RSA 암호 시스템을 위한 고속 멱승 처리기

이석용 정용진
광운대학교 전자통신공학과, 서울 노원구 월계동 447-1

A High Performance RSA Modular Exponentiator with Pipelining

Seok-Yong Lee Yong-Jin Jeong
Dept.of Electronic Communication Engineering, Kwangwoon University

요 약

본 논문에서는 RSA 암호 시스템의 핵심 과정인 모듈로 멱승(Modular Exponentiation) 연산에 대한 새로운 하드웨어 구조를 제시한다. 기존의 몽고메리 알고리즘을 사용하였지만 다른 논문들이 Dependence Graph 를 수직으로 매핑(Mapping)한 것과는 달리 여기서는 수평으로 매핑하여 1 차원 선형 어레이(linear array) 구조를 구성하였다. 본 논문에서 사용한 방법의 장점은 결과가 시리얼(serial)로 나와서 바로 입력으로 들어갈 수 있기 때문에 100%의 처리율(throughput)을 이룰 수 있고, 수직 매핑 방식에 비해 절반의 클럭 횟수로 연산을 해낼 수 있다는 점이다. 또한 내부 계산 구조의 지역성(Locality), 규칙성(Regularity) 및 모듈성(Modularity) 등으로 인해 실시간 고속 처리를 위한 VLSI 구현에 적합하다.

1. 개 요

최근 몇 년간 전자상거래 등에 필요한 정보암호화(Encryption), 전자 서명 및 인증(Electronic Sign and Authentica tion)에 대한 관심과 수요가 급증하고 있다. 공개키 방식 암호 시스템은 이러한 요구를 충족시키는 기본 알고리즘을 갖추고 있는데, 그 대표적인 것이 RSA 알고리즘이다. RSA 알고리즘은 1024 비트 이상의 큰 정수(key)를 기반으로 한 모듈로 연산에 의해 수행된다. RSA 를 위한 모듈로 연산은 내부에 곱셈과 나눗셈이 복합되어 있어 계산 구조가 복잡하고 워드 사이즈가 크기 때문에 고속 실시간 처리를 위한 하드웨어 모듈의 구현이 어렵다. 그러나, 근래 들어서는 VLSI 설계 기술의 발달과 함께 다양한 연산 알고리즘들이 연구되어 고속 RSA 모듈 구현에 대한 연구가 활발해지고 있으며[1][3][4][5], 하드웨어로 구현 했을 경우 키의 안전성 면에서도 소프트웨어보다 월등하다는 장점을 가진다. 본 논문에서는 고속 RSA 모듈 구현을 위한 파이프라인을 이용한 새로운 하드웨어 구조를 제안한다.

2. RSA 알고리즘과 모듈로 곱셈 연산

RSA 암호시스템 알고리즘의 핵심이 되는 암호화/복호화 연산은 식 1 과 같다.

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

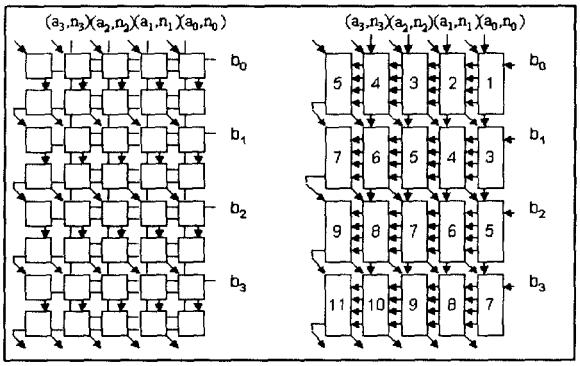
위의 연산에 사용되는 계수(modulus) N 은 두개의 큰 소수의 곱으로 이루어 지며, 공개키 E 나 비밀키 D 에 대해 모듈로 멱승을 취함으로써 암호화 (C) 및 복호화 (M)가 이루어 진다. 따라서 RSA 연산의 핵심은 모듈로 멱승(Modular Exponentiation)이며 이는 곱셈의 연속임을 알 수 있다. 또한 모듈로 곱셈은 연속된 덧셈연산으로 수행될 수 있는데, 곱셈 연산시 곱셈을 먼저 한후 모듈로 연산을 하는 modulo reduction after multiplication 방식과 곱셈중에 비트수를 감소시키는 modulo reduction during multiplication 방식이 있다. 곱셈기의 하드웨어 구현시 전자는 리소스를 많이 차지 하기 때문에 주로 후자의 방법을 사용한다. 이 때 modulo reduction 방법에는 MSB(JB algorithm) 우선방식[3]과 LSB(Montgomery algorithm)우선방식[2]이 있는데 연산시에 캐리의 지연을 고려치 않아도 되는 장점이 있는 몽고메리 알고리즘을 주로 사용한다.

