修 士 論 文

高速モンゴメリ乗算回路に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

吉原 智明

2003 年 3 月

修 士 論 文

高速モンゴメリ乗算回路に関する研究

指導教官　中野浩嗣 助教授

審査委員主査　中野浩嗣 助教授
審査委員　浅野哲夫 教授
審査委員　金子峰雄 助教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

110128 吉原 智明

提出年月: 2003 年 2 月

概 要

近年, 通信ネットワークを介して重要な情報が流通するようになり, その内容を外部から盗聴されないようにするための暗号化技術が必要になってきている.

その中にあって, RSA 暗号や楕円曲線暗号などに代表される公開鍵暗号システムは秘密通信や安全な電子商取引に欠くことのできない重要なセキュリティー技術の一つである. この公開鍵暗号の多くは, それまで知られてきた秘密鍵暗号方式よりも計算量が多く, 暗号化復号化に膨大な計算時間を必要とする. また, これらの暗号は剰余類による有限体上の演算を用いているので剰余付き乗算 $A \cdot B \bmod n$ という演算が繰り返しで実行される.

本研究では, 主な公開鍵暗号方式で繰り返し行われている大きな整数に対する剰余演算に着目する. そして, その演算部分を高速化することができれば, それによって, 公開鍵暗号用のハードウェア全体を高速化することができると考える.

本論文では, いろいろな剰余演算アルゴリズムの中からモンゴメリ乗算アルゴリズムを取り上げる. そして, FPGA という書き換え可能なハードウェアをもちいて, 剰余演算の高速化を図った. また, ソフトウェアでのモンゴメリ乗算と比較することで, 性能を評価した.

目 次

第1章　はじめに

1.1　研究の背景と目的

公開鍵暗号システムは, ネット上で秘密通信や安全な電子商取引に欠くことのできない重要なセキュリティー技術の一つである. その代表として, RSA 暗号, エルガマル暗号, DH 鍵交換などが挙げられ, 大きな整数の因数分解や離散対数問題の難しさをその安全性の根拠としている. これらは剰余類による有限体上の演算を用いてるが, それを楕円曲線上の有理点の加減算やスカラ倍算に置き換えて実現する楕円曲線暗号に関しても活発な研究が行われている.

公開鍵暗号の多くは大きな整数に対する乗剰余演算を繰り返すため, 膨大な計算時間を必要とする. 近年の CPU の高速化により, ソフトウェア実装でも実用的な速度が得られるようになってきているが, スマートカードを代表とするローエンド向けの 8 ビットや 16 ビットの組み込み CPU では処理時間が長く実用に耐えない. また SSL サーバーなど暗号処理が集中する用途において, ソフトウェア処理では十分なパフォーマンスを得ることができない. そのためローエンド, ハイエンド双方において暗号処理専用ハードウェアが必要となる.

RSA 暗号などのハードウェアはいろんな文献 [3] で見受けられる. 国内でも高速な RSA 暗号の LSI の開発 [10] は盛んに行われている. そして, 1,024 ビットの RSA 処理を約 2.4 ms で実行できる見通しを立てている. このように暗号システムのハードウェア化による高速化は研究され尽くしされているかに見える. しかし, FPGA 上でそれを実現し, ソフトウェアや既存のハードウェアと比較し, ソフトウェアよりも高速で, 既存のハードウェアとあまり変わらない速度を示すことができるなら, FPGA で実現したほうがメリットが大きいと考える. それは, 高度成長を続ける情報化社会においては, RSA 暗号などの信頼度はいつ弱くなるか分からない. その時点で, 暗号システムは, 鍵長を長くしたり, 暗号自体の変更を余儀なくされるだろう. その時, 新しい鍵長や暗号に合わせてハードウェアを再構築できる FPGA はとてもそれに向いたデバイスといえる.

そのために, 本研究では FPGA を用いて, 公開鍵暗号などで必要な剰余演算を高速に行えるハードウェアを実装し, FPGA でどの程度高速なものを構成できるかを示し, 暗号用に適しているのかを考察したい.

1.2 本論文の概要と流れ

本論文では, まず第 2 章で, 今日では必須の技術となった公開鍵暗号のシステムについて RSA 暗号を用いて述べる. またその公開鍵暗号に使用される演算処理である, 剰余演算についてのいろいろなアルゴリズムを 3 章で述べる. 次に, 4 章で実際にハードウェア化をおこなうための, モンゴメリ乗算アルゴリズムについて述べる. そして 6 章, 7 章でモンゴメリ乗算アルゴリズムの実装を行った結果を示し, その結果について評価を行う.

第2章　公開鍵暗号

2.1　公開鍵暗号の原理

暗号とは, 送信者が情報を秘匿して限定された受信者だけに伝達する技術である. 暗号は, 送信者と受信者が共通の鍵を用いて暗号化と復号化を行う秘密鍵暗号と, 送信者と受信者が異なる鍵を用いて暗号化と復号化を行う公開鍵暗号に分類できる. 公開鍵暗号では暗号鍵を公開しても復号鍵を求めることが計算量的に実行不可能であることが重要である.

秘密鍵暗号では鍵を秘密に配送する必要があるのに対し, 公開鍵暗号では暗号鍵が公開されても安全性が損なわれないので, 鍵の配送と管理が秘密鍵暗号の場合と比較して容易となる.

公開鍵暗号は次の三つの段階から構成されている. ここでは, 送信者 A が秘密にメッセージ m (平文ともよぶ) を受信者 B に送信する場合を考える. まず, 受信者 B は公開鍵 e_B を公開ファイルから読み出して, 暗号文 c を $c = E_{e_B}(m)$ により計算して B に送信する. 最後に, B は秘密鍵 d_B を用いて, $D_{d_B}(c) = D_{d_B}(E_{e_B}(m)) = m$ によりメッセージを復号する.

