情報保護學會論文誌
第12券 第2號, 2002. 4

# 나눗셈 체인을 이용한 RSA 모듈로 멱승기의 구현*

김 성 두**, 정 용 진***

# Implementation of RSA modular exponentiator using Division Chain

Seong-doo Kim**, Yong-jin Jeong***

## 요 약

본 논문에서는 최근 발표된 멱승방법인 나눗셈 체인을 적용한 새로운 모듈로 멱승기의 하드웨어 구조를 제안하였다. 나눗셈 체인은 제수(divisor) $d$=2 또는 $d=2^l+1$ 과 그에 따른 나머지(remainder) r 을 이용하여 지수 $E$를 새롭게 변형하는 방법으로 전체 멱승 연산이 평균 약 $1.4\log_2 E$ 번의 곱셈으로 가능한 알고리즘이다. 이것은 Binary Method가 하드웨어 구현 시 항상 worst case인 $2\log_2 E$의 계산량이 필요한 것과 비교할 때 상당한 성능개선을 의미한다. 전체 구조는 파이프라인 동작이 가능한 선형 시스톨릭 어레이 구조로 설계하였으며, DG(Dependence Graph)를 수평으로 매핑하여 $k$ 비트의 키 사이즈에 대해 두 개의 $k$ 비트 프레임이 $k/2+3$ 개의 PE(Processing Element)로 구성된 두 개의 곱셈기 모듈을 통해 병렬로 동시에 처리되어 100% 처리율을 이루게 하였다. 또한, 규칙적인 데이터 패스를 가질 수 있도록 나눗셈체인을 새롭게 코딩하는 방법을 제안하였다. ASIC 구현을 위해 삼성 0.5um CMOS 스탠다드 셀 라이브러리를 이용해 합성한 결과 최장 지연 패스는 4.24ns로 200MHz의 클럭이 가능하며, 1024비트 데이터 프레임에 대해 약 140Kbps의 처리속도를 나타낸다. 복호화 시에는 CRT(Chinese Remainder Theorem)를 적용하여 처리속도를 560Kbps로 향상시켰다. 전자서명의 검증과정으로 사용되기도 하는 암호화 과정을 수행할 때 공개키 $E$는 3,17 혹은 $2^{16}+1$의 사용이 권장된다는 점을 이용하여 $E$를 17 비트로 제한할 경우 7.3Mbps의 빠른 처리속도를 가질 수 있다.

## ABSTRACT

In this paper we propos a new hardware architecture of modular exponentiation using a division chain method which has been proposed in [2]. Modular exponentiation using the division chain is performed by recoding an exponent $E$ as a mixed form of multiplication and addition with divisors $d$=2 or $d=2^l+1$ and respective remainders $r$. This calculates the modular exponentiation in about $1.4\log_2 E$ multiplications on average which is much less iterations than $2\log_2 E$ of conventional Binary Method. We designed a linear systolic array multiplier with pipelining and used a horizontal projection on its data dependence graph. So, for $k$-bit key, two $k$-bit data frames can be inputted simultaneously and two modular multipliers, each consisting of $k/2+3$ PE(Processing Element)s, can operate in parallel to accomplish 100% throughput. We propose a new encoding scheme to represent divisors and remainders of the division chain to keep regularity of the data path. When it is synthesized to ASIC using Samsung 0.5 um CMOS standard cell library, the critical path delay is 4.24ns, and resulting performance is estimated to be about 140 Kbps for a 1024-bit data frame at 200Mhz clock. In decryption process, the speed can be enhanced to 560Kbps by using CRT(Chinese Remainder Theorem). Futhermore, to satisfy real time requirements we can choose small public exponent E, such as 3,17 or $2^{16}+1$, in encryption and verification process. in which case the performance can reach 7.3Mbps.

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

* 본 논문은 2001년 광운대학교 교내 학술 연구 지원과 광운대학교 반도체설계교육센터(IDEC) 의 설계 툴 지원에 의해 연구되었습니다.

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

## Ⅰ. 서 론

공개키 방식 암호 시스템은 단순히 데이터의 암호화에 사용되는 비밀키 암호 시스템과는 달리 전자서명 및 인증 기능까지 갖추고 있는데, 최근 늘어나는 전자상거래에서 인증 수단으로 그 수요가 급증하고 있다. 그 중 대표적인 것이 1978년에 R.L. Rivest, A. Shamir, L. Adleman에 의해서 제안된 RSA 알고리즘[1]이며, 본 논문은 RSA 알고리즘의 기본 연산인 모듈로 멱승연산을 효율적으로 처리하기 위한 하드웨어 구현에 관한 연구이다.

RSA 알고리즘은 큰 정수 계수 $N$(Modulus)의 소인수 분해가 매우 어려움에 그 안전성의 근거를 두고 있는데, 보안성을 높이기 위해 1024비트 이상의 큰 정수(key)를 기반으로 한 모듈로 멱승 연산을 수행해야만 한다. 모듈로 멱승 연산은 내부에 곱셈과 나눗셈이 복합되어 있어 계산 구조가 복잡하고 워드 사이즈가 크기 때문에 소프트웨어로 구현할 경우 상당한 처리시간이 필요하다. 때문에 최근 하드웨어 구현에 관한 다양한 연산 알고리즘들이 연구되어 고속 RSA모듈 구현에 대한 관심과 연구가 활발해지고 있으며, 하드웨어로 구현했을 경우에는 속도뿐만 아니라 키의 안전성 면에서도 소프트웨어보다 월등하다는 장점을 가진다.

본 논문에서는 이러한 속도 향상을 위한 연구로서, 최근 발표된 새로운 멱승 방법인 나눗셈 체인을 이용한 멱승[2]을 효율적으로 처리하기 위한 새로운 하드웨어 구조를 제안하였다.

모듈로 멱승 연산 방법들 중, Binary method[3~5]는 지수 E의 이진 비트열을 스캔해 가면서 제곱과 곱셈을 반복적으로 수행하면 되기 때문에 하드웨어 구현 시 컨트롤을 간단히 처리해 줄 수 있는 장점이 있어 일반적으로 많이 사용한다. 하지만 전체 멱승 시 필요한 곱셈의 횟수는 평균$(3/2)\log_2 E$ 번이며, 하드웨어 구현 시에는 worst-case를 보장하도록 설계되어야 하기 때문에 $2\log_2 E$ 번의 곱셈이 필요하다.[6,7] 또 M-ary Method는 $E$의 비트열을 여러 비트씩 묶어서 한번에 계산하는 방법으로 곱셈의 횟수를 현격히 줄일 수 있다. 하지만 묶여진 비트열의 값이 어떤 값이 될지 알 수 없기 때문에 발생할 수 있는 모든 경우의 값을 저장하고 있어야 한다. 이러한 이유로 많은 하드웨어 리소스 차지하는 단점을 가진다.[4]

본 논문에서 적용한 나눗셈 체인(Division chain)은 지수 $E$를 임의의 제수 $d$와 나머지 $r$의 합으로 표현하는데, $r$은 $d$를 구해 가는 과정에서 얻어질 수 있는 값이 되도록 하여 $r$을 구하는데 필요한 계산량을 줄임으로써 곱셈의 횟수를 줄이는 방법이다. 본 논문에서는 나눗셈 체인을 하드웨어로 처리하기에 적합하도록 변형하였고, 또 이를 위한 하드웨어 구조를 제시하였다.

곱셈기의 전체 구조는 몽고메리 알고리즘을 이용한 시스톨릭 어레이 구조로 설계하였다. 이때 DG (Dependence Graph)를 수직으로 매핑할 경우 나눗셈 체인의 적용으로 인해 처리율 50% 밖에 이룰 수가 없다. 따라서 수평 매핑방법을 적용하여, $k/2+3$ 개의 PE(Processing Element)만으로 $k$ 비트 RSA 연산이 가능하도록 하였다. 이로써 두 개의 곱셈기 모듈이 두 개의 $k$ 비트 프레임을 병렬로 동시에 처리하도록 함으로써 $k+6$ 개의 PE로 100% 처리율을 이룰 수 있도록 하였다. 또 복호화 과정은 수신자가 계수 $N$의 인수인 $p$와 $q$를 알고 있으므로CRT (Chinese Remainder Theorem)의 적용이 가능하다. 본 논문에서는 이의 적용을 위해 두 개의 독립된 곱셈기 모듈이 인터리빙을 이용해 4개의 $k/2$ 비트 암호문을 교대로 처리 할 수 있도록 하였다. 이렇게 함으로써 암호화 시 임의의 $k$ 비트 $E$ 값을 사용하였을 경우보다 4배의 처리속도를 향상시킬 수 있다.