몽고메리 알고리즘을 이용한 모듈로 곱셈 $V = MonPro(A, B) = A \cdot B \bmod N$ 의 계산은 식 2 와 같다. 식 2 에서 보는 바와 같이 몽고메리 알고리즘은 중간 결과 P_j 가 홀수이면 계수 N 을 더하여 짝수로 만든 후 right shift 하는 방법으로 비트수를 줄여나가는데 이는 RSA 암호화 방식에서 계수 N 이 홀수이기 때문에 가능하다.

$$\begin{aligned}
&(i)\quad P_{(0)} = 0 \\
&(ii)\quad P_{(j+1)} = (P_{(j)} + b_j \cdot A) \cdot 2^{-1} \bmod N \\
&(iii)\quad V = 2^k \cdot P_{(k)} \bmod N \\
&\qquad (P_{(k)} = 2^{-k} \cdot V \bmod N) \qquad (2) \\
&\text{Rewriting(ii)} : \\
&(ii)'\quad P^*_{(j+1)} = (P_{(j)} + b_j \cdot A) \\
&\qquad \text{if } P^*_{(j+1)} \text{ is odd}, \ P_{(j+1)} = (P^*_{(j+1)} + N) \cdot 2^{-1} \\
&\qquad \text{if } P^*_{(j+1)} \text{ is even}, \ P_{(j+1)} = (P^*_{(j+1)}) \cdot 2^{-1}
\end{aligned}$$

마지막 결과값은 기수(radix)로 k 번 나눈 결과 이므로 후처리(Post Processing)로 기수를 k 번 곱해야 정확한 결과를 얻게 된다. 즉, $R^* = A \cdot B \cdot 2^{-k} \bmod N$ 이므로 후처리로 $R^* \cdot 2^k \bmod N$ 을 수행하여 원하는 결과값 $R = A \cdot B \bmod N$ 을 얻는다.

앞에서 밝힌 바와 같이 RSA 의 주된 연산인 모듈로 멱승은 연속된 모듈로 곱셈으로 구성되므로 효율적이고 빠른 RSA 암호 시스템 구현의 관건은 모듈로 곱셈기의 설계에 있음을 알 수 있다. 몽고메리 곱셈 알고리즘의 내부계산구조를 보여주는 데이터 종속그래프(DG:Data Dependence)는 그림 1 과 같다. 이를 이용하여 모듈로 곱셈기를 구현할 때 다른 논문들[3][4][5]이 수직으로 매핑한 것과는 달리 수평으로 매핑하였는데 수직매핑과의 차이점은 다음과 같다. 먼저, 수직 매핑은 데이터 흐름(Data Flow)이 양방향이다. 결과는 2k 부터 나오지만 옆에서 들어가는 입력은 2k-1 에 걸쳐서 한 클럭씩 쉬면서 들어가므로 100% 처리율(throughput) 을 구현하기