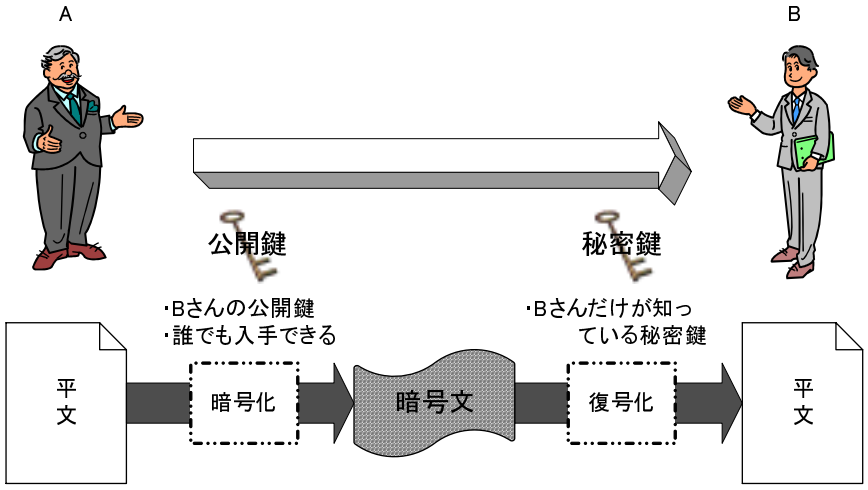

図 2.1: 公開鍵暗号

2.2 RSA暗号

公開鍵暗号として最も有名なものにRivest-Shamir-Adlemanという人たちが発明した, RSA暗号がある. これは大きい数の素因数分解が現在の段階では極めて困難である事実を利用するものである. 彼らは論文[1]の中で, 例えば60桁の2つの素数の積からなる120桁の数を元の素数がわからないで因数分解するには, 現在（1977年）最高の電算機をもってしても10^{19}年かかるであろうといっている.

公開する暗号の鍵は3つであり, 文章を数字文に換字するコードと, 2つの大きな素数pとqの積rと, ある数sである. 暗号文を作る方法は原文をコードによって数字暗号に変え, rより小さい数の群で区切り, 各群をはじめから$T_1, T_2, T_3 \cdots$とする. 次に各群ごとに$T_i^s \equiv C_i (\mathrm{mod} r)$の計算は手作業では大変な作業であるが, 電算機を使用するとさほど困難ではない. sの桁数が大きくなると大変なように思われるが, 作業はTの次にT^2, 次に$T^4, T^8, T^{16}, \cdots$というように, $T^{2\alpha}$でαが$2^\alpha \leq s \leq 2^{\alpha+1}$となる$\alpha$まで計算し, この$\alpha+1$個の$T^{2\alpha}$を組み合わせて掛ければ$T^s$を求めることができるから, 掛け算の回数は$\alpha \sim 2\alpha$である. そこで$\alpha \approx \log_2 s$から$s$が増加しても掛け算の回数が驚くほど増加することはない.

この論文の例をそのまま引用すると, 換字表は$A \to 01, B \to 02, C \to 03, \cdots, Z \to 26$, スペースを00で表わす. $r = 2,773 (= 47 \times 59)$, $s = 17$, で原文 “ ITS ALL GREEK TO ME ” を暗号化すると次のようになる.

第一次暗号

$$0920 \quad 1900 \quad 0112 \quad 1200 \quad 0718 \quad 0505 \quad 1100 \quad 2015 \quad 0013 \quad 0500$$

第一語目0920の暗号化の手順を見てみると

$$T_1^0 \equiv 1, \quad T_1^1 \equiv 920, \quad T_1^2 \equiv (920)^2 \equiv 635,$$
$$T_1^4 \equiv (635)^2 \equiv 1,140, \quad T_1^8 \equiv (1,140)^2 \equiv 1,836,$$
$$T_1^{17} \equiv (1,836)^2 \cdot 920 \equiv 948 \pmod{2,773}$$

同様の手順で全暗号文は以下のようになる.

$$0948 \quad 2342 \quad 1084 \quad 1444 \quad 2663 \quad 2390 \quad 0778 \quad 0774 \quad 0219 \quad 1655$$

この暗号を復号するには復号鍵tが分かっていればよい. $C^t \equiv T \pmod{r}$として暗号文より第一次暗号文が得られる. 上の例の場合, $t = 157$であるこのtはrが因数pとqに分解できれば, 以下のように求めることができる.

$$s \cdot t \equiv 1 \pmod{(p-1)(q-1)}$$

しかし, pとqが不明ならば簡単には求めることはできない.

では, このtを用いればなぜ簡単に復号できるのかを説明すると

一般にフェルマーの定理

$$a^{p-1} \equiv 1 \pmod{p} \quad (\ p \text{は素数で} (a,p) = 1\ )$$

からオイラーの定理

$$a^{\varphi(r)} \equiv 1 \pmod{r} \quad ((a,r)=1, \varphi(r) \text{はオイラー数})$$

がよく用いられる. オイラーの定理の両辺を k 乗すると

$$a^{k\cdot\varphi(r)} \equiv 1 \pmod{r}$$

となる. さらに両辺に a を乗じて

$$a^{k\cdot\varphi(r)+1} \equiv a \pmod{r}$$

となり

$$a\left\{a^{k\cdot\varphi(r)}-1\right\} \equiv 0 \pmod{r}$$

は a が r の倍数のときも成り立つ.