암호화 시에 사용하는 지수 $E$ 값은 공개키이므로 $E=3,17$ 혹은 $2^{16}+1$와 같이 충분히 작은 값을 사용하여도 보안상에는 문제가 되지 않으면서 처리속도를 빠르게 할 수 있다. 또 암호화 과정은 디지털 서명의 검증과정으로 사용되기도 하기 때문에 빠른 처리속도는 필수적이라 할 수 있다. 따라서 디지털 서명은 한번 만들어 두면 반복적으로 사용할 수 있기 때문에 그것의 생성에는 시간이 걸리더라도, 검증과정에서 앞서 언급한 $E$ 값들을 사용하도록 하여 실시간 응답에 대한 요구를 충분히 만족시킬 수 있도록 하였다.

본 논문의 구성은 2장에서 기존의 구현 예들을 살펴보고 그에 따른 장단점에 대해 검토하였으며 3장에서 RSA 알고리즘 및 하드웨어 구현에 따른 성능개선 방법과 본 논문에서 제안하는 새로운 하드웨어 구조 및 컨트롤 방법에 대해 언급하였다. 4장에서 기존의 구현 예들과 비교를 통하여 본 논문에서 제안하는 하드웨어 구조의 성능개선 정도를 나타내었으며, 향후 개선방향과 함께 결론을 맺었다.

## Ⅱ. 구현사례

RSA모듈 구현에 대한 기존의 방법들을 살펴보면, [8]에서는 모듈로 곱셈을 수행할 때 중간 결과 값(partial product)에 대한 모듈로 감소 과정에서, 몽고메리 알고리즘이 LSB값에 따라 계수를 더한 것과는 달리, MSB의 값에 따라 계수의 보수를 더하는 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 놓기 위한 추가의 레지스터가 들어가지만, 후처리(post processing)가 필요하지 않기 때문에 적은 양의 계산 시에는 몽고메리 알고리즘보다 유리할 수 있다.

[9]에서는 몽고메리 알고리즘을 이용하여 모듈로 곱셈을 2차원 평면상에 시스톨릭 어레이(systolic array)로 구현하였다. $k$ 비트의 입력에 대해 처음으로 출력이 나오기 시작할 때까지 $2^k+2$ 의 클럭이 소요되지만, 2차원 평면상에 구현된 것이라 그 크기 때문에 실제로 하드웨어로 구현하기는 힘들다.

[6]에서는 몽고메리 알고리즘을 이용한 멱승 구조를 하나의 FPGA(Field Programmable Gate Array)에 들어갈 수 있도록 PE들의 한 줄만으로 매핑하여 선형 어레이 구조로 구현하였다. 입력 값 $A$, $B$, $N$에 대한 몽고메리 모듈로 곱셈식 $(R_i+a_iB_i+q_iN)/2$의 계산을 매번 반복하기보다는 미리 $B+N$ 값을 계산해 놓고 $a_i$ 와 $q_i$ 값에 따라서 0, $N$, $B$, $B+N$ 값을 멀티플렉서를 이용하여 선택하는 방식을 취함으로써 덧셈기를 하나로 줄였다. 그러나, 이를 멱승에 적용 시에는 매 곱셈이 끝날 때마다 $B+N$을 다시 계산해야 한다. 수직 매핑으로 인한 50%의 처리율을 극복하기 위해 제곱과 곱셈의 인자를 인터리빙하기 때문에 추가의 레지스터와 콘트롤이 복잡해지는 단점도 존재한다. 데이터의 크기 $k$ 비트 입력에 대해서 출력이 나오기까지는 $2(k+2)(k+4)$의 클럭 사이클이 소요된다.

[10]에서는 속도를 높이기 위한 다양한 방법으로 CRT(Chinese Remainder Theorem), Carry Completion Adder, Quotient Pipelining을 복합 사용하여 그들이 새로이 제안한 프로그래머블 능동 메모리(PAM: Programmable Active Memory) 구조에 구현하였다. 그러나, RSA 암호화와 복호화 시 서로 다른 PAM 디자인을 사용해야하고, 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 Ccarry Completion Detection 회로를 사용하였다.

[7]에서는 몽고메리 알고리즘을 적용한 시스톨릭 어레이 구조를 수평으로 매핑하고, 전체 멱승 연산을 위해 Binary Method 를 적용하였다. 암호화 시 $E$는 17 비트로 제한하였으며, 복호화 시 CRT를 적용하였다. 곱셈연산 시 클럭수를 줄이기 위해 하이 래딕스(high-radix)연산 방식을 적용하였는데, 파이프라이닝을 위한 많은 레지스터를 필요로 하기 때문에 상당한 하드웨어 리소스를 필요로 한다. 또 Binary Method의 적용은 전체 멱승연산에 $2\log_2 E$ 번의 곱셈이 필요하다는 단점을 가진다.

본 논문에서는 CRT의 적용과 시스톨릭 어레이 구조, 몽고메리 알고리즘 등 위 논문들의 장점을 최대한 활용하는 한편, 반복적 곱셈의 연속인 멱승 연산에서 곱셈의 횟수를 줄이기기 위해 [2]에서 제안한 나눗셈 체인(Division Chain)을 이용하여 보다 빠른 처리속도를 이룰 수 있도록 한다.

## Ⅲ. 본 론

### 3.1 RSA 알고리즘과 모듈로 연산

RSA 암호 알고리즘은 1024비트 이상의 크기를 갖는 계수 $N$ 과 평문 메시지 $A=A^{DE} \bmod N$ 인 관계를 갖는 두 개의 키 $D$, $E$를 가진다. 1024비트 크기의 계수 $N$은 두 개의 512비트 크기의 소수 $p$, $q$의 곱으로 이루어지며 키 $E$는 $\Phi(N)$ 이 Euler's totient function 즉, $\Phi(N)=(p-1)(q-1)$일 때 $\gcd(E,\Phi(N))=1$ 인 관계를 갖는 $1<E<\Phi(N)$ 의 수 중에서 하나를 임의로 선택한다. 키 $D$는 $ED \bmod \Phi(N)=1$로 계산되고, $E$와 $N$을 공개키로서 공개하며, $D$와 $p$, $q$는 개인키로서 공개되지 않는다. 공개되지 않는 $p$ 와 $q$ 를 계수 $N$ 으로부터 정확히 소인수 분해하기는 1024비트이상의 키 사이즈 때문에 사실상 불가능하며, 이러한 사실이 RSA알고리즘의 안전성을 보장해 준다. 이 키들을 이용하여 식 (1)과

같이 암호화 $C=A^E \bmod N$, 복호화 $A=C^D \bmod N$를 수행하게 된다. 따라서, RSA 암호시스템은 공개키 $E$나 개인키 $D$에 대해 모듈로 멱승을 취함으로써 두 과정이 인자만 바뀌고 동일한 연산으로 이루어진다.

$$C=A^E(\bmod N), \text{where } E=\sum_{i=0}^{s-1} e_i\times 2^i$$
$$A=C^D(\bmod N), \text{where } D=\sum_{i=0}^{s-1} d_i\times 2^i \qquad (1)$$

RSA 연산을 위한 모듈로 멱승 연산의 하드웨어 구현 시 고속 처리를 위해 두 가지 면에서 접근이 가능하다. 첫째는 멱승에 필요한 곱셈의 횟수를 줄이는 방법으로, 하드웨어 구현이 용이하면서 하드웨어 리소스를 적게 차지하는 Binary Method를 많이 사용한다. Binary Method는 지수부 $E$를 $E=(e_{k-1}, e_{k-2}, \cdots, e_0)=\sum_{i=0}^{k-1} e_i 2^i$와 같이 2진 표현에 의해 나타내고 비트열을 왼쪽 또는 오른쪽으로 스캔해 나가는 방법에 따라 LR-method와 RL-method가 있다.[6,10] LR-method의 경우 스캔한 비트가 1이면 현재 출력되는 제곱의 결과 값에 $A$를 곱해주는데, 한번에 한 비트씩 스캔하기 때문에 비트 수 만큼의 제곱과 $h(E)-1$($h(E)$:$E$의 이진 표현에서 1의 개수)만큼의 곱셈이 필요하다. 하지만 여러 비트들을 묶어서 한번에 처리할 경우 추가의 레지스터를 두어 중간에 필요한 값들을 미리 계산해서 저장하면 이 값들을 곱셈연산에 사용 할 수 있기 때문에 제곱의 횟수는 동일하더라도 곱셈의 횟수를 줄일 수 있다. 이러한 방법에는 한번에 $\log_2 m_e$ 비트씩 스캔하는 M-ary Method [3][5]가 대표적이다.

