_k = P/p_i$ 이고, $c_i$는 계수 $p_i$에 대한 $P_i$의 곱셈에 대한 역원 즉, $c_i P_i \equiv 1 \pmod{p_i}$이다.

위의 내용은 RSA의 복호화 과정인

$$M = C^D \pmod{p \times q}$$

를 다음과 같이 나누어 계산할 수 있음을 나타낸다.

$$M_p = C^D \pmod{p}$$
$$M_q = C^D \pmod{q}$$

이렇게 하여 계수를 s/2비트로 줄일 수 있으며, 여기에 다시 Fermat's theorem을 적용하여, $D_p = d \pmod{(p-1)}$, $D_q = d \pmod{(q-1)}$이라 놓으면,

$$M_p = C^{D_p} \pmod{p}$$
$$M_q = C^{D_q} \pmod{q}$$

와 같이 계산할 수 있다. 또한 $C_p = C \pmod{p}$, $C_q = C \pmod{q}$로 치환하면 모든 인자를 s/2비트로 줄일 수 있다.

$$M_p = C_p^{D_p} \pmod{p}$$
$$M_q = C_q^{D_q} \pmod{q}$$

위와 같이 하여 계산된 $M_p$와 $M_q$를 가지고 복호화된 결과값 M을 구하는 과정은 식 2에 의해,

$$M = M_p c_p^* (pq/p) + M_q c_q^* (pq/q) \pmod{N}$$
$$= M_p c_p^* q + M_q c_q^* p \pmod{N}$$

이며, 여기서, $c_p^* = q^{-1} \pmod{p}$, $c_q^* = p^{-1} \pmod{q}$ 이다. 이의 증명은 다음과 같이

$$M \pmod{p} = M_p \times 1 + 0 = M_p$$
$$M \pmod{q} = 0 + M_q \times 1 = M_q$$

이루어진다.

이와 같은 방법으로 RSA 암호 알고리즘을 하드웨어로 구현한다면, s/2비트인 p와 q에 대한 연산을 병렬로 동시에 수행함으로써, s비트 연산시와 동일한 하드웨어 리소스로 약 4배정도의 속도향상이 있다. 즉, s/2비트 RSA 모듈로 멱승 모듈 2개를 병렬로 사용하여 곱셈에 대해 절반의 클럭수, 멱승에 대해 절반의 클럭수, 전체적으로는 클럭수가 1/4로 줄어들어 속도는 4배 정도 향상된다.

### 3.3 하이래딕스 몽고메리 알고리즘

모듈로 멱승은 연속된 모듈로 곱셈으로 이루어진다. 따라서, 모듈로 멱승을 수행하는 기본 연산은 모듈로 곱셈이며, 모듈로 곱셈기의 설계가 RSA의 하드웨어 구현에 매우 중요한 관건이 된다. 이러한 곱셈 연산을 효율적으로 하기 위해 제안된 방법 중에서 몽고메리가 1985년에 제안한 방법[4]이 가장 효율적이라고 알려져 있는데, 이 알고리즘은 모듈로 연산을 나눗셈으로 처리하지 않고 쉬프트연산과 덧셈연산으로 처리한다. 그 때문에 일반적인 컴퓨터 구조에 알맞은 방법일 뿐 아니라, 하드웨어로의 구현도 쉬워 가장 많이 사용되고 있는 방법이다. 또한 파이프라인 구조로 구현할 경우, 덧셈 연산 시에 각 PE (Processing Element)의 연산 순서와 캐리의 전달 순서가 동일한 장점이 있다.

이러한 몽고메리 알고리즘을 이용한 하이 래딕스 ($2^k$) 모듈로 곱셈식은 식 (3)과 같다.

$$
\begin{aligned}
&S_0 = 0 ;\\
&For\ i = 0\ to\ i = n-1\ do\\
&\quad q_i = ((S_i + b_iA)\ mod\ 2^k \cdot N^*)\ mod\ 2^k ;\\
&\quad S_{i+1} = (S_i + q_iN + b_iA)\ div\ 2^k ;\\
&End ;
\end{aligned}
\qquad (3)
$$

위의 식에서 n을 입력 값의 디지트 수라 할 때, $4N < 2^{kn}$이고, $N^*$은 $-NN^* \pmod{2^k} = 1$ 인 관계를 가지며, $q_i$ 값은 중간 결과 값의 하위 k 비트를 '0'으로 만드는 계수 N의 배수를 결정하는 값이다. 마지막 결과 값 $S_n = A \cdot B \cdot 2^{-kn} \pmod N$은 기수(radix)로 n번 나눈 결과이므로 후처리(Post processing)로 $2^k$를 n 번 곱해야 정확한 결과를 얻게 된다. 즉, $R^* = A \cdot B \cdot 2^{-kn} \pmod N$이므로 후처리로 $R^* \cdot 2^{kn} \pmod N$을 수행하여 원하는 결과 값 $R = A \cdot B \pmod N$을 얻는다.

식 (3)을 그대로 하드웨어로 구현하면 클럭의 주기를 결정하는 최장 지연 패스(Critical path)는 $q_i$값을 결정하는 부분 때문에 훨씬 길어지게 되고, 파이프라인의 구현 시 $q_i$의 결정은 LSB부분에서 이루어지므로 나머지 부분에서는 속도면에서 손해를 보게 된다. 따라서, q값의 결정이 빠르게 이루어질 수 있도록 다음과 같이 몽고메리 알고리즘을 변형하여 사용하고 있다.[9,10]