オイラー数 $\varphi(r)$ は r を素因数に分解して $r = p_1^{\alpha}\cdot p_2^{\beta}\cdot p_3^{\gamma}\cdots$ となるときは

$$\varphi(r) = r\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\left(1-\frac{1}{p_3}\right)\cdots$$

となることが証明されている. $r = p\cdot q$ であれば

$$\varphi(r) = (p-1)(q-1) = \varphi(p)\cdot\varphi(q)$$

$s\cdot t \equiv 1 \pmod{(p-1)(q-1)}$ は $s\cdot t \equiv 1 \pmod{\varphi(r)}$ と書け, また $s\cdot t-1$ は $\varphi(r)$ の倍数であるから, ある整数 k で $s\cdot t-1 = k\cdot\varphi(r) = k\cdot\varphi(p)\varphi(q)$ と書ける. 故に

$$s\cdot t = k\cdot\varphi(p)\cdot\varphi(q)+1.$$

いま暗号文 C は $C \equiv T^s \pmod{r}$ で求めたものであるから $C^t \equiv (T^s)^t \equiv T^{s\cdot t}$ となる. そこで

$$T^{s\cdot t} = T^{k\cdot\varphi(p)\cdot\varphi(q)+1} = T^{k'\cdot\varphi(p)+1} \quad (k' = k\cdot\varphi(q)),$$

$$T^{k'\cdot\varphi(p)+1} \equiv T \pmod{p}$$

は T が p と互に素な場合と T が p の倍数の場合に成立する. 同様に

$$T^{k't\cdot\varphi(p)+1} \equiv T \pmod{q}$$

も成立するので

$$T^{k\cdot\varphi(p)\cdot\varphi(q)+1} \equiv T \pmod{p\cdot q = r}$$

すなわち, $C^t \equiv T^{s\cdot t} \equiv T \pmod{r}$ が成立して復号可能であることが分かる.

第3章　剰余演算

前章において, RSA 暗号に剰余演算が用いられていることを述べてきた. この章では, 剰余演算を効率よくとくアルゴリズムを紹介する. ここにあるほとんどが, ハードウェアでも試された方法である [4][5][9].

法 N における整数 X の剰余とは, X を整数 N で割った余りであり, $X \bmod N$ と書く. 集合 $\mathbf{Z}_N = \{0, 1, \ldots, N-1\}$ の要素 A, B に対して, 和および積をそれぞれ整数 A, B の和および積の剰余で定義すると, N が素数の場合には, $\mathbf{Z}_N$ は素体 $\mathbf{F}_N$ になり, 合成数の場合には環になる.

一般的な剰余演算の四則演算については以下のようになっている.
$0 \leq A, B < N$ の数に対して

$$A + B \bmod N = \begin{cases} A + B & A + B < N \text{ の場合} \\ A + B - N & \text{上記以外} \end{cases}$$

$$A - B \bmod N = \begin{cases} A - B & A - B \geq 0 \text{ の場合} \\ A - B + N & \text{上記以外} \end{cases}$$

$A \cdot B \bmod N = R$　ここで $A \cdot B = Q \cdot N + R, 0 \leq R < N$ をみたす Q が存在すること.

$A^{-1} \bmod N = A'$　ここで $A \cdot A' = Q \cdot N + 1$ をみたす Q が存在すること.

3.1　乗算剰余

乗算剰余 $A \cdot B \bmod N$ を求める場合には大きく分けて, 2 通りの方法がある.

- $A \cdot B$ を計算してから剰余を求める.

  テーブル参照方式

  逆数方式

- $A \cdot B$ の部分積の剰余をとりながら, 最終的に $A \cdot B \bmod N$ を求める.

3.1.1 テーブル参照方式

あらかじめ計算しておいた剰余を剰余テーブルに格納しておいて, 必要に応じて逐次参照する方式をテーブル参照方式という. 一般にテーブルの大きさと計算時間にはトレードオフが存在する.

《 剰余アルゴリズム 1 》

$A \cdot B = Y$ は q^{2m}の程度の数であるから,

$$Y = \sum_{i=0}^{2m-1} y_i \cdot q^i$$

と表わすことができる. ここで

$$Y \bmod N = \sum_{i=0}^{2m-1} y_i \cdot (q^i \bmod N) \bmod N$$

が成り立つことに注目して, あらかじめ $(q^i \bmod N)(i = m, m+1, m+2, \ldots, 2m-1)$ を剰余テーブルに格納しておく.

【剰余テーブルを用いた剰余アルゴリズム】

Input: $q^{m-1} \leq N < q^m, \quad 0 \leq Y < q^{2m}$
前計算: $Q_i = q^i \bmod N \quad (i = m, m+1, m+2, \cdots, 2m-1)$
output: $W = Y \bmod N$

Step1: $W := Y$

Step2: $W := \sum_{i=0}^{2m-1} w_i \cdot q^i$ と q 進数展開する.

Step3: $W := \sum_{i=0}^{m-1} w_i \cdot q^i + \sum_{i=0}^{2m-1} w_i \cdot Q_i$

Step4: もし $W \geq q^m$ ならば Step2 へ.

Step5: $W < N$ となるまで $W := W - N$ を繰り返す.

Step6: W を出力.

このようにすると, レジスタ m^2 個分のサイズのテーブルを用いて, 乗算 1 回とほぼ同じ程度の処理で剰余を求めることができる.

《 剰余アルゴリズム 2 》

前述の 1 と同様に,

$$Y \bmod N = \sum_{i=0}^{m-1} y_i \cdot q^i + \sum_{i=m}^{2m-1} (y_i \cdot q^m \bmod N) \cdot q^{i-m} bmodN$$

も成り立ち, あらかじめ $(y \cdot q^m \bmod N)(0 < y < q)$ をテーブルに格納しておく.

【剰余テーブルを用いた剰余アルゴリズム 2】

Input: $q^{m-1} \leq N < q^m, \quad 0 \leq Y < q^{2m}$
前計算: $T_i = i \cdot q^m \bmod N \quad (i = 1, 2, \cdots, q-1)$
output: $W = Y \bmod N$

Step1: $W := Y$

Step2: $W := \sum_{i=0}^{k} w_i \cdot q^i$ と q 進数展開する. ここで k は $w_k \neq 0$ となる最大の数.