본 논문에서 적용한 나눗셈 체인은 지수 $E$를 임의의 제수(Divisor) $d$로 나누고 이때 발생한 몫에 대해 또 다른 $d$를 적용하여 나눗셈하는 과정을 반복적으로 수행하면 나머지(Remainder) $r$의 집합과 그때 사용된 $d$들의 집합을 얻을 수 있는데 이렇게 얻어진 $d$와 $r$을 이용해 곱셈의 횟수를 줄일 수 있는 방법이다. 그 내용을 간단히 살펴보면 $E$는 임의의 제수 $d$로 나누어 음이 아닌 가장 작은 나머지 $r$을 남긴다고 할 때 $E=dE'+r$의 형태로 쓸 수 있다. 이렇게 표현되는 $A^E$를 계산하기 위해 필요한 최소 곱셈의 횟수를 $F(E)$ 라고 하면 $F(E)\le F(E')+F(d)+F(r)+1$이 된다. 하지만 $d=2^i+1$이고 $r=0$, $2^{i-1}+1$, $2^j$ ($j<i$)인 경우 위 등식이 성립하지 않는 경우가 존재하는데, 위 조건을 만족할 경우 $r$은 $d$를 구해 가는 과정에서 얻어 질 수 있으므로 $F(r)$ 이 제거 될 수 있기 때문이다. 예를 들어 $A^{349}$ 를 $d=17$ 로 나누면 $(A^{17})^{20}A^9$ 가 되고, $A^{17}$을 $A^2\to A^4\to A^8\to A^9\to A^{17}$의 순서로 계산한다면 $A^9$는 따로 계산할 필요가 없다. 때문에 Binary Method를 적용시키면 $F(E)=13$이지만 위와 같이 변형하여 사용할 경우 11번의 곱셈으로 $A^{349}$ 를 계산할 수 있다.

위의 조건을 만족하는 제수 $d$와 나머지 $r$들의 쌍을 선정하고 이를 반복적으로 수행하면 $d_0$, $d_1$, $d_2$, ... $d_{n-1}$, $d_n$의 제수들의 집합을 얻을 수가 있고, 이에 따라 나눗셈에 대한 나머지 $r_0$, $r_1$, $r_2$, ... , $r_{n-1}$, $r_n$ 의 집합을 얻을 수 있다. 따라서, $A$에 대한 $E$ 는

$$A^E=A^{r_0+d_0(r_1+d_1(\cdots+d_{n-1}(r_n+d_n)\cdots))}$$

가 되고, 다시 $(d,r)$의 형태로 표현하면

$$(d_0, r_0)(d_1, r_1)(d_2, r_2)\cdots(d_n, r_n) \qquad (2)$$

와 같이 쓸 수 있다. 식 2와 같이 멱승 연산에 직접적으로 사용될 수 있는 $(d,r)$의 시퀀스(Sequence)를 나눗셈 체인(Division Chain)이라고 하며, $E$ 값으로부터 나눗셈 체인을 만들어 내는 예를 식 3에 보인다.[2]

| | | | |
|---|---|---|---|
| If | $E\equiv 0 \bmod 2$ | then | $d=2$ |
| elseif | $E\equiv 0 \bmod 3$ | then | $d=3$ |
| elseif | $E\equiv 1,2,5,8 \bmod 9$ | then | $d=9$ |
| else | $E\equiv 4,7 \bmod 9$ | then | $d=3$ |

(3)

식 (3)에 임의의 $E=875$를 적용하면, $E=2+9(1+3(0+2(0+2(0+2(0+2(0+2))))))$와 같이 표현되고 또 식 (2)를 이용하면 (9,2)(3,1)(2,0)(2,0)(2,0)(2,0)(2,0) 와 같이 고쳐 쓸 수 있다. 전체 연산은 $(d,r)$이 (9,2)인 왼쪽에서 오른쪽으로 이루어진다

Addition Chain은 지수 $E$를 표현하는 가장 효과적인 방법으로 알려져 있는데, 임의의 시퀀스

$$(a_0, a_1, \ldots, a_n) \text{ where } a_0=1,\ a_0=E \qquad (4)$$

를 가정하고, 임의의 $j$ 에 대해

$$a_i=a_j+a_k \quad (i>0, k<i)$$

가 항상 성립할 때 식 4를 $E$ 에 대한 Addition Chain이라고 한다.

위의 예를 각각 Addition Chain으로 표현하면 (1,2,4,8,9)(1,2,3)(1,2)(1,2)(1,2)(1,2)(1,2)가 되고, 전체 연산은(1,2,4,8,9,11,18,27,54,108,216, 432,864,875)가 된다. 계산이 일어나는 순서는 먼저 (9,2)에서 $A^9$를 구해야 하는데, LR-method를 적용하여 $A^2$->$A^4$->$A^8$->$A^9$의 순서로 계산한다. $A^9$을 구하기 위해 $A^8$ 과 곱하는 $A$는 임의의 레지스터에 저장되어 있어야 하고, 나머지 값인 $A^2$ 는 또 다른 레지스터에 저장한다. 다음으로 (3,1)의 계산은 $B=A^9$ 이 되어 $B^2$->$B^3$의 순서로 계산된다. 이때 나머지 값이 존재하므로 $B$를 이전에 나머지로 저장하고 있던 $A^2$와 곱해 $A^{11}$을 나머지 값으로써 다시 저장한다. $C=A^{27}$이고 $D=A^{54}$, $E=A^{108}$, $F=A^{216}$, $G=A^{432}$, $H=A^{864}$가 되며 최종적으로 나머지 값으로 저장되어 있던 $A^{11}$ 과 $H$를 곱하면 원하는 값을 얻을 수 있다. 즉 Binary Method를 적용하면 $E$= 1101101011이 되어 16번의 곱셈이 필요하지만 나눗셈 체인을 적용할 경우 (9,2)의 계산에 나머지 처리를 포함하여 5번, $A^2\times B$ 계산에 1번, (3,1)에 2번, (2,0)에 각 한번씩 5번의 곱셈이 필요하다. 최종적으로 나머지 값과 $H$를 곱하는데 1번의 곱셈이 필요하므로 전부 14번의 곱셈이 필요하게된다. RSA와 같이 1024비트 이상의 $E$를 사용할 경우 곱셈의 횟수는 현격히 줄어들어 식 (3)과 같이 $d$=2,3,9를 사용하였을 경우 평균 약 $1.4064\log_2 E$ 번의 곱셈으로 멱승 연산이 가능하다.[2]

성능개선을 위한 두 번째 접근방법은 멱승 연산 시 무수히 많이 발생하는 곱셈을 효율적으로 처리하는 것이다. 모듈로 곱셈은 연속된 덧셈 연산으로 수행될 수 있으며 곱셈 연산 시 곱셈을 먼저 한 후 모듈로 연산을 하는 곱셈 후 모듈로 감소(modulo reduction after multiplication)방식과 곱셈 중에 모듈로 연산을 반복하는 곱셈 중 모듈로 감소(modulo reduction during multiplication)방식이 있다. 모듈로 곱셈기의 하드웨어 구현 시 전자의 경우 입력 값의 비트수를 $k$ 비트라 할 때, $k\times k$ 비트의 곱셈기와 $2k$비트의 레지스터 그리고 $2k\times k$ 비트의 나눗셈기 등 리소스를 많이 차지한다. 때문에 주로 후자의 방법을 사용하며 이 방법은 연속적인 덧셈에 의해서 만들어지는 중간 값(Partial Product)을 어떻게 처리하느냐에 따라 두 가지 방법으로 나뉘어 진다. 첫째로 MSB방향으로 커지는 비트를 보고 적당한 계수의 배수를 뺌으로써 새로 늘어난 비트를 제거하는 방법으로 전체적인 비트수를 감소, 유지시키는 MSB우선 방식[8]과 둘째로 LSB부분을 '0'으로 만드는 적당한 계수의 배수를 더한 다음 오른쪽으로 쉬프트하는 방법으로 LSB부분을 제거함으로써 전체적인 비트수를 감소, 유지시키는 LSB우선 방식[15]이 있다. 이 중에 몽고메리 알고리즘은 LSB 우선방식의 대표적인 예로써 계수의 덧셈에 대한 선택이 하위 비트에서 이루어지기 때문에 파이프라인 구현 시 캐리의 지연을 고려치 않아도 되는 장점이 있어 고속 연산을 위해 주로 사용된다. 본 논문에서도 나눗셈 체인을 적용하기 위해 몽고메리 알고리즘을 적용하여 모듈로 곱셈기 를 설계하였다.

### 3.2 CRT(Chinese Remainder Theorem)

전자 서명의 생성과정이기도 한 RSA를 이용한 데이터 복호화 과정은

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

로 표현된다. 이때 키의 생성자인 수신자가 계수 $N$의 인자 $(p,q)$를 알고 있으므로 CRT를 이용하여 보다 빠르게 수행할 수 있는데, 이 방법은 Quisquater와 Couvreur에 의해 제안되었다.[16]

서로 다른 $p_i$에 대해 $i$=1, 2, ..., $k$이고, $p_i$가 각각 서로 소인 관계를 갖는 정수라고 할 때, 즉,

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

일 때 $u_i \in [0, p_i-1]$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 $u$가 존재한다. 또한, 이 유일한 $u$는

$$u = \sum_{i=1}^{k} u_i c_i P_i \pmod P \qquad (5)$$

와 같이 구할 수 있다. 여기서, $P_i = p_1 p_2 \ldots p_{i-1} p_{i} \ldots p_k = P/p_i$ 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 내용을 통해

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

와 같으며 $p$, $q$를 이용해 다음과 같이 나누어 계산

할 수 있다.