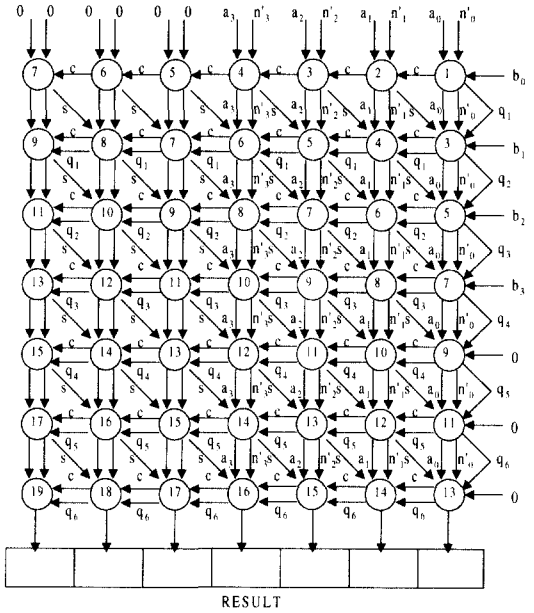
먼저, 식 (3)에서 $N^*$을 $\bar{N} = NN^*$로 바꾸면, $q_i = (S_i + b_iA) \bmod 2^k$가 되어 $N^*$과의 곱셈 과정이 없어지고, 여기에 다시 A를 k-bit 왼쪽으로 쉬프트하여 $2^k \cdot A$를 입력으로 넣으면 B와 곱할 때 LSB는 항상 '0'이 된다는 사실을 이용하여 식 (4)를 유도할 수 있다.

$$
\begin{aligned}
&S_0 = 0 ;\\
&For\ i = 0\ to\ i = n-1\ do\\
&\quad q_i = Si\ mod\ 2^k ;\\
&\quad S_{i+1} = (S_i + q_i\bar{N})\ div\ 2^k + b_iA ;\\
&End ;
\end{aligned}
\qquad (4)
$$

또한, 식 (4)에서 $\bar{N}+1$ 이 $2^k$로 나누어 떨어지는 성질을 이용하면, $S_{i+1} = S_i \text{ div } 2^k + q_i((\bar{N}+1) \text{div } 2^k) + b_iA$로 변형시킬 수 있다. $N' = (\bar{N}+1) \text{ div } 2^k$로 놓으면 식 (5)와 같이 표현 가능하며, 그 결과로 식 (3)의 반복문에서의 $q_i$값의 계산 과정이 제거되었으며, 최장 지연 패스는 단순히 곱셈과 덧셈 1회로 줄어든다.

$$
\begin{aligned}
&S_0 = 0 ;\\
&For\ i = 0\ to\ i = n\ do\\
&\quad q_i = S_i\ mod\ 2^k ;\\
&\quad S_{i+1} = S_i\ div\ 2^k + q_iN' + b_iA ;\\
&End ;
\end{aligned}
\qquad (5)
$$

이렇게 변형된 몽고메리 알고리즘의 내부 계산 구조를 보여주는 데이터 종속 그래프(Dependence Graph: DG)를 키 사이즈 16비트, 16진 연산방식을 예로 들어 [그림 1]에 보였다. 이에 대해 어떤 방향으로 매핑하고, 어떻게 스케쥴링 하느냐에 따라

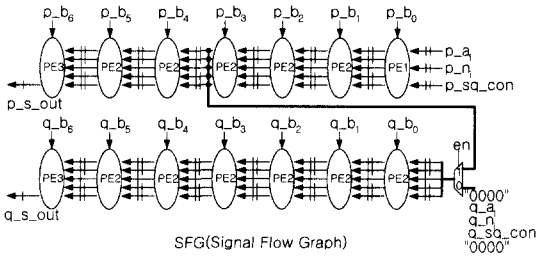



〔그림 1〕 하이래딕스 몽고메리 알고리즘의 DG

다양한 곱셈기 모듈이 구현될 수 있는데, 지금까지의 다른 모든 논문들이 이를 수직으로 매핑한 것과는 달리 본 논문에서는 100%의 처리율과 쉬운 컨트롤을 이루기위해 수평 방향으로 매핑하여 곱셈기를 구성하였다.

〔그림 1〕에서 결과 값이 출력되는 부분인 제일 하단 행에서는 쉬프트가 일어나지 않기 때문에 변형된 몽고메리 알고리즘 적용시 쉬프트 동작은 n번만 일어나게 된다. 〔그림 1〕에 대해, 32비트 RSA연산을 가정하여 CRT를 이용하여 수행할 수 있도록 수평으로 매핑한 SFG(Signal Flow Graph)와 이로부터 유도된 PE를 각각 〔그림 2〕, 〔그림 3-1〕, 〔그림 3-2〕, 〔그림 3-3〕에 보였다.

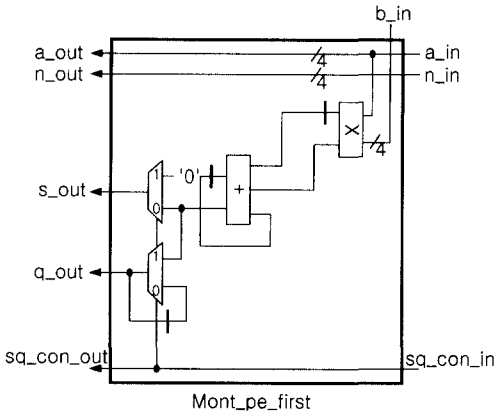
〔그림 2〕는 복호화 시에 상하의 16비트의 곱셈기 모듈이 병렬로 동시에 수행되며, 암호화 시에는 직렬로 연결되어 32비트에 대해 수행되도록 한 것이다.



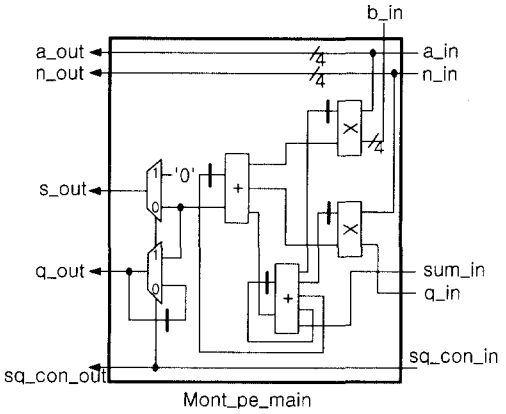

〔그림 2〕 Radix-16 몽고메리 알고리즘의 SFG

MSB쪽으로 3개의 PE가 추가된 것은 III-4절에서 설명될 멱승 시의 입력 조건을 맞추기 위한 것이다.