Step3: $W := \sum_{i=0}^{k-1} w_i \cdot q^i + T_{wk}$($T_{wk}$ はテーブルを用いて参照する $w_k \cdot q^m \bmod N$ の値)

Step4: もし $W \geq q^m$ ならば Step2 へ.

Step5: $W < N$ となるまで $W := W - N$ を繰り返す.

Step6: W を出力.

この方式は前のアルゴリズムと比べるとテーブルサイズがレジスタ $m \times (q-1)$ 個分とかなり大きくなる. それは通常は $m \ll q$ であるためだが, 加算だけで剰余演算を計算するという特徴をもつ.

3.1.2 逆数方式

次に, $Y = A \cdot B$ を求めたのち, 剰余テーブルを用いないで, あらかじめ計算しておいた $t = \lfloor q^{2m}/N \rfloor$ を利用した方式を紹介する.

$Y \bmod N$ は $Y - \lfloor Y/N \rfloor \cdot N$ により求められるが, $\lfloor Y/N \rfloor$ を実際に計算する必要がある.

そこで, $t = \lfloor q^{2m}/N \rfloor$ をあらかじめ計算しておき,

$$\lfloor Y/N \rfloor \approx \lfloor \lfloor Y/q^m \rfloor \cdot t/q^m \rfloor$$

により近似する. このとき $\lfloor X/q^m \rfloor$ は X の上位のレジスタ値そのものであるから, 右辺は一回の乗算で計算できる. 後は剰余を N 以下に補正するために可能な限りの N を引けばよい.

【逆数を用いた剰余アルゴリズム】

Input: $q^{m-1} \leq N < q^m, \quad 0 \leq Y < q^{2m}$
前計算: $t = \lfloor q^{2m}/N \rfloor$
output: $Y \bmod N$

Step1: $Y := Y - \lfloor \lfloor Y/q^m \rfloor \cdot t/q^m \rfloor \cdot N$

Step2: $Y < N$ となるまで $Y := Y - N$ を繰り返す.

Step3: Y を出力.

この方式は乗算 2 回分程度の演算を要するが, 剰余テーブルを利用しないのでメモリ容量を減らすことができるというメリットがある.

3.1.3 部分積方式

次に, 部分積を求めながら剰余をとる方式を紹介する.
$A \cdot B$ の積は B の q 進数展開をもちいて

$$A \cdot B = \sum_{i=0}^{m} A \cdot b_i q^i$$

と表わすことができる.
そこで

$$A \cdot B \bmod N = \left\{ \sum_{i=0}^{m} (A \cdot b_i q^i \bmod N) \right\} \bmod N$$

と, 各 i について部分積 $A \cdot b_i q^i$ をもとめるたびに剰余をとるアルゴリズムを考える.

【部分積を用いた剰余アルゴリズム】

Input: $N, 0 < A < N, 0 < B < N$
Output: $W = A \cdot B \bmod N$

Step1: $i := m - 1 \quad W := 0$

Step2: $W := W \cdot q + A \cdot b_i \quad i := i - 1$

Step3: $W := W \bmod N$

Step4: もし $i > 0$ ならば Step2 へ.

Step5: W を出力.

ここで Step3 の剰余演算として上に述べたどのアルゴリズムを用いてもよい. この方式では剰余演算を m 回行うことになるが, 計算途中の W の大きさが一定に抑えられるというメリットがある.

3.2 逆数演算

逆数演算 $A^{-1} \bmod N$ の計算方法はEuclid互除法を用いておこなう.

【Euclid互除法】

Step1: 与えられた整数 m, n に対して, $n_0 := m; n_1 := n$ とおく.

Step2: 除法 $n_{i-1} = q_i \cdot n_i + n_{i+1}, 0 \leq n_{i+1} < |n_i|$ を実行する $(i = 1, 2, \ldots)$.

Step3: $n_{i+1} = 0$ ならば $d := n_i$ で完了; そうでない場合は次の i について Step2 に戻る.

実際には $a := m; b := n; a \div b = q$ 余り r とし, $r \neq 0$ ならば $a := b; b := r$ として次へ進めばよい. 整商 q は不要だが, もしも

$$u_0 := 1, \quad u_1 := 0, \quad v_0 := 0, \quad v_1 := 1$$

として毎回除法のつど

$$u_{i+1} := u_{i-1} - u_i q_i; v_{i+1} := v_{i-1} - v_i q_i$$

を実行すれば, 最後に $n_{l+1} = 0, n_l = d$ となったとき, $u_l m + v_l n = d$ である. これにより m, n が互いに素なとき, $um \equiv 1 \pmod{n}$ である $u(n$ を法とする m の逆数) が計算できる.

3.3 べき乗剰余算のアルゴリズム

べき乗剰余演算とは $X^E \bmod N$ を求めることであり, これは以前の章で述べた RSA 暗号の核となる演算である [8].

3.3.1 べき乗演算のアルゴリズム

指数 E を $E = 2E_1 + E_2 \quad (E_2 = 0$ または $1)$ と分解した場合には, $X^E = (X^{E_1})^2 \cdot X^{E_2}$ が成り立つ. すなわち, X を E 回掛け合わせることをしなくても, 半分の E_1 回と高々2 回の乗算を行えばよいことがわかる. また, E_1 を分解してより少ない回数で求めることができる.

この原理を利用し一般化したものが, 「べきのバイナリ法」というものである. 指数 E の 2 進数展開 $E = \sum e_i \cdot 2^i$ を用いると

$$X^E = \prod X^{e_i \cdot 2^i}$$

が成り立つ. この等式に基づくアルゴリズムは以下のようになる.