(그림 1) 몽고메리 알고리즘의 Dependence Graph

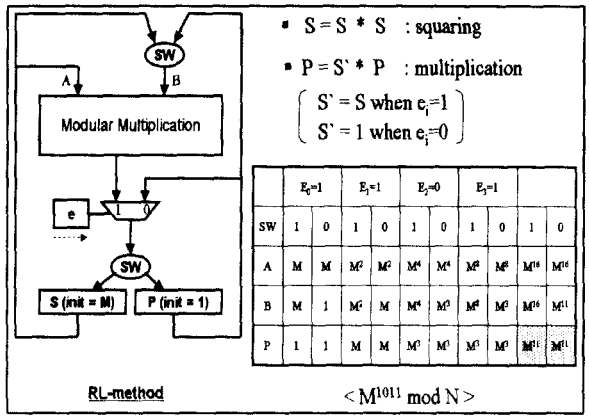
가 어렵다. 100% 처리율을 이루기 위해서는 [5]의 경우처럼 인터리빙방식으로 사이사이에 제곱과 곱셈에 들어가는 인자를 스위칭하면 되지만 그럴 경우 입력값을 저장할 레지스터가 2 개 필요하고 또한 결과가 한클럭 딜레이를 가지고 오버랩(overlap)되어 나오기 때문에 결과 또한 2 개의 레지스터에 나누어서 저장해야 한다. 결과적으로 출력이 병렬로 나오는 효과를 가져온다. 반면, 수평 매핑은 데이터 흐름(Data Flow)이 단방향이다. 결과는 수직매핑과 똑같이 2k 부터 나오지만 옆에서 들어가는 입력은 매 클럭마다 하나씩 k 의 구간에 들어가므로 나머지 k 의 시간에 다른 계산(곱셈 또는 제곱)의 인자를 넣을 수 있다. 또한 위에서 병렬로 들어가는 입력은 1,3,5,7,- 의 순서로 제 시간에 들어가기만 하면 되므로 레지스터와 멀티플렉서(Multiplexer)를 써서 시리얼로 나오는 출력을 바로 받을 수 있게 구현하였다. 따라서 수직매핑에 비해 콘트롤이 한결 쉬워지며 100% 처리율의 구현도 용이한 장점이 있지만, 한편으로는 파이프라인을 위한 레지스터가 증가하는 단점도 존재한다.

모듈로 곱셈을 한번만 수행할 때는 중간값이 k+2 비트까지 나오므로 캐리를 염두에 두면 k+1 비트로 충분하지만[4] 결과값은 N 보다 커질 경우가 생긴다. 이렇게 되면 멱승시 다음 입력값으로는 적합하지가 않기 때문에 여분의 ' 0' 를 입력에 패딩(Padding)하는 방법을 사용하여 k+3 비트로 입력을 넣어주면 결과값이 항상 입력조건에 부합하게 되어 멱승시에 중간 결과값에 대한 추가의 reduction 과정을 거칠 필요가 없어지게 된다.[5] 그러므로, 여분의 3 비트를 추가하면 3 cycle 의 클럭이 더 들어가지만 다음 입력조건에 부합되기 때문에 쉬는 시간(Idle time)없이 바로 다음 계산의 입력으로 들어갈 수 있는 장점을 가진다.

위와 같이 수평으로 매핑한 후에 2 차원으로 구현하면 하드웨어 복잡도(Hardware complexity)가 k^2 이 되어 상당한 리소스가 필요하기 때문에 하드웨어 복잡도가 k 가 되도록 한 행(row)만으로 파이프라인 구조의 1 차원 선형 어레이 곱셈기(linear array multiplier)를 구현하였다.

3. 모듈로 멱승 알고리즘

멱승을 계산하는 binary method 는 지수(Exponent)를 스캔하는 방향에 따라 MSB 부터 스캔해 나가는 LR-method(Horner' s rule)와 LSB 부터 스캔해 나가는 RL-method 의 두가지 방법이 있다.[1] 수평으로 매핑한 경우 입력이 (k+3) 클럭에 걸쳐 들어간후 결과가 2(k+3)부터 나오기까지 (k+3)의 클럭을 쉬기 때문에 이를 이용하기 위하여 그림 2 처럼 제곱과 곱셈이 서로 독립적으로 병렬 처리될



(그림 2) RL-method

수 있는 RL-Method 를 이용하여 멱승을 구현하였다. 그림 2 에서 보는 바와 같이 멱승은 제곱과 곱셈을 번갈아 함으로써 연산이 가능하기 때문에, 지금까지 논의해 온 곱셈기를 그대로 사용, 공유하여 하드웨어 리소스도 적게 차지 하게 된다.

몽고메리 알고리즘은 곱셈시 후처리(postprocessing)를 꼭 해주어야 하기 때문에 적은 양의 곱셈시에는 오히려 비효율적이다. 그러나, 멱승과 같이 여러 번의 곱셈을 수행할 경우에는 전처리(preprocessing)와 후처리를 멱승의 처음과 마지막에 1 회만 해주면 되기 때문에 많이 사용되는 방법이다. 이에 대한 모듈로 멱승 계산은 그림 3 과 같이 이루어 질 수 있다.

```
/* (z =r^2 mod N) is precalculated */
function MonExp(M, E, N) { N is an odd number, r=2^(k+3)}
M* = M*r mod N = MonPro(M, z)
X* = 1*r mod N = MonPro(1, z)
for i=0 to k-2 do
    if e_i =1 then X* = MonPro(M*, X*)
    M* = MonPro(M*, M*)
if e_(k-1) = 1 then X*= MonPro(M*, X*)
X = MonPro(X*, 1)
Return X
```