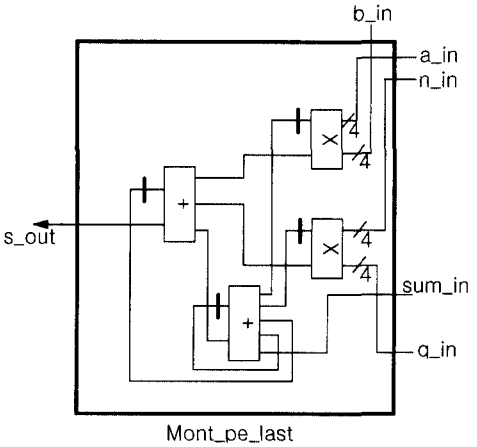
〔그림 3-1〕은 수평으로 매핑시 데이터를 받아들이는




〔그림 3-1〕 Radix-16 몽고메리 알고리즘의 PE1




〔그림 3-2〕 Radix-16 몽고메리 알고리즘의 PE2




〔그림 3-3〕 Radix-16 몽고메리 알고리즘의 PE3

제일 우측에 위치하고, 그림 3-3은 결과값을 순차적으로 내보내는 제일 좌측에 위치한다. [그림 3-2]는 그 사이에 반복적으로 인스턴스되는 PE이며, sq_con_ in 신호는 [그림 1]의 제일 오른쪽 열에서 sum값과 $q_i$ 값을 선택해 주는 신호이다.

[그림 3-2]에서 곱셈기의 지연 시간과, 이전 스텝에서 쉬프트된 sum_in 값과 캐리 값들을 더하는 덧셈기의 지연 시간은 동일하며, 각 곱셈기와 덧셈기의 캐리 부분에는 플립플롭이 존재하기 때문에 PE 전체에 대한 최장 지연 패스는 곱셈기와 4비트 3-입력 덧셈기의 sum값이 계산되는 패스와 멀티플렉서로서, 4⊿FA+1⊿HA+1⊿MUX의 지연 시간이다.

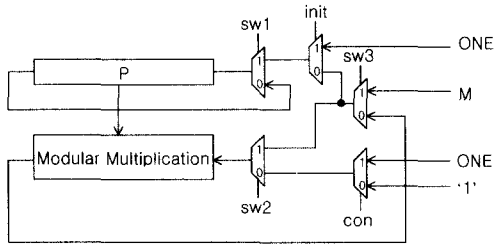
하이래딕스($2^k$)를 사용한 PE는 이진 연산의 구조에서 k 비트를 하나의 블록으로 묶은 것이기 때문에 곱셈 연산에 대한 클럭 주기는 늘어난다. 하지만, 전체 클럭수가 1/k 로 줄어들고 파이프라이닝 플립플롭의 감소에 따른 지연시간 축소로 전체적인 처리 시간이 줄게 된다.

### 3.4 모듈로 멱승

RSA 암호 알고리즘의 핵심이 되는 모듈로 멱승 연산은 곱셈의 연속으로 수행시 계산량이 너무 많아지게 된다. 계산량을 줄이기 위한 방법으로 지수(exponent)에 따라 일련의 제곱과 곱셈으로 멱승을 수행하는 방법이 있는데, 그 중 하나인 binary method는 식 6과 같이 지수를 스캔하는 방향에 따라 LR-method (Left-to-Right)와 RL-method (Right-to-Left)가 있다.

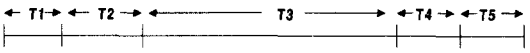
[그림 1]을 수평으로 매핑한 경우, 입력이 n+1 클럭에 걸쳐 들어간 후 결과가 2(n+1) 부터 나오기까지 n+1 클럭을 쉬기 때문에 이를 이용하기 위하여 제곱과 곱셈이 서로 독립적으로 병렬 처리될 수 있는 RL-Method를 이용하여 멱승을 구현하였다. 이를 하드웨어로 구현하기 위한 블럭도를 [그림 4]에 보였다. S는 제곱을 위한 구간이며, M은 곱셈을 위한 구간으로 각각 n+1클럭의 구간이다. 또, 표에 있는 'X'는 don't care를 의미하고, pre, init, main, post는 [그림 5]의 프로세스 이름으로서, 앞으로 설명될 내용이다.

[그림 4]에서 보는 바와 같이 모듈로 곱셈 블록을 통과한 결과 값이 연속해서 입력으로 피드 백 됨을 알 수 있다. 따라서, 모듈로 곱셈기를 이용하여 멱승을 구현할 때 곱셈기의 입력 값의 범위와 출력 값의




| | pre | | init | | main | | | | post | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | E = 1 | | E = 0 | | | |
| | S | M | S | M | S | M | S | M | S | M |
| sw1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | X | 1 |
| init | 1 | X | 0 | 0 | 0 | 0 | 0 | 0 | X | 0 |
| sw2 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | X | 0 |
| con | X | 0 | X | 1 | X | X | X | 1 | X | 0 |
| sw3 | 1 | X | 1 | X | 0 | 0 | 0 | X | X | 0 |

[그림 4] RL-method의 구현




*T1 : preprocessing*

$Cp^* = MonPro(Cp, 2^{2kn}) = Cp \cdot 2^{kn} \bmod p$

$1^* = MonPro(1, 2^{2kn}) = 2^{kn} \bmod p$

*T2 : init processing*

$S = MonPro(Cp^*, Cp^*) = Cp^2 \cdot 2^{kn} \bmod p$

$P = MonPro(1^*, Cp^*) = Cp \cdot 2^{kn} \bmod p$

*T3 : main processing*

$S = MonPro(S, S)$

$P = MonPro(S^*, P)$ $\quad \begin{cases} S^* = S & when\ e_i = 1 \\ S^* = 1^* & when\ e_i = 0 \end{cases}$

*T4 : post processing*

$P = MonPro(P, 1)$

*T5 : serial result out*

[그림 5] RSA 연산의 전체 프로세스

범위를 잘 조절해 줄 필요가 있는데, 이는 출력 값이 입력 값보다 커지게 되면 피드백됨에 따라 점차 출력 값의 크기가 늘어나기 때문이다. 식 (4)에서 초기 입력 값 A, B를 A, B〈2N~이라 하면, 결과 값은 $S_n$이 아닌 $S_{n+1}$에서 2N~보다 작아지게 된다. 따라서 A, B〈2N~이라는 조건에서 2개의 디지트가 추가되며 $S_{n+1}$이 결과 값이 되므로 전체적으로 3개의 디지트를 추가함으로써 입력 조건을 만족하게 된다. 이 때문에 [그림 2]에 3개의 PE가 추가된 것이다.