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

계수 $N$을 구성하는 $p$, $q$는 각각 $k/2$ 비트이며, 여기에 다시 Fermat's theorem을 적용하여, $D_p = D \pmod{(p-1)}$, $D_q = D \pmod{(q-1)}$ 이라 놓으면

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

가 되어 개인키로 사용되는 $D$는 $p$와 $q$에 대해 각각 $k/2$ 비트로 만들 수 있다. 또한 $C_p = C \pmod{p}$, $C_q = C \pmod{q}$로 치환하면 모든 인자를 $k/2$ 비트로 줄일 수 있다.

$$M_p = C_p^{D_p} \pmod{p}$$
$$M_q = C_q^{D_q} \pmod{q} \qquad (6)$$

식 (5)에 $k/2$ 비트 $A_p$와 $A_q$를 대입하여

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

와 같이 $k$ 비트 $A$을 구할 수 있다. 여기서, $C^*_p = q^{-1} \pmod{p}$, $C^*_q = p^{-1} \pmod{q}$ 이며

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

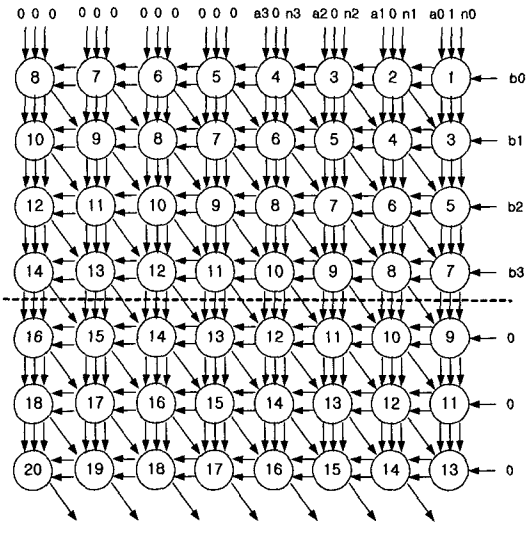
와 같이 간단히 증명된다.

본 논문에서는 $k$ 비트 암호문 $C_1$ 과 $C_2$에 CRT를 적용하여 $k/2$ 비트 $C_{p1}$, $C_{p2}$, $C_{q1}$, $C_{q2}$으로 나누고, 두 개의 $k/2$ 비트 곱셈기 모듈을 통해 동시에 처리되도록 하기 위해 지수부로 사용되는 $p$에 대한 $C_{p1}$, $C_{p2}$ 을 먼저 처리하고 $q$에 대한 $C_{q1}$, $C_{q2}$를 인터리빙하는 방법을 이용하였다. 이로써 곱셈 시 클럭수를 절반으로 줄이고, 멱승 시 클럭수를 절반으로 줄일 수 있으며, 전체 처리율은 100%가 되고 클럭수가 1/4로 줄어들어 4배 향상된 속도를 얻을 수 있도록 하였다.

### 3.3 파이프라인드(Pipelined) 몽고메리 곱셈기

나눗셈 체인을 이용한 멱승 연산을 위해서는 식 (2)에 나타나는 각각의 $(d,r)$을 위한 곱셈기를 우선적으로 설계하여야 한다. [그림 1]은 몽고메리 알고리즘을 이용한 모듈로 멱승 처리기의 내부 계산 구조를 보이기 위해 그림 2의 PE를 이용하여 4비트의 키 사이즈에 대한 데이터의 종속 그래프(DG : Dependence Graph)를 나타낸 것이다. 이를 어떤 방향으로 매핑하고, 어떻게 스케줄링 하느냐에 따라 다양한 곱셈기 모듈이 구현 될 수 있는데, 본 논문에서는 데이터 이동방향을 고려하여 처리율 100%의 파이프라인 동작이 가능하도록 하기 위해 DG를 수평으로 매핑하는 방법을 택하였다.

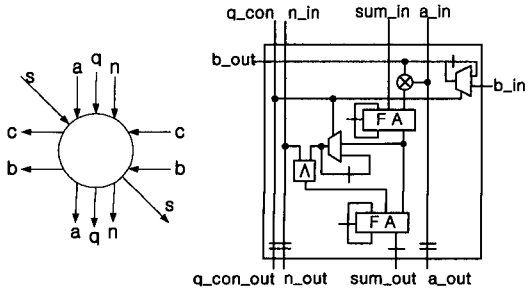
식 (2)의 $(d_i,r_i)$는 LR-method를 이용하기 때문에 이전 결과 값을 받아서 계산해야 되는데 [그림 2]를 수평매핑 한다고 할 때 한번의 곱셈연산이 끝나고 14 클럭 후 최초의 출력이 나오기까지 다음 입력을 위해 기다리게 되면 50%의 처리율 밖에 이룰 수 없다. 그 이유는 첫 번째 PE는 8클럭 후에 입력이 없으면 아무런 연산을 수행하지 않기 때문이다. 이는 8클럭 후에 sum_out 값을 받아서 연산을 시작해야 하는 5번째 PE를 첫 번째 PE가 대신할 수 있도록 하면 100%처리율을 이룰 수 있다. [그림 3]에 그 신호의 흐름을 나타내었다. 이로써 4개의 PE로 전체 연산이 가능하기 때문에 필요한 PE의 개수를 반으로 줄일 수 있으며 $k$ 비트 연산에 대해 절반의 하드웨어 리소스로 연산이 가능하게된다. 요구되



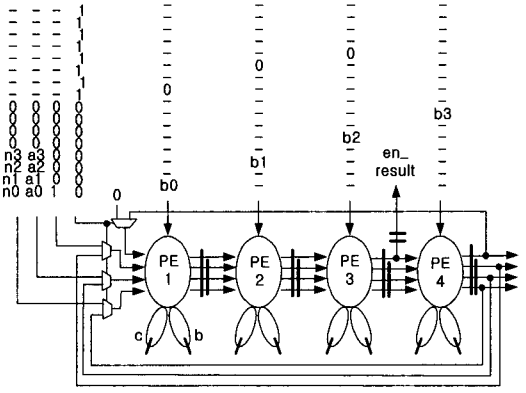

[그림 1] 몽고메리 곱셈기의 DG(k=4)

는 clock cycle은 최초의 출력 값이 나올 때까지 기다렸다가 다음 출력으로 사용해야 되기 때문에 한 번의 곱셈에 $2(k+4)$이 필요하고 전체 멱승에는 평균 $1.4064(2k(k+4))$클럭이 필요하다.[2] 이는 $k/2$ 개의 PE를 사용하였을 경우이며 $k$ 비트 키 사이즈에 대해 두 개의 연속하는 $2k$ 비트의 평문을 입력으로 받아 두 개의 서로 다른 $k$비트 평문이 병렬로 동시에 처리되도록 하면 처리율 100%를 이룰 수 있을 뿐 아니라 성능은 두 배가된다.

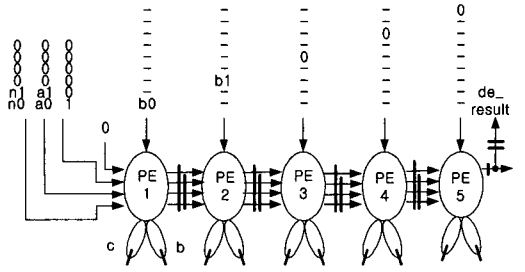
〔그림 2〕의 (a)의 그래프를 선형 어레이 곱셈기로 구현하기 위해 수평으로 매핑한 후 각 PE 간의 신호의 흐름을 나타내는 SFG(Signal Flow Chart)를 〔그림 3〕의 (a)에 나타내었으며, 〔그림 3〕의 (b)는 복호화 시의 SFG를 나타낸 것이다. 〔그림 3〕의 (a)에서, 제일 상단에 위치한 멀티플렉서는 PE를 재사용하기 위해 추가한 것으로, 8 클럭까지 "$a_0a_1a_2a_3$0000", "$n_0n_1n_2n_3$0000"와 PE 내부 멀티플렉서를 컨트롤하기 위한 "10000000"의 q_con이 입력되고, 9번째 클럭에서 피드백 되는 값들을 받기 위해 추가한 멀티플렉서인 get_feed를 통해 PE4로부터 출력되는 sum 과 $n$, $a$, q_con을 입력으로 받을 수 있다. 복호화 과정에서 CRT의 적용 시 암호문 $C_p$는 식 (6)에 의해 각 $k/2$ 비트인 $C_{p1}$, $C_{p2}$가 되고 〔그림 3〕의 (b)와 같이 $k+3$개의 PE를 가지는 동일한 두 개의 곱셈기 모듈의 처음 입력으로 동시에 들어가고, 6 클럭 후에 $C_{q1}$과 $C_{q2}$에 해당되는 암호문이 입력으로 들어가 $p$에 대한 암호문과는 독립적으로 처리되도록 한다. 즉, 첫 번째 PE는 6 클럭 동안 $C_p$의 암호문이 입력되고, 7번째 클럭 부터 12번째 클럭 까지 $C_q$의 암호문이 입력된다. $C_p$에 대한 결과 값이 10 클럭 후에 출력되지만 첫 번째 PE가 $C_q$를 입력으로 받고 있는 상태이므로, 다음 곱셈을 위한 입력으로 사용하기 위해서는 두 클럭을 쉬어야 한다. 12 클럭 후의 최초 출력 값은 다음 곱셈을 위해 사용한다든지