【 べきの右向きバイナリ法 】

Input: $N, 0 < X < N, E$
output: $A = X^E \bmod N$

Step1: $A := 1$

Step2: $i := \lfloor log_2 E \rfloor + 1$

Step3: $A := A^2 \bmod N; \quad i := i - 1$

Step4: もし $e_i = 1$ ならば $A := A \cdot X \bmod N$ を繰り返す.

Step5: もし $i \neq 0$ ならば Step 3 へ.

Step6: A を出力.

このアルゴリズムは約 $log_2 E$ 回の 2 乗剰余, $e_i = 1$ なる i の数だけの乗算剰余を必要とする. 剰余演算としてはアルゴリズムを選ばないが, Step3 から Step5 のループの中では必ずしも N 未満の剰余を求める必要はない. N を法として合同である扱いやすい数 (たとえば q^m 以下の数) を求められるのであれば, 計算を続けることができる. 最終的に引き算や除算を行い正確な剰余を算出する. また, このアルゴリズムとは反対に, 指数 E の下位ビットから計算する「べきの左向きバイナリ法」も同様にして設計することができる.

3.3.2 モンゴメリ乗算

本節では剰余演算に対応する計算を高速に行う モンゴメリ乗算 [2] の紹介とそれを繰り返し用いることで, べき乗剰余を求める方法について述べる.

$N < R = q^m$ とし, N^2 程度の Y に対して $M = Y \cdot R^{-1} \bmod N$ を出力する Montgomery Reduction を最初に述べる.

【 モンゴメリ乗算 】

Input: Y
前計算: $V = -N^{-1} \bmod R$
output: $M = Y \cdot R^{-1} \bmod N$

Step1: $W_1 := Y \cdot V \bmod R$

Step2: $W_2 := Y + W_1 \cdot N$

Step3: $M := W_2/R$

Step4: $M < N$ となるまで $M := M - N$ を繰り返す.

Step5: M を出力.

まず, この演算が容易に計算されることを示してから, これを基本演算として用いたべき乗剰余アルゴリズムを紹介する. 以下では, 入力 Y に対するモンゴメリ乗算の出力 M を $MR(Y)$ と表わす.

$MR(Y)$ は一見, 入力 Y に対して Step 1 の乗算剰余, Step 2 の乗算および加算, さらに Step 3 の除算から成り立っているようにみえるが, 実際の演算処理は多くはない. まず Step 1 の乗算剰余は R をレジスタ単位 q のべきにしているため, 乗算結果の下位半分のみ計算すればよい. つまりハードウエアで演算するときにも余分な回路を必要としない. 次に $W_2 \bmod R = 0$ に注目すると Step 2 の乗算および加算の下位半分は全て 0 であることがわかる. したがって Step 2 の乗算および加算は上位半分だけを求めればよく, 上位半分が M となる (Step 3). したがって, 半分の乗算 2 回分で $MR(Y)$ は計算することができ, 結果として, N 程度の数を得ることができる.

このように容易に計算できる上記の演算をどのようにべき乗剰余演算の核として用いるかを考察する. まず, $W_2 \bmod N = Y$ が成り立つので, 出力 $M = W_2/R \bmod N$ が $Y \cdot R^{-1} \bmod N$ に等しくなる. 次に

$$A^{(R)} = A \cdot R \bmod N \longleftrightarrow A \bmod N$$

という対応を考える. R と N が互いに素の場合, これは一対一対応であり

$$MR(A^{(R)}) = A^{(R)} \cdot R^{-1} = A \bmod N$$

という関係にある. また,

$$MR(A^{(R)} \cdot B^{(R)}) = A^{(R)} \cdot B^{(R)} \cdot R^{-1} = (AB) \cdot R = (AB)^{(R)}$$

が成り立つ.

したがって, $Y = X^E \bmod N$ を計算するために, これに対応する $Y^{(R)}$ を計算量の少ない モンゴメリ乗算 を用いて求めてから, Y を求めることにする. 以下に, べきの 2 進数計算法に モンゴメリ乗算 を導入したべき乗剰余アルゴリズムを示す.

【 モンゴメリべき乗剰余アルゴリズム 】

Input: $N, 0 < X < N, E$
output: $Y = X^E \bmod N$

Step1: $A^{(R)} := R \bmod N \quad X^{(R)} := X \cdot R \bmod N$

Step2: $i := \lceil log_2 E \rceil$

Step3: $A^{(R)} := MR(A^{(R)} \cdot A^{(R)}); \quad i := i - 1$

Step4: もし $e_i = 1$ ならば $A^{(R)} := MR(A^{(R)} \cdot X^{(R)})$

Step5: もし $i \neq 0$ ならば Step3 へ.

Step6: $Y := MR(A^{(R)})$ を出力.

第4章　モンゴメリ乗算回路

4.1　アルゴリズム

前の章では一般的なモンゴメリ乗算のアルゴリズムを示してきた. これまでも, モンゴメリ乗算回路はハードウェアで実装されている [6][8]. ここでは実際にハードウェア化するアルゴリズムを細かく説明する.

法を $N(<R)$(ここで $R=2^{r\cdot m}$), 入力を $A(0<A<N), B(0<B<N)$, とする, ただし, A_0 とは A の基数を 2^r としたときの最小ビットである.

【 モンゴメリ乗算アルゴリズム 】

Input: $N, 0<A<N, 0<B<N$
前計算: $V=-N^{-1} \bmod 2^r$
Output: $M=A\cdot B\cdot R^{-1} \bmod N$

Step1: $M:=0 \quad i:=0$

Step2: $Q:=(M+A_i\cdot B_0)V \bmod 2^r$

Step3: $M:=(M+A_i\cdot B+Q\cdot N)$

Step4: $M:=M/2^r \quad i:=i+1$

Step5: もし $i<m$ ならば Step2 へ.