*Approach* 1 (*LR method*)

*If* $e_{s-1}=1$ *then* $C=M$ *else* $C=1$

for $i=s-2$ *downto* 0

$C=C*C \bmod N$ $\quad \cdots\cdots(i)$

if $e_i=1$ *then* $C=C*M \bmod N$ ……(*ii*)
*return* $C$

*Approach* 2 (*RL method*)
$C=1$ ; $P=M$ ;
*for* $i=1$ *to* $s-2$
if $e_i=1$ *then* $C=C*P \bmod N$ ……(*iii*)
$P=P*P \bmod N$ ……(*iv*)
if $e_{s-1}=1$ *then* $C=C*P \bmod N$
*return* $C$ (6)

몽고메리 모듈로 곱셈식은 MonPro(A, B)=A·B·$2^{-kn}$ mod N 이므로 올바른 값을 얻기 위해서는 멱승 도중에 값을 보정해야 하는 오버헤드가 존재하게 된다. 그러나, 이를 중간 결과 값들에 대해 매번 수행하게 되면 매우 비효율적이고 속도 또한 느려지게 된다. 따라서, 일반적으로 초기 입력 값을 $2^{2kn}$으로 미리 곱해 놓고 계산하는 것이 효율적인데, A=A·$2^{2kn}$ (mod N), B=B·$2^{2kn}$ (mod N)

/*($z=r^2$ (*mod* $p$)) *is precalculated* */
*function MonExp*($C_p$, $D_p$, $p$) {$r=2^{kn}$}
$C_p^*=C_p \cdot r$ (*mod* $p$)$=MonPro(C_p, z)$
$X_p^*=1 \cdot r$ (*mod* $p$)$=MonPro(1, z)$
*for* $i=0$ *to* $s-2$ *do*
*if* $e_i=1$ *then* $X_p^*=MonPro(C_p^*, X_p^*)$
$C_p^*=MonPro(C_p^*, C_p^*)$
if $e_{s-1}=1$ *then* $X_p^*=MonPro(C_p^*, X_p^*)$
$X_p=MonPro(X_p^*, 1)$
*return* $X_p$ (7)

이라 할 때, 모든 중간 결과 값은 항상 R=MonPro($A^*$, $B^*$)=A·B·$2^{kn}$ mod N 처럼 $2^{kn}$의 factor 를 가지게 되므로 멱승의 최종 결과 값에 간단히 1을 곱하는 후처리로써 원하는 결과 값을 얻게 된다. 이에 대한 모듈로 멱승 계산을 식 (7)에 보였다. 식 (7)은 CRT 적용시, 하나의 s/2비트 모듈로 멱승 모듈에 관해 기술한 것이지만, 암호화 시에 직렬로 연결되는 s비트 모듈에 대해서도 마찬가지로 적용된다.

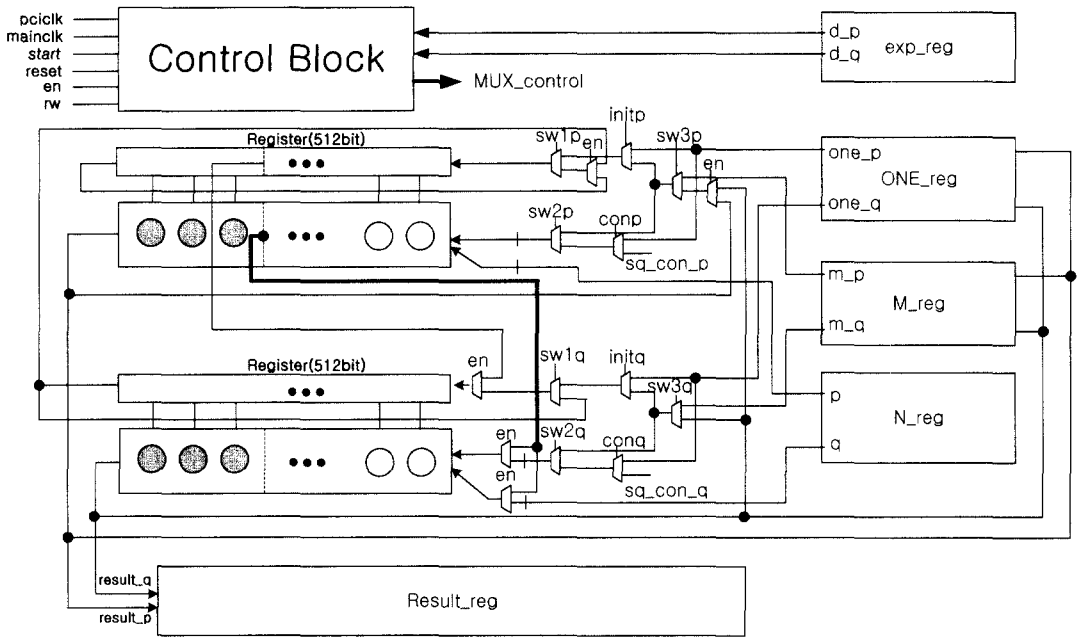
본 논문에서 제안하는 [그림 2]의 새로운 선형 모듈로 곱셈기를 기본으로 하여 식 (7)의 멱승 계산을 효율적으로 처리하기 위한 모듈로 멱승의 과정을 입력 값을 받아서 결과가 출력될 때까지의 전체 RSA 연산 프로세스(Process)로써 각 구간별로 나누면 [그림 5]와 같다. T1 동안에는 각 키 데이터들을 레지스터에 로드하고 식 (7)에서 기술한 것처럼 초기 입력 데이터에 $2^{kn}$의 factor를 붙이는 전처리 과정이고, T2는 전처리의 결과 값을 다시 레지스터에 시리얼로 로딩하고, $e_0$에 대한 계산을 하는 프로세스이다. T3는 주된 연산인 모듈로 멱승을 수행하고, T4는 후처리로 원하는 결과 값을 얻어내기 위해 최종 결과 값에 1을 곱하는 과정이며, 그 결과가 T5에 시리얼로 나오게 된다. 여기서 주목할 것은 암호칩의 계산 모듈이 쉬지 않고 거의 100%의 처리율로 계속해서 데이터를 받아 처리한다는 점이다.