〔그림 2〕 몽고메리 곱셈기의 노드함수와 PE




(a) 암호화(k=4)



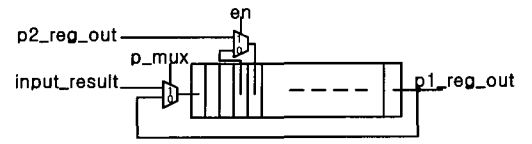

(b) 복호화(k/2=2)

〔그림 3〕 몽고메리 곱셈기의 SFG

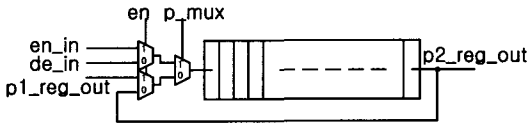
레지스터에 저장될 입력 값으로 사용 할 수 있다.

전체 연산에 필요한 레지스터는 각 키 값들을 저장하기 위한 레지스터와 중간결과 값들을 저장하기 위한 레지스터가 필요하다. $A^m$을 저장하기 위한 레지스터를 M 레지스터(m_reg)라 하고, $A^r$ 즉, 나머지 값을 저장하기 위한 레지스터를 P 레지스터(p_reg)라 한다. 몽고메리 알고리즘을 모듈로 멱승에 적용하기 위해서는 각 곱셈의 정확한 값을 구하는데 필요한 후처리 과정을 없애기 위해 전처리 과정에서 $A$에 mont_val=$2^{2(k+3)}$mod $N$ 값을 곱해 주어야 하는데,[7] 최초 계산이 일어나기 전 M 레지스터에는 $A$를 저장하고 P 레지스터에는 mont_val을 저장한다.

두 개의 레지스터 P와 M은 복호화 시 CRT를 적용하기 위해 각각 두 부분으로 나뉘어 진다. p_reg1, p_reg2는 복호화 과정에서 멱승의 유효한 입력 값의 크기를 맞추기 위해 추가한 4비트와 함께 각각 독립적인 $k+4$비트 쉬프트 레지스터를 구성한다. en 이 0인 상태에서 복호화가 이루어지고, 1이면 암호화가 이루어지는데, 암호화 시 p_reg2의 결과 값이 p_reg1의 입력으로 사용되어 〔그림 4〕의 p_reg1와 쌍을 이루어 하나의 $k+4$비트 쉬프트 레지스터를 구성한다.




(a) P1 레지스터




(b) P2 레지스터

〔그림 4〕 P 레지스터

m_reg는 p_reg와 동일한 구조를 가지며, n_reg는 n_reg2에서 곱셈기 블록으로부터의 입력이 없다는 것을 제외하면 p_reg와 동일한 구조를 가진다.

### 3.4 나눗셈 체인(Division Chain)을 이용한 모듈로 멱승

컨트롤 방법이 간단한 Binary Method의 경우 지수 $E$는 1 또는 0으로 표현되기 때문에 한 비트씩 이동하면서 항상 제곱을 수행하고, 해당되는 비트가 1이면 곱셈을 수행한다.

여기서 $(d_0,r_0)(d_1,r_1)(d_2,r_2)\ldots(d_n,r_n)$으로 표현되는 나눗셈 체인을 직접적으로 멱승연산에 이용할 경우 각각의 $(d,r)$로부터 위와 같은 규칙성을 찾기가 힘들다. 또 $d$=2,3,9의 경우보다 더 많은 $d$를 사용하는 경우[2] 어떤 $d$들을 사용하느냐에 따라 컨트롤 또한 틀려지기 때문에 하드웨어 구현에 많은 어려움이 따른다. 예를 들어 식 (3)에서 생성된 나눗셈 체인에서 일어날 수 있는 모든 경우에 대해 디코딩해서 컨트롤 시그널을 생성한다고 가정할 때 경우의 수는 7이 된다. 각각의 경우를 할당하기 위해 3비트면 충분하므로 비교적 간단한 하드웨어 구조를 가진다. 하지만 $d$=2,3,5,9,17,33을 이용한다고 할 때 생성될 수 있는 $(d,r)$의 경우의 수는 24가 된다.[2] 각각을 할당하기 위해 5비트가 필요하고 이 5비트로부터 컨트롤 시그널 생성을 위해 디코딩한다고 할 때 $d$=2,3,9의 경우와 다른 하드웨어를 다시 만들어야 하는 문제점이 발생한다.

또 성능향상을 위해 사용되는 $d$를 더 많이 사용한다고 할 때 항상 하드웨어를 바꾸어주어야 할뿐만 아니라 서로 다른 경우를 정의하기 위해 필요한 비트 수는 계속 늘어난다.

| (d,r) | div_code |
|---|---|
| (2,0) | 11 |
| (2,1) | 10 11 |
| (3,0) | 01 11 |
| (3,1) | 10 00 11 |
| (3,2) | 01 10 11 |
| (9,0) | 01 00 00 11 |
| (9,1) | 10 00 00 00 11 |
| (9,2) | 01 10 00 00 11 |
| (9,5) | 01 00 01 10 11 |
| (9,8) | 01 00 00 10 11 |

(a) encoding table

| | | | |
|---|---|---|---|
| (2,0) | A | | |
| | A | | |
| | 11 | | |
| (2,1) | P | A | |
| | A | A | |
| | 10 | 11 | |
| (3,0) | A | $A^2$ | |
| | A | M | |
| | 01 | 11 | |
| (3,1) | P | A | $A^2$ |
| | A | A | M |
| | 10 | 00 | 11 |
| (3,2) | A | P | $A^2$ |
| | A | $A^2$ | M |
| | 01 | 10 | 11 |

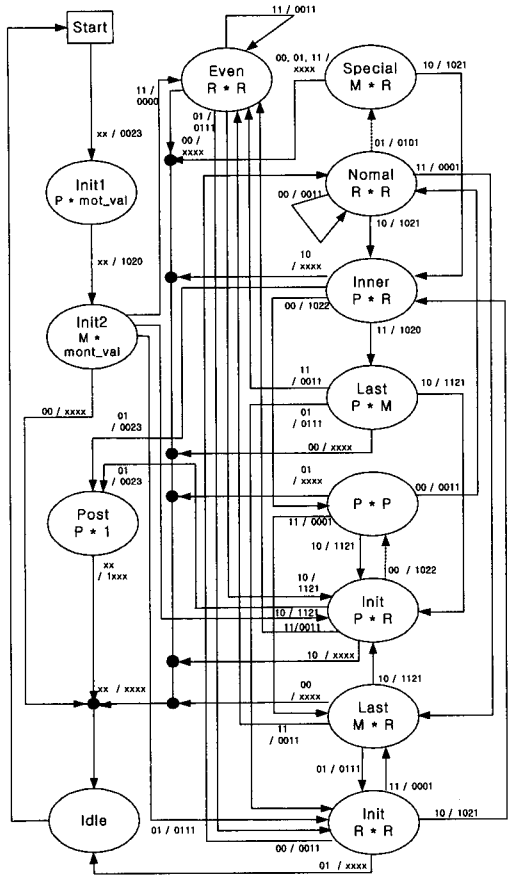
| | | | | | |
|---|---|---|---|---|---|
| (9,0) | A | $A^2$ | $A^4$ | $A^8$ | |
| | A | $A^2$ | $A^4$ | M | |
| | 01 | 00 | 00 | 11 | |
| (9,1) | P | A | $A^2$ | $A^4$ | $A^8$ |
| | A | A | $A^2$ | $A^4$ | M |
| | 10 | 00 | 00 | 00 | 11 |
| (9,2) | A | P | $A^2$ | $A^4$ | $A^8$ |
| | A | $A^2$ | $A^2$ | $A^4$ | M |
| | 01 | 10 | 00 | 00 | 11 |
| (9,5) | A | $A^2$ | $A^4$ | P | $A^4$ |
| | A | $A^2$ | M | $A^5$ | $A^5$ |
| | 01 | 00 | 01 | 10 | 11 |
| (9,8) | A | $A^2$ | $A^4$ | P | $A^8$ |
| | A | $A^2$ | $A^4$ | $A^8$ | M |
| | 01 | 00 | 00 | 10 | 11 |