Step6: もし $N<M$ ならば M:=M-N

Step7: M を出力.

モンゴメリ乗算は, 法の倍数を加算することで, 下位の bit を 0 にして, 右シフトさせながら除算をすつというしくみである. つまり, 上の変数を用いると, 法 N を Q 倍して $M+A_i\cdot B$ と加算し, 下位 $r$bit を 0 にすることで, 剰余値を保存しながら $r$bit 右シフトする演算である. そのための前計算として, Step2 で Q を求めるために必要である V を求めておく. V は, 下位 $r$bit を 0 にするための $M+A_i\cdot B+Q\cdot N=0 \pmod{2^r}$ という式から, $Q=(M+A_i\cdot B)(-N^{-1}) \pmod{2^r}$ と変形するとでてくる, 法 N が決まればすぐに求めることができる $-N^{-1} \pmod{2^r}$ の値である.

4.1.1 初回の Step2∼4

Step2 からの説明は分かりやすいように図で説明する.

まず, $A \cdot B$ の部分積である $A_0 \cdot B$ を求める. 次に, 一回目の Step2 では $M = 0$ であるから, の値に V を掛けることで, 法 N を何倍すれば, 加算して 0 になるかという Q の値を求めている. その際には 2^r を法とする剰余演算であるから, B_0 だけを掛ければ良いのだが, Step3 で $A_0 \cdot B$ の値を使用することになるので実際には B のビット全てを掛け合わせる. そして, 2^r での除算をしなくても, $A_0 \cdot B$ の下位 $r$bit だけの値を取り出すことで, 簡単に剰余演算となる.

次に, Step4 では, すでに値が 0 になっている下位 $r$bit をビットシフトで取り除く. つまり, 初めから下位 $r$bit をのぞいた bit のみをレジスタに入れるようにすればよい.

アルゴリズムを走らせて, 一回目の Step2, Step3, Step4 の図が, 図 4.1 である.

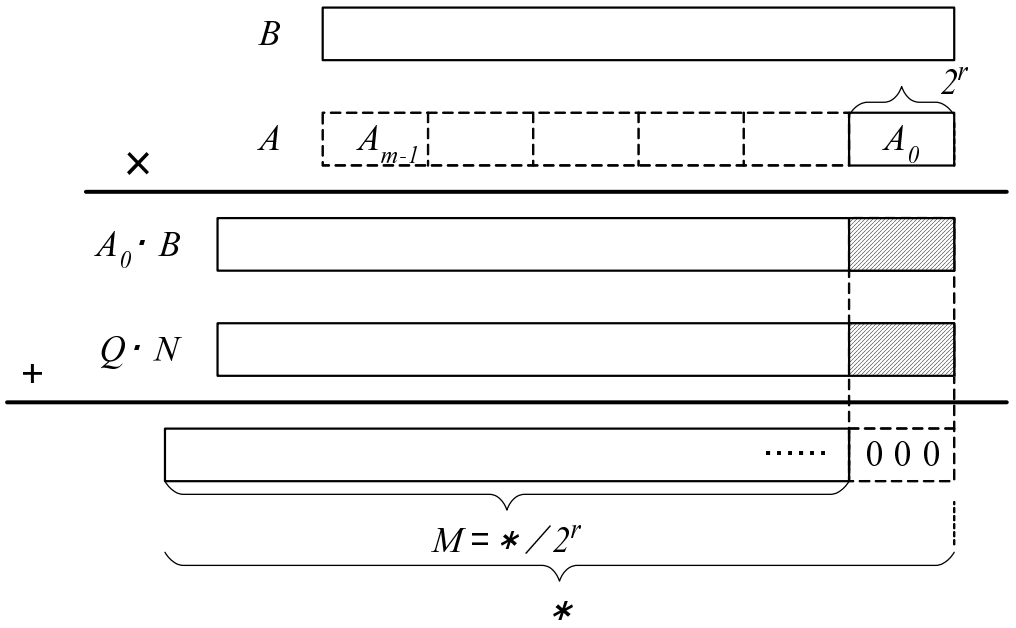

図 4.1: Step2 ・ Step3 ・ Step4

4.1.2 Step2∼Step4 の繰り返し

下の図 4.2 では 2 回目以降の Step2, Step3, Step4 の様子を描いている.

Step2 では, $(M + A_i \cdot B_0) - N^{-1} \bmod 2^r$ という演算で下位 r ビットが 0 になるような Q が求められる. Step3 では $Q \cdot N$ を求め, $M + A_i \cdot B$ の部分は Step2 で計算が終わっているのでそれと加算を行う. 2^r で割った新しい M を $A \cdot B$ の部分積と加算しながら, 上の図の動作を i が $m - 1$ になるまで繰り返す.