각각의 프로세스는 2(n+1)의 클럭을 소요하고, main프로세스 구간의 경우는 지수의 비트수를 s라고 할 때, 2(n+1)(s-1)의 클럭을 소요한다.

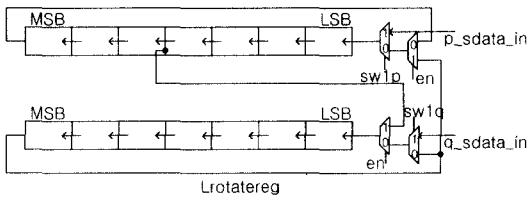
지금까지 상술한 매핑과정과 프로세스에 따라 구현한 멱승기의 전체구조는 그림 6과 같다. 중간 부분의 원 모양은 [그림 2]에서 구현한 PE의 집합으로서, RSA의 연산 코어부분이다. 키 값들을 저장하고, RSA 연산 시에 이 값들을 하위 디지트부터 연산 코어로 제공하는 역할을 하는 레지스터의 구현은 다음과 같다.

1) 복호화할 때는 s/2비트로 나뉘어 병렬로 동시에 동작되고, 암호화할 때는 s비트로서 직렬로 연결되어 동작한다.
2) 암호화 할 때는, p_reg부분이 LSB부분에 해당하며, q_reg부분이 MSB부분에 해당한다.
3) exp_reg를 제외한 모든 레지스터는 16진 모듈로 곱셈기에 데이터를 제공하기 위해 4비트씩 쉬프트 동작을 한다.

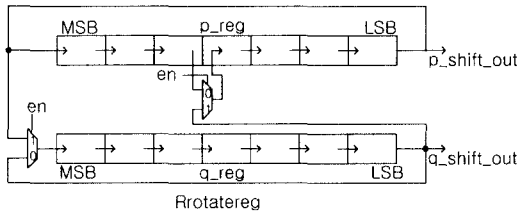
[그림 4]에서 P레지스터의 역할을 하는 것이 Lrotatereg 로서, 식 (6)의 (iii), (iv)의 두 구간 동안 값을 유지하고 있어야 하기 때문에 [그림 7-1]과 같이 순환 쉬프트하도록 구현하였다. 즉, (iii)의 구간에는 시리얼로 출력되는 데이터를 받아들이고, (iv)구간에는 순환 쉬프트 동작을 한다. 복호화시의 p, q와 암호화시의 N값은 RSA연산도중 그 값이 바뀌지 않으므로 순환 쉬프트 레지스터로 구현하였으며, [그림 7-2]에 나타내었다.