(b) operand 와 division code

〔그림 5〕 나눗셈 체인의 encoding table

이러한 문제점을 해결하기 위해 본 논문에서는 $d=2^i+1$인 모든 경우에 적용가능 하도록 〔그림 5〕와 같이 소프트웨어를 이용해 나눗셈 체인을 한번의 곱셈에 두 비트만을 할당하는 새로운 코딩방법을 제안한다. 예를 들어 $E$=875일 때 $d$=2,3,9를 적용하면 (9,2)(3,1)(2,0)(2,0)(2,0)(2,0)(2,0)과 같은 나눗셈 체인을 가진다는 것을 앞서 언급하였는데, 이 경우 새롭게 코딩된 비트열은 〔그림 5〕의 (a)에 따라 01100000111000111111111111과 같은 비트열을 가지게 된다. 새롭게 변형된 나눗셈 체인의 비트 열을 Div_code라 부르기로 한다. Div_code는 곱셈이 한번 일어날 때마다 좌측으로 두 비트씩 스캔하고 이 두 비트가 FSM(Finite State Machine)의 상태를 변화시키는 입력으로 사용되도록 하였다. 이에 따른 상태천이에 관한 내용은 〔그림 6〕의 FSM과 함께 언급하였다.

변형된 비트열의 저장은 $k$ 비트 레지스터(encoded_dc_reg)에 최초로 키 값들의 저장과 함께 $k$ 비트를 저장하고 $k/2$번의 곱셈이 끝난 후 새롭게 코딩된 비트열의 연속하는 $k$ 비트를 encoded_dc_reg 레지스터에 다시 저장하는 방법을 이용한다. 곱셈이 최대 $2\log_2 E$ 번 필요하다고 할 때 필요한 레지스터는 $4k$ 비트이고 이를 전부 저장하는 것은 하드웨어 리




〔그림 6〕 컨트롤 블록의 FSM

소스를 효율적으로 이용하지 못하는 결과를 초래하기 때문에 추가적으로 3번만 데이터 교환이 이루어지도록 구현하여 하드웨어리소스를 낭비하지 않도록 하였다. 추가적인 데이터 교환은 1024비트 연산을 수행하는데 수 ms가 소요됨을 감안하면 전체적인 성능에 큰 영향을 미치지 않는다.

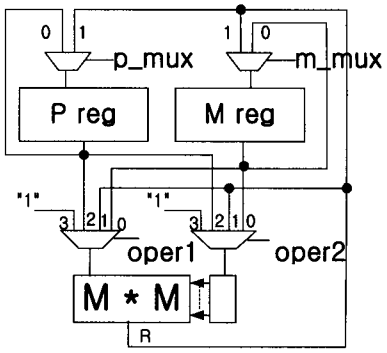
〔그림 5〕의 (a)는 divisor 2, 3, 9를 사용하였을 때 발생할 수 있는 모든 경우를 나타낸 것이며 각각의 div_code 생성은 다음과 같다. $(d,r)=(2,0)$ 즉, 2로 나누어 0을 남기는 경우는 현재 출력되는 이전의 결과 값으로 제곱하는 경우이며 이때의 결과 값은 레지스터에 저장하지 않아도 되고 div_code=11이 된다. 새로운 $(d,r)$이 시작될 때 이전의 결과값을 받아 제곱이 실행될 경우 즉 $(d,r)\in\{(3,0),(3,2),(9,0),(9,2),(9,5),(9,8)\}$인 경우 최초의 div_code는 01이 된다. 이때 제곱에 사용된 $A$ 는 $A^d$을 구하는데 사용되므로 M 레지스터에 저장 되어야한다. 새로운 $(d,r)$이 시작되는 부분이지만 1을 나머지로 남기는 경우 즉 $(d,r)\in\{(2,1),(3,1),(9,1)\}$의 경우 이전곱셈의 결과 값과 P 레지스터의 값을 곱해서 다시 P 레지스터로 저장해 주어야 한다. 이때 div_code=01과 마찬가지로 현재 출력되는 결과 값은 M 레지스터에 저장되어야하며 이때는 div_code=10이 되도록 한다. $A^d$ 을 구해 가는 과정에서 단순히 이전의 결과 값을 이용하여 제곱이 일어나는 경우는 div_code=00이 되도록 한다. $A^9$ 또는 $A^3$은 현재 곱셈의 결과로 얻어진 값인 $A^8$ 또는 $A^2$과 M 레지스터에 저장되어 있는 의 곱으로 얻어지는데 이런 경우div_code=11이 된다. (2,0)의 경우 div_code=11이기 때문에 이 경우와 div_code는 동일하지만 상태천이의 시점이 틀리기 때문에 충돌은 일어나지 않는다. 특히 div_code=10은 나머지를 위한 계산이 이루어지는 과정이므로 div_code=10 다음의 곱셈을 수행할 때 P레지스터에는div_code=10일 때의 결과 값을 저장할 수 있도록 해야 한다. 이상과 같은 방법을 (9,5)의 경우에 적용시킬 경우, $A^4$ 과 M 레지스터의 값인 $A$와 곱셈이 일어날 때 위 설명대로 코딩한다면 div_code=01 00 11 -- -- 이 되기 때문에 {(3,1), (9,1), (9,2)}와 같이 마지막 네 비트가 00 11 인 경우와 구별이 되지 않는다. 따라서 div_code=00 다음에는 div_code=01의 계산이 필요한 경우가 존재하지 않으므로 div_code=01로 만들고 이를 (9,5) 즉 $r=2^{i1}+1$ 경우에서 사용한다. 이때 $A^5$를 계산한 후 더 이상 $A$를 저장하고 있어야 할 필요가 없기 때문에 $A^5=A^4\times A$의 계산에 사용한 $A^4$를 M 레지스터에 저장하고 $A^9$의 계산에 사용한다. 나머지 계산을 위해 $A^5$의 결과를 P 레지스터의 값과 곱함과 동시에 P 레지스터에 저장해 두었다가, M 레지스터에 저장되어 있는 $A^4$와 곱하여 $A^9$을 계산하는데 사용 할 수 있도록 하며div_code=11이 된다.

〔그림 5〕의 (b)는 각각의 곱셈에 따른 두 개의 대상(operand)과 그때 사용되는 상태천이를 위한 입력 값을 함께 나타낸 것인데, 예를 들어 $(d_n,r_n)=(9,2)$의 경우 $A^2\rightarrow P\times A^2\rightarrow A^4\rightarrow A^8\rightarrow A^9$의 순서로 곱셈을 한다. 〔그림 5〕의 (a)에 따라 최초의 div_code=01이며, $n=0$ 라면 M 레지스터에 저장되어 있는 값을 제곱하고 $n\neq 0$이면 현재 출력되는 값을 제곱한다. 다음으로 div_code=10 이며 $A^2$은 나머지 값으로 다시 사용되어야 하기 때문에 나머지 값을 저장하고 있는 P 레지스터의 값과 곱해주어야 한다. 이전에 저장되어 있던 나머지 값이 존재하지 않는다

면 P 레지스터는 $A^2$이 되고, 이전에 값을 가지고 있다면 그 값과 $A^2$의 값을 곱한 결과를 P 레지스터에 저장한다. $A^4$는 $A^2$ 의 제곱으로 얻어지는데 $A^2$은 div_code=01의 계산이 끝나고 출력되는 값이며 div_code=10으로 천이 하면서 P 레지스터에 일시적으로 저장한다. 이는 div_code=10에서 P 레지스터의 값은 현재 연산을 위해 곱셈기로 입력되고 있는 상태이므로 일시적으로 비어 있는 상태이므로 가능하다. div_code=00에서 현재 출력되는 결과 값에 대한 제곱이 두 번 일어나고 div_code=11에서 $A^4 \times A^4$ 결과 값으로 출력되는 $A^8$과 M 레지스터에 저장되어 있는 $A$와 곱하는데, 그 결과 $A^9$ 이 출력되고 다음 $(d,r)$을 위해 $A^9$은 M 레지스터에 저장된다.

이상과 같은 코딩방법을 적용하면 $d=2^i+1$인 모든 경우에 적용가능하며, 앞서 언급한 것처럼 서로 다른 $(d,r)$을 디코딩 해서 컨트롤 시그널을 생성한다고 할 때 하드웨어를 다시 구성해야 하는 문제점을 해결할 수 있다.