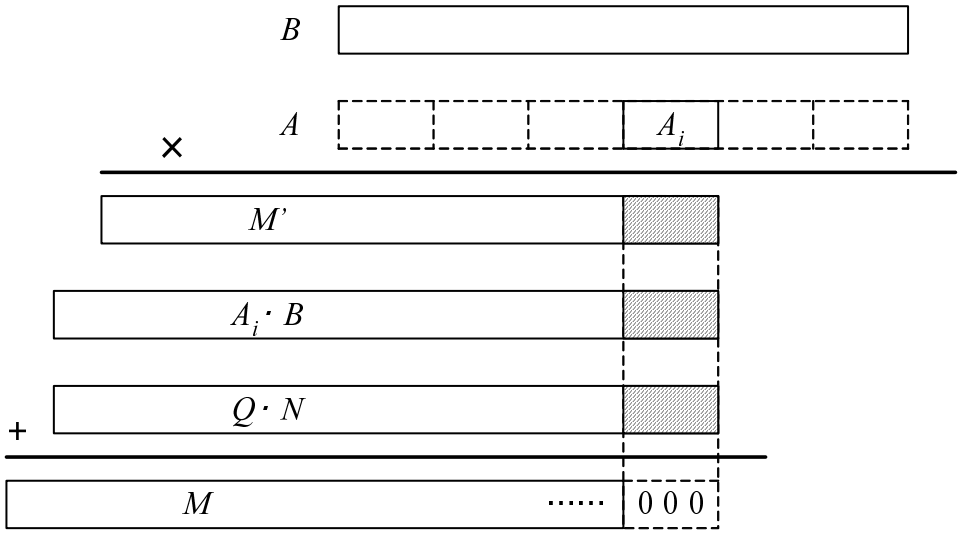

図 4.2: Step2 〜 Step4 の繰り返し

4.1.3 Step 全体

下の図はの Step 全体を簡単に表わしたものである.

このように法の倍数を R ビット分 0 にするために m 回足し合わせることで, 結果として $A \cdot B \cdot R^{-1} \bmod N$ を出力できる. これは剰余演算では法の倍数を足しても値が保存されることを利用している. ちなみに, 図の $\alpha \sim \zeta$ は倍 Q のことである.

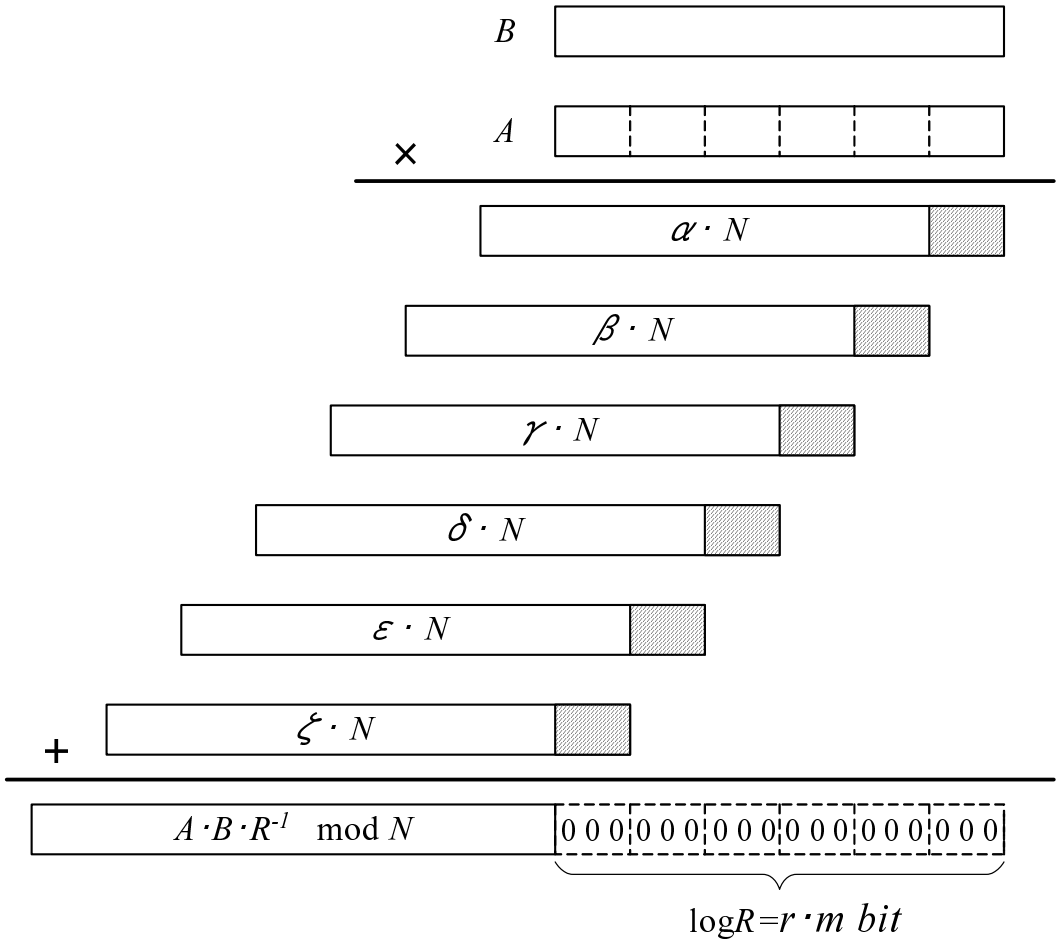

図 4.3: Step 全体簡略図

4.2 ブロック図

前節のアルゴリズムからブロック図を作成した．以下に示しているのは，$N < 2^{512}$，$r = 16$ のブロック図である．

図 4.4 に示す乗算ブロックは，アルゴリズムの Step2 と Step3 で行う，$A_i \cdot B$ と $Q \cdot N$ の乗算を行うブロックである．このような $16bit \times 512bit$ の乗算はこの 2 つだけである．また，$Q \cdot N$ の計算されるタイミングは，$A_i \cdot B$ と V との演算結果である Q を用いた計算であるから，$A_i \cdot B$ とは違うタイミングで行うことができるので，一つの乗算器で実現できる．