(그림 3) (k+3)비트에 대한 모듈로 멱승 알고리즘

$MonPro(A,B) = A \cdot B \cdot 2^{-(k+3)} \bmod N$ 이므로 올바른 중간 결과값을 얻기 위해서는 후처리로 $2^{(k+3)}$ 의 곱셈이 더 필요하다. 이를 중간 결과값들에 대해 매번 수행하게 되면 매우 비효율적이므로 그림 3 처럼 초기 입력값을 $2^{2(k+3)}$ 으로 미리 곱해 놓고 계산하는 것이 편리함을 알 수 있다. 모든 중간 결과값은 $R^* = MonPro(A^*, B^*) = A \cdot B \cdot 2^{(k+3)} \bmod N$ 처럼 $2^{(k+3)}$의 factor 를 가지게 되므로 최종 결과값에 간단히 1 을 곱하는 후처리를 해 주면 원하는 결과값을 얻게된다.

Preprocessing :

$$A^* = MonPro(A, 2^{2(k+3)}) = A \cdot 2^{(k+3)} \bmod N$$

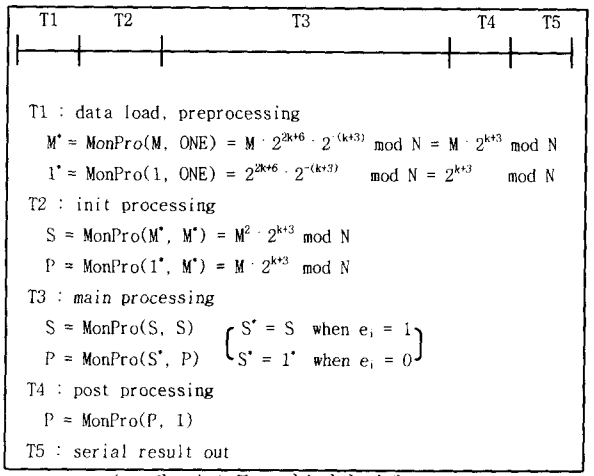
$$B^* = MonPro(B, 2^{2(k+3)}) = B \cdot 2^{(k+3)} \bmod N$$

Postprocessing :

$$R = MonPro(R^*, 1) = R^* \cdot 1 \cdot 2^{-(k+3)} \bmod N$$

$$= A \cdot B \bmod N$$

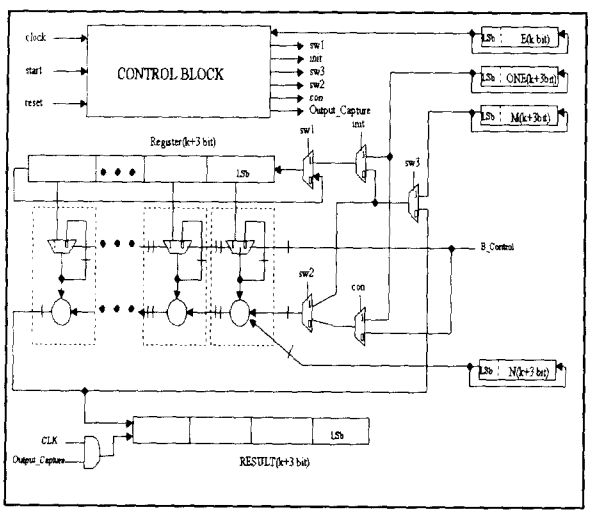
위의 과정을 입력값을 받아서 결과가 출력될 때까지의 전체 RSA 계산 프로세스(Process)로써 각 구간별로 나누면 그림 4 와 같



(그림 4) 모듈로 멱승기의 전체 Process

다. T1 동안에는 각 키 데이터들을 레지스터에 로드하고 앞에서 기술한 것처럼 중간의 후처리를 없애기 위해서 초기 입력 데이터에 2^{k+3} 의 factor 를 붙이는 과정이고, T2 는 전처리의 결과값을 M 과 ONE 레지스터에 시리얼로 로딩하는 프로세스이다. 이 때 M 과 ONE 레지스터는 곱셈기를 거쳐 계산된 값을 받아야 하기 때문에 다음 계산에 입력으로 들어가기 전에 k+3 클럭의 쉬는 시간이 존재하게 된다. T3 는 주된 연산인 모듈로 멱승을 수행하고, T4 는 후처리로 원하는 결과값을 얻어내기 위해 최종 결과값에 1 을 곱하는 과정이며, 그 결과가 T5 에 시리얼로 나오게 된다.