전체 연산에서 두 비트 div_code가 입력됨에 따라 한번의 곱셈이 일어나게 되는데, 그 때 곱셈의 대상이 달라지고 레지스터의 입력 값이 달라진다. 곱셈이 일어나는 경우에 대해 하나의 상태로 정의하고, 상태천이에 따른 state diagram을 [그림 6]에 나타내었다. 상태는 13개가 필요하고 각각의 상태에 대해 간단히 언급하면, 각각의 상태에 R*R이 붙는 경우 즉 Even R*R, Normal R*R, Init R*R은 이전 상태의 결과 값을 제곱하는 경우이다. Special M*R, Last M*R은 이전 상태의 결과 값과 M 레지스터에 저장되어 있는 값을, 그리고 Inner P*R은 이전상태의 결과 값에 P 레지스터의 값을 곱하는 경우이다. Last P*M은 P 레지스터의 값과 M 레지스터의 값을 곱하며 P*P는 P 레지스터의 값을 제곱하는 경우이다. Init1과 Init2는 각각 1과 평문 M 을 mont_val 와 곱한다. [그림 6]에서 Start 시그널이 입력되면 Init1과 Init2를 통해 몽고메리 모듈러 곱셈을 위한 전처리 과정을 수행하고, [그림 6]과 함께 제시한 예에서 addition chain이 (1,2,4,8,9)인 $(d,r)=(9,2)$에서 div_code=01가 된다. 때문에 Init2에서 Init R*R 상태로 천이 하게 된다. 이때는 Init2에서 곱셈기의 출력 값인 $A=M\times 2^{2(k+3)} \bmod N$을 이용하여 제곱을 수행한다. 마지막의 1001은 현재 출력되는 최종 값과 P 레지스터의 값을 곱하기 위해 Init P*R 또는 Inner P*R로 천이하고, 최종적으로 Post 상태로 천이 하여 1을 곱하여 원하는 결과 값을 얻을 수 있도록 하기 위해 추가한 것이다.




0110000011 100011 11 11 11 11 11 (1001)
(9,2) (3,1) (2,0)(2,0)(2,0)(2,0)(2,0)

Start -> Init1 -> Init2 -> Init R*R -> Inner P*R ->
( P:=K×1) ( M:= K×M ) (P:=A×A) ( P:=P×$A^2$)
01 10

P*P -> Nomal R*R -> Last M*R -> Init P*R -> P*P
($A^2$*$A^2$) ($A^4$*$A^4$) (P,M:=$A^8$*A) (P:=$A^9$*P) ($A^9$*$A^9$)
00 00 11 10 00

-> Last M*R -> Even R*R -> Even R*R -> Even R*R
($A^{18}$*$A^9$) ($A^{27}$*$A^{27}$) ($A^{54}$*$A^{54}$) ($A^{108}$*$A^{108}$)
11 11 11 11

-> Even R*R -> Even R*R -> Init P*R -> Post P*1
($A^{216}$*$A^{216}$) (P,M :=$A^{432}$*$A^{432}$) (P*$A^{864}$) (P:=$A^{875}$*1)
11 11 10 01

-> End -> Idle

( Where K = mont_val = $2^{2(k+3)} \bmod N$ , A = M×K )

[그림 7] div code를 이용한 전체 멱승

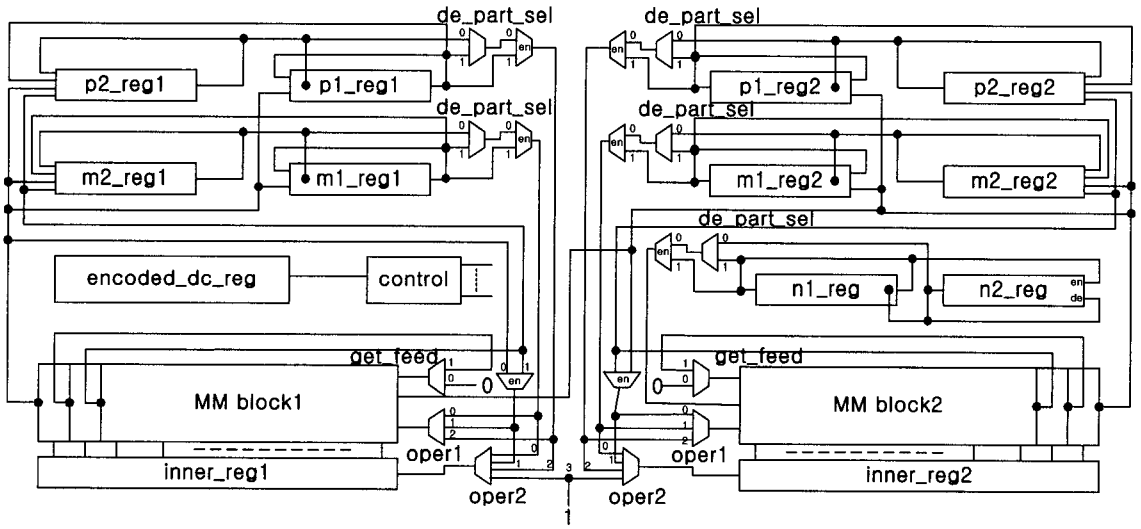
[그림 7]은 [그림 6]에서 출력되는 컨트롤 시그널이 곱셈기와 P 레지스터, 그리고 M 레지스터를 어떻게 컨트롤하는지를 나타내기 위해 $E=875$일 때의 예와 함께 나타내었으며, [표 1]에 상태 테이블을 나타내었다.

앞에서 설명한 멱승기의 하드웨어 구조를 [그림 8]에 나타내었다. 모듈러 곱셈기 블록인 MM 1 블럭과 MM 2 블록은 동일한 구조를 가지며, $k$ 비트의 모듈로 멱승기를 구현할 때 각각 $k/2+3$개의 PE로 구성된다. 멀티플렉서 콘트롤 시그널 de_part_sel은 복호화 시에 $C_p$와 $C_q$에 대한 연산이 교대로 이루어지므로 각각의 레지스터 쉬프트 이네이블 시그널과 동일한 시그널을 이용하여 현재 쉬프트 하고 있는 레지스터 값이 출력이 되도록 한다.

〔표 1〕 상태 테이블(State Table)

| | next state | | | | output | | | |
|---|---|---|---|---|---|---|---|---|
| present state | input = 00 | input = 01 | input = 10 | input = 11 | input = 00 | input = 01 | input = 10 | input = 11 |
| Init1<br>1*mont_val | Init2<br>P*mont_val | Init2<br>P*mont_val | Init2<br>P*mont_val | Init2<br>P*mont_val | 1023 | 0023 | 0023 | 0023 |
| Init2<br>M*mont_val | Idle | Init R*R | Init P*R | Even R*R | xxxx | 0111 | 1121 | 0000 |
| Even R*R | Idle | Init R*R | Init P*R | Even R*R | xxxx | 0111 | 1121 | 0011 |
| Special M*R | Idle | Idle | Last M*R | Idle | xxxx | xxxx | 1021 | xxxx |
| Nomal R*R | Nomal R*R | Special M*R | Inner P*R | last M*R | 0011 | 0101 | 1021 | 0001 |
| Inner P*R | P*P | Post P*1 | Idle | Last P*M | 1022 | 0023 | xxxx | 1020 |
| Last P*M | Idle | Init R*R | Init P*R | Even R*R | xxxx | 0111 | 1121 | 0011 |
| P*P | Nomal R*R | Idle | Init P*R | Last M*R | 0011 | xxxx | 1121 | 0001 |
| Init P*R | P*P | Post P*1 | Idle | Even R*R | 1022 | 0023 | xxxx | 0011 |
| Last M*R | Idle | init R*R | init P*R | Even R*R | xxxx | 0111 | 1121 | 0011 |
| Init R*R | Nomal R*R | Idle | Inner P*R | Last M*R | 0011 | xxxx | 1021 | 0001 |
| Post P*1 | Idle | Idle | Idle | Idle | 1xxx | 1xxx | 1xxx | 1xxx |
| Idle | Idle | Idle | Idle | Idle | xxxx | xxxx | xxxx | xxxx |

input = (div_code1,div_code2)  output = (p_mux, m_mux, oper1, oper2)




〔그림 8〕 RSA 멱승 처리기의 전체 구조

## Ⅳ. 결 론

본 논문에서는 RSA알고리즘의 하드웨어 구현에 있어서 보다 빠른 성능을 이루기 위해 나눗셈 체인 룰고메리 알고리즘, 복호화 시에 CRT방법을 적용하여 구현하였다. 나눗셈 체인을 이용한 멱승방법은 알고리즘 자체로는 상당한 성능개선을 보이지만 하드웨어로 구현할 때 데이터 패스(data path)의 규칙성을 찾기가 힘들어 컨트롤이 복잡해질 뿐만 아니라 더 많은 경우의 $d$를 사용할 경우 각각에 대해

하드웨어를 다시 구성해야 하는 단점을 가진다. 따라서 본 논문에서는 나눗셈 체인을 그대로 사용하지 않고 새롭게 코딩하는 방법을 제안하여 위의 문제점을 해결하고자 하였다. 또 몽고메리 알고리즘의 DG를 수평으로 매핑하여 PE를 재사용할 수 있도록 하였기 때문에 $k$+6개의 PE로 100%의 처리율을 가질 수 있도록 하였다.