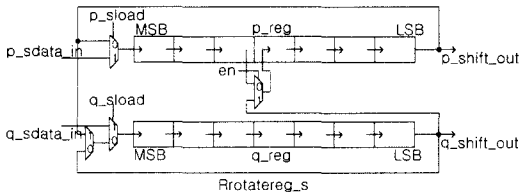

〔그림 6〕 구현된 RSA 전체 연산 구조



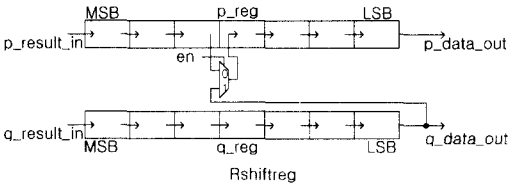

〔그림 7-1〕 내부 레지스터의 구현



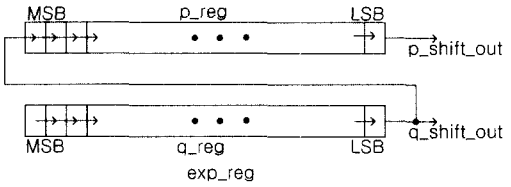

〔그림 7-2〕 N 레지스터의 구현




〔그림 7-3〕 M, ONE 레지스터의 구현

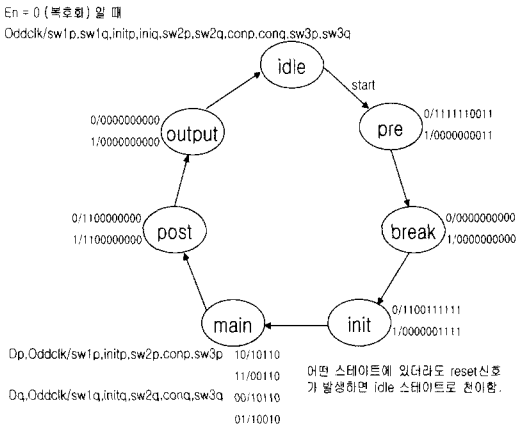



〔그림 7-4〕 Result 레지스터의 구현

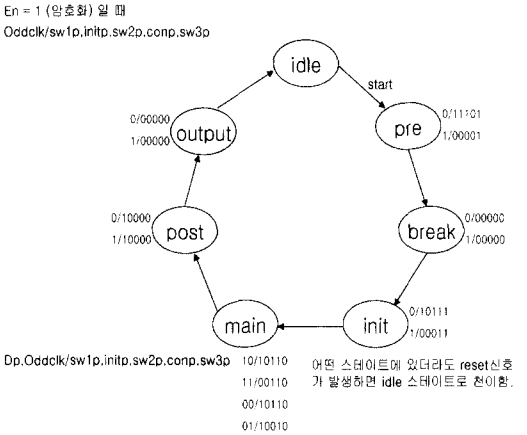



〔그림 7-5〕 지수 레지스터의 구현

M_reg와 ONE_reg는 각각 초기 입력값으로 $C_p$, $C_q$와 $2^{2kn} \pmod p$, $2^{2kn} \pmod q$ 값을 저장하고 있다가 〔그림 5〕의 T1구간에서 계산된 $C_p \cdot 2^{kn} \pmod p$, $C_q \cdot 2^{kn} \pmod q$와 $2^{kn} \pmod p$, $2^{kn} \pmod q$ 값을 다시 받아야 하므로 〔그림 7-3〕과 같이 순환 쉬프트 동작과 함께 중간에 시리얼로 데이터를 받아들일 수 있도록 구현하였다. 결과값을 저장하는 레지스터인 〔그림 7-4〕의 Rshiftreg는




〔그림 8〕 복호화할 때 컨트롤 블록의 FSM




〔그림 9〕 암호화할 때 컨트롤 블록의 FSM

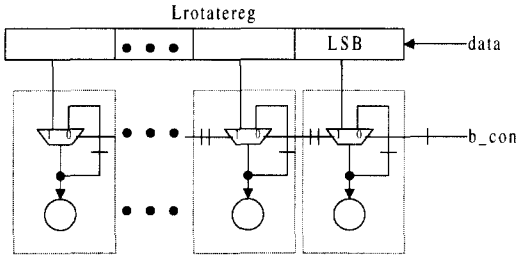
컨트롤 블록에서 발생되는 output_capture신호를 이네이블(enable) 신호로 하여 시리얼로 쉬프트 동작을 하며 최종적인 결과 값을 받도록 구현하였다. 지수를 저장하는 〔그림 7-5〕의 exp_reg는 비트단위로 쉬프트를 수행하게 된다.

컨트롤 블록은 암호화 또는 복호화 신호에 맞추어 레지스터 및 RSA 연산 코어를 〔그림 5〕의 프로세스대로 제어하기 위해 〔그림 4〕의 멀티플렉서를 제어하는 신호를 발생한다. 각 스테이트는 2(n+1)의 클럭동안 유지되며, main 스테이트의 경우는 지수 값이 모두 스캔될 때까지 유지된다.

클럭을 세는 카운터를 따로 두어 n+1의 클럭이 지날때마다 새로운 클럭 lclk과 oddclk을 토글하며 발생한다. 〔그림 8〕, 〔그림 9〕의 FSM(Finite State Machine)은 lclk에 동기하여 동작하며, main 스테이트에서는 oddclk에 따라 제곱과 곱셈을 반복하게 된다. output_capture 신호는 output 스테이트에서 oddclk 이 1인 구간에 발생되어 연산 코어에서 시리얼로 나오는 결과를 result 레지스터가 받아 들이도록 한다. 또한, 각 레지스터를 선택하는 어드레스 디코딩도 컨트롤 블록에서 처리한다.

구현한 멱승 모듈은 중간의 곱셈 결과 값이 시리얼로 나오고 다음 계산을 위한 입력으로도 시리얼로 들어가게 된다. 그러나, Lrotatereg에서 PE로 병렬 입력되는 데이터는 그림 1의 DG에서 보는 것처럼 1,3,5,7,..의 시간에 각 PE에 도착하여 n+1 클럭 동안 값을 유지해야 하기 때문에 〔그림 10〕처럼 멀티플렉서를 사용하여 구현하였다.

또한, b_con의 경우는 "100···0"으로 sq_con_in 신호와 동일한 시퀀스의 비트열이며, 후처리시 사용

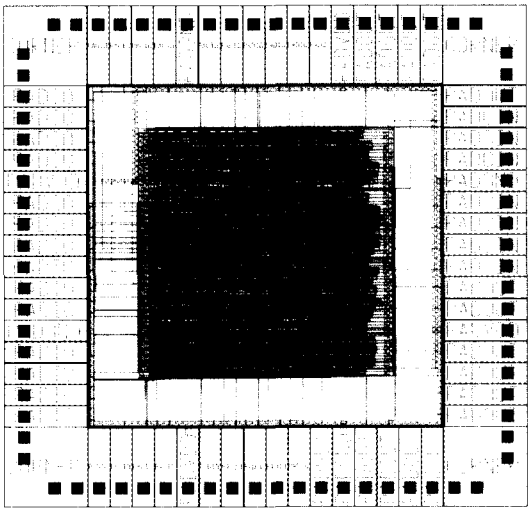



〔그림 10〕 Serial to parallel 의 구현

되는 '1'의 값도 동일하기 때문에 sq_con_in을 각각에 그대로 사용하였다.

## Ⅳ. 성능 분석 및 비교

지금까지 제안한 하드웨어 구조는 1024 비트의 키 사이즈에 대해, ASIC으로 구현을 위해서 Verilog 언어를 사용하여 기술한 후, Synopsys에서 합성하고 Verilog XL로 시뮬레이션을 마친 상태이다. 또, IDEC MPW공모전에 참가하기 위해 32비트로 축소한 모듈에 대해 0.6um 공정으로 Layout한 그림을 〔그림 11〕에 추가하였다. 이는 IDEC C631 0.6um 스탠다드 셀 라이브러리를 이용하여 Synopsys에서 최적화한 후, Mentor의 IC Station을 이용하여 Layout한 것이다. 32비트의 키 사이즈에 대해서 예상 성능은 area 면에서 15600 게이트, 최장 지연 패스는 9.86ns였는데, 실제 구현후의 결과는 16200 게이트, 11.2ns였다. 이는 제안된 하드웨어 구조의 확장성과 규칙성 때문에 1024비트에 대해서도 본 논문에서의 성능 예상 값이 실제 구현후의 성능과 거의 동일할 것임을 의미한다.