지금까지 상술한 매핑과정과 프로세스에 따라 구현한 멱승기의 전체구조는 그림 5 와 같다.



(그림 5) 멱승기의 전체 구조

모든 레지스터는 T1 이 시작하자 마자, 데이터가 로딩(loading)되어야 하므로 병렬 로딩기능이 있어야 하고 특히, M 과 ONE 레지스터는 Preprocessing 된 값을 다시 받아야 하므로 시리얼 로딩 기능도 가져야 한다. 구현한 시스템은 결과 값이 시리얼로 나오고 멱승시 입력으로도 시리얼로 들어가게 된다. 그러나, 위에서 들어가는 데이터는 DG 에서 보는 것처럼 1,3,5,7,..의 적절한 시간에 각 PE 에 도착하여 k+3 클럭 동안 값을 유지하고 있으면 되기 때문에 위에서처럼 쉬프트 레지스터와 멀티플렉서(Multiplexer)를 사용하여 구현하면 된다.

4. 결 론

본 논문에서 제시한 1024 비트 RSA 시스템 멱승기의 하드웨어 리소스와 전체 계산을 위한 클럭 수는 표 1 과 같다. 이를 근거로 성능을 계산 하면 클럭 주기를 10ns 라고 했을 때 1024 비트의 워드 사이즈에 대해서 약 50kbps 의 성능(performance)을 나타내고 있다.

클럭수	T1 : 1 + 2(k+3) <= load + preprocessing T2 : 2(k+3) + (k+3) <= init processing,(k+3)idle time T3 : 2(k+3)(k-1) <= main processing T4 : 2(k+3) <= post processing T5 : 2(k+3) <= result out T_{total}: $2k^2$ + 13k + 22	
하드웨어 리소스	● 각 PE 당 : AND Gate 1 개 MUX(2:1) 4 개 Full Adder 2 개 Flip Flop 15 개	● PE 외부 구성요소 : (k+3)bit Register 6 개 MUX(2:1) 5 개 Flip Flop 3 개 State Machine 1 개

(표 1)

이러한 성능은 지금까지 국내외에서 칩으로 실제 구현된 어떠한 RSA 칩과도 대등 혹은 더 우수한 것이다.

본 논문에서는 각 비트당 하나의 PE(Processing Element) 로 구현하였으며, 클럭 사이클 타임(clock cycle time) 을 결정하는 요소인 최장 지연 패스(Critical Path)가 두개의 FA(Full Adder)와 세개의 MUX 로 이루어져 있어 수백 MHz 의 클럭까지도 구현이 가능하지만 ASIC 설계시의 클럭 분배 문제를 위해 100MHz 로 시뮬레이션하였다. 이를 하이래딕스(High Radix)를 사용하면 PE 당 3 비트 혹은 4 비트(r = 8 or 16)까지 구현이 가능하여 클럭 사이클 수를 현저히 줄일 수 있어 더 한층 성능을 향상(약 10 배 ~ 20 배)시킬 수 있다. 속도를 수 Mbps 까지 올리기 위해서 하이래딕스를 사용하는 방법을 연구중이며 아울러 스마트 카드에의 적용을 위해 low frequ ency, low power 설계를 위한 연구가 진행중이다.

5. 참고 문헌

[1] Cetin Kaya Koc, " RSA Hardware Implementation ", RSA Laboratories, August 1995.

[2] P. Montgomery, " Modular multiplication without trial division", Mathmatics of computation, vol. 44, pp.519-521, 1985.

[3] YJ.Jeon g and W.Burleson, " VLSI array algorithms and architectures for RSA modular multiplication", IEEE Tran. On VLSI Systems, vol.5, pp.211-217, June 1997.

[4] Colin Walter, " Systolic modular multiplication ", IEEE Tans. On Computers, vol.42, March 1993.

[5] Tho mas Blum, Christof Paar, " Montgomery Modular Exponentiation on Reconfigurable Hardware ", IEEE Symposium on Computer Arithmetic, April 14 -16, 1999, Adelaide, Australia.