복호화의 경우에는 사용자가 합성수 $N$의 인수를 알고 있기 때문에 CRT의 적용이 가능하다. 따라서 두 개의 독립된 곱셈기 모듈이 인터리빙을 이용해 4개의 $k/2$ 비트 암호문을 교대로 처리 할 수 있도록 하였다. 이렇게 함으로써 CRT를 사용하지 않을 경우 보다 4배 정도의 속도 향상을 이루었다. CRT의 사용이 불가능해 복호화에 비해 상대적으로 속도가 느린 암호화과정에 대해서는 공개키 $E$ 값을 17비트 이하로 제한하여 사용함으로써 전체적으로 빠른 속도를 유지하였다.

그 결과를 다른 RSA의 하드웨어 구현 논문들 중 성능이 뛰어나다고 판단되는 논문[6,10,7]들과 비교하여 〔표 2〕와 〔표 3〕에 나타내었다. 〔7〕에서는 곱셈연산에 하이래딕스 연산방식을 적용하여 래딕스의 크기에 따라 하나의 PE내에 $\log_2 m_p \times \log_2 m_p$ 비트의 곱셈기와 파이프라이닝을 위한 많은 레지스터를 필요로 하기 때문에 상당한 하드웨어 리소스를 요구하게된다. 반면 본 논문에서 제안하는 하드웨어의 구조는 $m_p$=2를 사용하였기 때문에 상대적으로 적은 하드웨어 리소스로 구현이 가능하다. 〔6〕에서는 FPGA에 구현하였는데 사용되는 CLB(Configurable Logic Block)의 숫자를 줄이기 위해 $u$개의 PE를 하나의 CLB에 구현하는 방법을 사용하였다. 전체구조는 모듈로 곱셈기의 시스톨릭 어레이를 수직으로 매핑하였고 직렬로 입력되는 한 비트에 대해 $u$개의 비트가 처리되도록 하였기 때문에 하드웨어 리소스는 $m_p$=2와 비슷하다. 〔표 2〕를 통해 삼성 0.5$um$ CMOS 스탠다드 셀 라이브러리[17]를 근거로 이를 예측하고 비교하여 나타내었다.

본 논문의 설계 방법론에 따라 곱셈연산에 $m_p$=16의 하이래딕스 연산 수행을 가정하면 임의의 $E$를 사용한 암호화의 경우 445Kbps의 속도로 처리가 가능하며, CRT를 적용한 복호화는 1.8Mbps의 속도로 처리할 수 있다. 따라서, 본 논문이 하드웨어 구현에 따른 알고리즘의 효율적인 적용에 관심의 초점을 두고 이진 연산방식을 적용한 것을 고려하면 본 논문의 성능개선은 다른 방법들보다 방법론적인 측면에서 상당히 우수하다고 할 수 있다. 이를 성능면에서 비교하여 〔표 3〕에 나타내었다.

〔표 2〕 게이트 카운트(gate count) 비교

| | [6] | [7] | Here |
|---|---|---|---|
| PE complexity | 2 to 1 mux 2개<br>3 to 1 mux 1개<br>4 to 1 mux 1개<br>Full adder 1개<br>3x8 Decoder 1개<br>Flip Flop 8개 | 2 to 1 mux 12개<br>AND 32개<br>Full adder 32개<br>Half adder 12개<br>Flip Flop 50 개 | 2 to 1 mux 2개<br>AND 2개<br>Full adder 2개<br>Flip Flop 11개 |
| External reg & Control | (k+3)bit Register 12<br>State Machine * 1 | (k+3)bit Register * 6<br>State Machine * 1 | (k+3)bit Register * 7<br>State Machine * 1 |
| Gate Count ( 0.5 um ) | 180,000 | 200,000 | 140,000 |

〔표 3〕 성능 비교

| | | [6] | | [10] | | [7] | | Here | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ms | Mbps | ms | Mbps | ms | Mbps | ms | Mbps | ms | Mbps |
| k = 1024 | (random E) | 40.50 | 0.025 | 24.80 | 0.04 | 3.30 | 0.31 | 7.40 | 0.14 | 2.30 | 0.445 |
| | ($E = 2^{16}+1$) | 0.75 | 1.37 | | | 0.07 | 15.5 | 0.14 | 7.3 | 0.04 | 23.27 |
| | CRT | 10.18 | 0.1 | 6.20 | 0.165 | 0.83 | 1.2 | 1.66 | 0.56 | 0.58 | 1.8 |
| Frequency | | 52Mhz | | 40Mhz | | 160Mhz | | 200Mhz | | 160Mhz | |
| Radix (mp) | | ( u = 4 ) | | 4 | | 16 | | 2 | | 16 | |
| | | always | | always | | always | | average | | | |

ASIC으로 구현을 위해 삼성 0.5um CMOS 스탠다드 셀 라이브러리[17]을 이용하여 합성한 결과 최장 지연패스는 덧셈기(full adder) 2개의 덧셈결과값이 나오기까지의 지연 시간과 두 개의 논리합, 그리고 두 개의 2:1 멀티플렉서를 거치는 지연 시간을 더한 값이 되어 4.24ns를 얻을 수 있다. 전체 게이트 카운트는 2입력 NAND 게이트를 기본단위로 하여 약 14만 게이트를 차지한다. 〔표 3〕에 보이듯이 클럭 주파수 200MHz에서 암호화 시 140Kbps, 복호화 시 CRT의 사용으로 560Kbps의 성능을 갖게 된다.

## Reference


〔1〕 R.L. Rivest, A. Shamir and L. Adlemn, "A method for obtaining digital signatures and public-key cryptosystems", Communications of the Association for Computing Machinery, 21(2) pp. 120~126, February 1978.

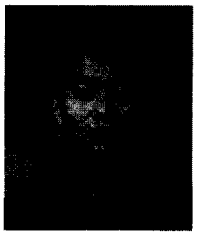
〔2〕 Colin Walter, "Exponentiation Using Division Chains", IEEE Trans On Computers, Vol 47, No. 7, JULY 1998.

〔3〕 Cetin Kaya Koc. "High-Speed RSA Implementation", RSA Laboratories Technical Report, November 1994.

〔4〕 Cetin Kaya Koc, "RSA Hardware Imple-

mentation", RSA Laboratories Technical Report, August 1995.
[5] D.E Knuth, The Art of Computer Programming, Vol. 2, Seminumerical Algorithms. Second ed, Addison-Wesley, 1981.
[6] Thomas Blum, Christof Paar, "Montgomery Modular Exponentiation on Reconfigurable Hardware", IEEE Symposium on Computer Arithmetic, April, 1999.
[7] S.Lee, S.Kim and Y.Jeong, "Implementation of High-radix Exponentiator for RSA using CRT", Journal of The Korean Institute of Information Security and Cryptology, Vol. 10, No. 4, December, 2000.
[8] 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.
[9] Colin Walter, "Systolic modular multiplication", IEEE Trans. On Computers, Vol. 42, March 1993.
[10] M. Shand and J. Vuillemin, "Fast implementations of RSA cryptography", Proceedings of 11th IEEE Symposium on Computer Arithmetic, pp. 252~259, 1993.
[11] Cetin Kaya Koc, "Analysis of Sliding Window Techniques for Exponentiation", Computers and Mathematics with Applications, pp. 17~24, 1995.
[12] Jurjen N. Bos, Matthijs J. Coster "Addition Chain Heuristics", CRYPTO 89, 1989.
[13] Ömer Egecioglu and Cetin K. Koc, "Exponentiation using Canonical Recoding", Theoretical Computer Science, pp. 407~417, Vol. 129(2), 1994.
[14] A. D. Booth, "A signed binary multiplication technique", Q.J. Mech. Appl. Math, 4(2), pp. 236~240, 1951
[15] P. Montgomery, "Modular multiplication without trial division", Mathmatics of computation, Vol. 44, pp. 519~521, 1985.
[16] J. Quisquater and C. Couvreur, "Fast Decipherment Algorithm for RSA Publickey Cryptosystem", IEE Eletronics Letters, 18, pp. 905~907, October 1982.
[17] Samsung Electronics, "ASIC STD85/STDM85 0.5um High Density CMOS Standard Cell Library," Samsung electronics, September 1997.

## 〈著者紹介〉



**김 성 두 (Seong-doo Kim)**
2000년 2월 : 서울산업대학교 전자공학부 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
1999년 3월~현재 : 광운대학교 전자공학부 조교수
〈관심분야〉 컴퓨터 연산 알고리즘, ASIC 설계, 무선 통신, 정보보호