〔그림 11〕 Layout Plot

구현된 하드웨어는 PC와 인터페이스하기 위해 데이터를 전송하고 수신하는데 사용하는 PCI(Peripheral Component Interconnect) 클럭과 RSA 연산 중에 사용하는 메인 클럭, 두 개의 클럭을 사용하였다. n을 각 키의 디지트 수라고 할 때, 한번의 모듈로 멱승을 수행하는데, 복호화할 때는 $8n^2$-n-9의 클럭이 소요되고, 암호화할 때는 41n+41의 클럭이 소요된다.

구현된 하드웨어에 대해 0.5um ASIC으로 구현할 경우를 가정하여, 삼성 스탠다드 셀 라이브러리[13]를 근거로 하드웨어 리소스 및 성능을 〔표 1〕에서 분석하였다. 이는 IDEC C-631 0.6um 라이브러리를 이용하여 실제 합성한 결과를 기준으로 예상한 수치이다. 1024비트의 키 사이즈에 대해, 게이트 카운트는 2 입력 nand 게이트를 기본 단위로 계산하여 약 20만 게이트를 차지한다. 암호화 시에는 암호화 지수 E값을 17비트로 제한하였기 때문에 클럭 주파수 160Mhz에서 15Mbps의 성능을 보이며, 복호화 시에는 동일한 클럭 주파수에서 약 1.22Mbps의 성능을 나타낸다. 이와 같은 성능은 몽고메리 곱셈 알고리즘을 수평 매핑하고, 곱셈기를 구현할 때 q에 대한 계산이 필요없도록 알고리즘을 변형하여 사용하였기에 가능한 것이었으며, 복호화 및 서명 시에 CRT를 적용하였기 때문이다.

본 논문에서 구현한 하드웨어 구조의 우수성을 입증하기 위해 지금까지 실제 하드웨어로 구현된 논문[8,9]와 성능을 비교하여 〔표 2〕에 보였다. 지금까지는 논문 〔9〕의 성능이 가장 빠르다고 알려져 있는데,

〔표 1〕 제안된 구조의 성능 분석

| | |
|---|---|
| Clock cycles | Decryption : $8n^2 - n - 9$<br>Encryption : 41n + 41 |
| Critical path delay | 6.166 ns<br>1Δ F/F + 2ΔMUX + 1 Δ AND<br>+ 4ΔFA + 1ΔHA + Δsetup |
| Performance | Decryption : 1.22 Mbps<br>Encryption : 15 Mbps<br>at 160Mhz |
| # of PE | n+1 |
| PE complexity (gates) | 548<br>50 F/F,12 MUX, 32 FA, 12 HA, 32 AND |
| External reg. & control (gates) | 54000<br>k(n+1) bit Register * 6<br>State Machine * 1 |
| Total area (gates) | 200,000 |

〔표 2〕 성능 비교

| | | 512비트 | | | 1024비트 | | |
|---|---|---|---|---|---|---|---|
| | | 수행시간 (ms) | 속 도 (Mbps) | 주파수 (Mhz) | 수행시간 (ms) | 속 도 (Mbps) | 주파수 (Mhz) |
| 암호화 | 논문 [8] | 0.35 | 1.46 | 56 | 0.75 | 1.37 | 52 |
| | 본 논문 | 0.034 | 15 | 160 | 0.066 | 15 | 160 |
| 복호화 | 논문 [8] | 2.37 | 0.216 | 56 | 10.18 | 0.1 | 52 |
| | 논문 [9] | 0.85 | 0.6 | 40 | 6.2 | 0.165 | 40 |
| | 본 논문 | 0.22 | 2.38 | 160 | 0.83 | 1.22 | 160 |

〔표 2〕에 나타낸 것처럼 본 논문의 하드웨어가 논문 〔9〕에 비해 약 7배 정도 빠른 성능을 가진다. 〔8〕과 〔9〕는 각각 PAM과 FPGA에 구현된 것이라 ASIC으로 구현한다면 〔표 2〕의 성능보다는 더 높은 성능이 나오리라고 예상 된다.

본 논문에서 사용한 몽고메리 알고리즘 DG의 수평 매핑과 수직 매핑의 차이점은 다음과 같다. 수직 매핑의 경우는 제곱 또는 곱셈을 위해 시리얼로 입력되는 데이터가 1, 3, 5, 7...의 순서이기 때문에 처리율을 높이기 위해서는 사이사이에 곱셈 또는 제곱의 입력값을 인터리빙 하여 넣어 주어야 한다. 또한, 결과 값의 경우는 〔표 3〕에서 보이는 바와 같이 한 클럭씩 지연되면서 오버랩되어 출력되기 때문에 추가의 레지스터가 필요하다. 따라서, 수평매핑은 입출력 데이터를 컨트롤하기 어려우며, 100% 처리율도 이루기 힘들다. 반면, 수평 매핑의 경우는 제곱과 곱셈의 입력이 순차적으로 들어가고, 결과 또한 순차적으로 출력되기 때문에 컨트롤이 쉬워져 100%의 처리율을 쉽게 이룰 수 있다. 수평 매핑으로 인해 파이프라이닝 플립플롭의 수가 상대적으로 증가하지만 이는, 수직매핑으로 인한 레지스터의 추가에 따른 하드웨어 리소스의 증가분과 거의 비슷한 정도이다.

〔표 3〕 수직 매핑과 수평 매핑의 데이터 흐름 비교

| | 데이터 입력* | 데이터 출력** |
|---|---|---|
| 수평 매핑 | O O ... O X X ... X | $S_0 S_1 S_2 S_3 \ldots S_k M_0 M_1 M_2 M_3 \ldots M_k$ |
| 수직 매핑 | O X O X ... O X | $S_0 S_1 S_2 S_3 \ldots S_k$<br>$M_0 M_1 M_2 M_3 \ldots M_k$ |

* O : 제곱을 위한 입력 X : 곱셈을 위한 입력

** $S_i$ : 제곱 결과의 i 번째 디지트 $M_i$ : 곱셈 결과의 i 번째 디지트

위의 성능 분석은 삼성 0.5um 스탠다드 셀 라이브러리를 근거로 계산한 것이기 때문에 이를 0.25 내지는 0.18um 테크놀러지에 적용한다면 성능은 훨씬 더 향상될 것이다.

## V. 결 론

본 논문에서는 RSA알고리즘의 하드웨어 구현에 있어서 기존의 이진 연산 방법[11] 보다 빠른 성능을 이루기 위해 하이래딕스 방식의 몽고메리 알고리즘과 복호화 시에 CRT방법을 적용하여 구현하였다.

하이래딕스 연산 방식을 사용함으로써, 클럭 수가 줄어들지만, PE 내부의 가산기의 입력 비트 수가 늘어남에 따라 최장 지연 패스가 길어져 클럭 주파수는 낮아진다. 그러나, 이진 연산 방식에 비해 PE 사이의 파이프라이닝 플립플롭의 수를 줄임으로써 속도향상을 이루었다. 래딕스를 올리면 올릴수록 클럭 수 및 파이프라이닝 플립플롭의 수는 점점 줄어들지만 그 내부의 최장 지연 패스와의 trade-off 때문에 16진 연산을 하도록 구현하였다.

사용자가 합성수 N의 인수를 알고 있는 복호화의 경우에는 CRT를 적용하여 s/2비트의 RSA 처리 모듈을 병렬로 동시에 수행함으로써 CRT를 사용하지 않을 경우 보다 4배 정도의 속도 향상을 이루었다. 복호화에 비해 상대적으로 속도가 느린 암호화에 대해서는 공개키 E 값을 17비트 이하로 제한하여 사용함으로써 전체적으로 빠른 속도를 유지하였다. E 값의 크기는 작더라도 보안에는 아무 문제가 없으며,공개되는 값이고, 일반적으로 서명 검증을 빠르게 할 수 있기 때문에 일반적으로 작은 크기의 지수 값을 사용한다. 또한 컨트롤 신호 하나로 암호화와 복호화를 스위칭할 수 있도록 레지스터와 RSA연산 코어를 유연성있게 구성하였으며, 그 구조의 규칙성과 모듈성 때문에 VLSI로의 구현이 용이하다.

구현한 하드웨어의 RSA 연산 처리 속도는 1024 비트에 대해서, 160Mhz의 클럭 주파수로 암호화시 15Mbps, 복호화시 1.22Mbps이며, 이는 현재까지 실제 하드웨어로 구현된 국내외의 다른 어느 논문보다도 우수한 성능이다.

향후로는 본 논문에서 제안한 하드웨어 구조를 ASIC칩으로 제작하고, 분할(Partitioning)을 통해 하드웨어 리소스 및 파워소모를 줄임으로써 스마트 카드에도 응용할 예정이다.

## 참 고 문 헌


〔1〕 W. Diffie, M. E. Hellman, "New Directions in Cryptography," *IEEE Trans. Info. Theory*, Vol. IT-22, No.6, Nov 1976, pp. 644~654.

〔2〕 R. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications on the ACM*, 21(2) : February 1978, pp. 120~126.

〔3〕 Cetin Kaya Koc, "RSA Hardware Implementation", *RSA Laboratories*, August 1995.

〔4〕 P. Montgomery, "Modular multiplication without trial division", *Mathmatics of computation*, Vol. 44, pp. 519~521, 1985.

〔5〕 J. Quisquater, C. Couvreur, "Fast Decipherment Algorithm for RSA Public-key Cryptosystem," *Electronics Letters*, 18, pp. 905~907, October 1982.

〔6〕 Y.Jeong and W.Burleson, "VLSI array algorithms and architectures for RSA modular multiplication", *IEEE Tran. On VLSI Systems*, Vol. 5, pp. 211~217, June 1997.

〔7〕 Colin Walter, "Systolic modular multiplication", *IEEE Tans. On Computers*, Vol. 42, March 1993.

〔8〕 Thomas Blum, Christof Paar, "Montgomery Modular Exponentiation on Reconfigurable Hardware," *IEEE Symposium on Computer Arithmetic*, April 14-16, 1999, Adelaide, Australia.

〔9〕 M. Shand and J. Vuillemin, "Fast implementations of RSA cryptography," *Proceedings 11th IEEE Symposium on Computer Arithmetic*, 1993, pp. 252~259.

[10] H.Orup, "Simplifying quotient determination in high-radix modular multiplication," *Proceedings 12th Symposium on Computer Arithmetic*, pp. 193~199, 1995.
[11] S.Lee, Y.Jeong, "A High Performance RSA Modular Exponentiator with Pipelining," *KISS 2000 Spring Conference*, 2000.
[12] S.Lee, S.Kim, Y.Jeong, "A High-Radix RSA modular Exponentiator with CRT," *KICS 2000 Summer Conference*, 2000.
[13] Samsung Electronics, "ASIC STD85/STDM85 0.5um High Density CMOS Standard Cell Library," *Samsung electronics*, September 1997

## 〈著者紹介〉

**이 석 용 (Seok-Yong Lee)**
2000년 2월 : 광운대학교 전자공학부 졸업
2000년 3월~현재 : 광운대학교 전자통신공학과 석사과정
〈관심분야〉 통신용 칩 설계, 무선 통신, 정보보호

**김 성 두 (Seong-Doo Kim)**
2000년 2월 : 서울산업대학교 전자공학과 졸업
2000년 3월~현재 : 광운대학교 전자통신공학과 석사과정
〈관심분야〉 통신용 칩 설계, 무선 통신, 정보보호

**정 용 진 (Yong-Jin Jeong) 정회원**
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