またここで，Q' となっているのは，次のページにある図 4.5 で入力と出力の値が違うことを意味している．

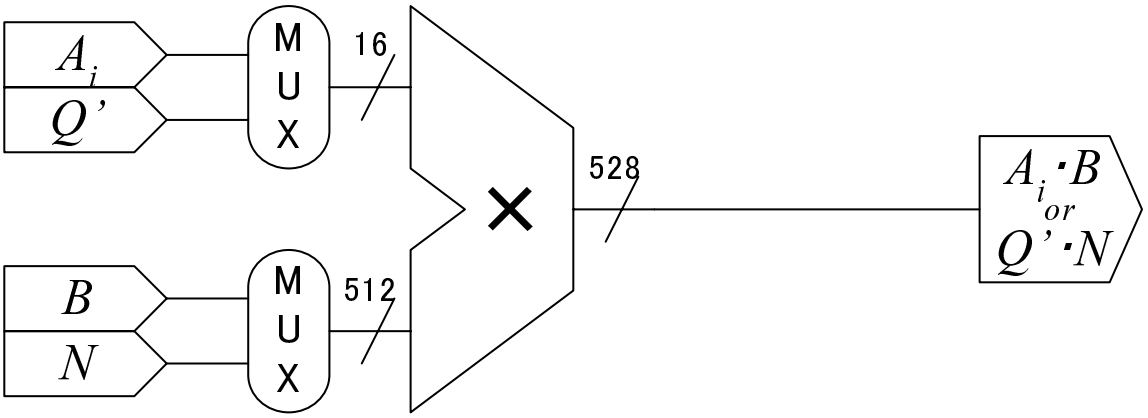

図 4.4: 乗算ブロック

一方, 図 4.5 に示す加算ブロックでは, アルゴリズムの Step2 と Step3 で行う, $M+A_i \cdot B_0$ と $M + A_i \cdot B + Q \cdot N$ の加算の部分をおこなう.

ここで, 加算器の出力である, M の下位 r(ここでは 16)bit を取り出し, V と乗算しているのは, この場合加算器の bit 数がこの乗算器の bit 数よりも大きく, 下位 $r$bit の加算が終わってから, 全ての桁の加算が終了する前に, 乗算器に入力値を与えて高速化を図るためである. また, 下の図を見て分かるように, 乗算器の出力は 2^r を法とする剰余演算であるから, 下位 $r$bit に削られる.

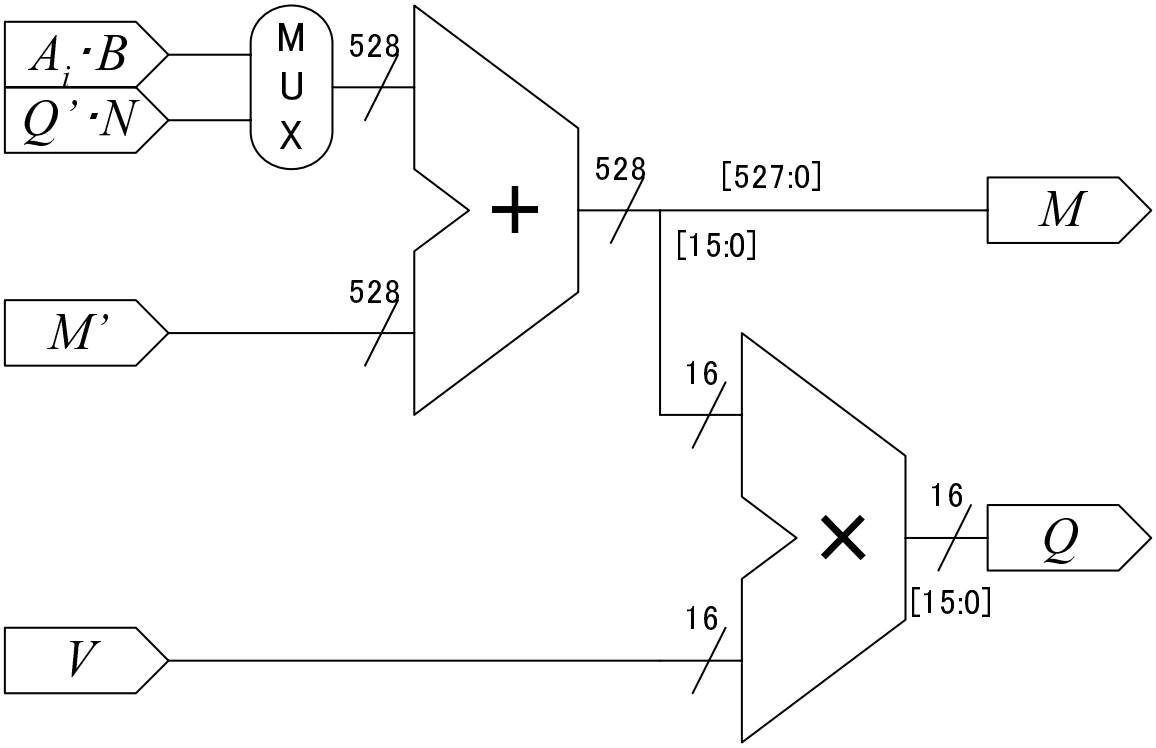

図 4.5: 加算ブロック

第5章　実装

5.1　ハードウエア実装

この章では, モンゴメリ乗算アルゴリズムのハードウェア実装について述べる.

5.1.1　実装の流れ

今回ハードウェアとして扱うものは, FPGA とよばれる回路構成が書き換え可能な VLSI である. LSI や IC などのハードウェアと違うのは, ハードウェア記述言語を用いて記述する点である. 実装の流れは図 5.1 に示した. 今回使用した FPGA は, Xilinx 社製 XC2V3000 で, 3000 万ゲート相当のもので, 高速な 18bit 乗算器を 96 個内蔵している. そして, ハードウェア記述言語である VerilogHDL を用いて, 記述した. 記述には, RTL(回路の構成要素であるレジスタやカウンタなどと, それらの間でのデータの転送状態（接続状態）をクロックの概念をいれて表現しているレベル) を使用するのが一般的で, 詳細なブロック図をもとにそのまま記述できる.

そして, 出来上がった VerilogHDL ソースを Xilinx 社製の ISE ロジックデザインツールという論理合成ツールを用いて, FPGA に直接ダウンロードできる形であるネットリストという形コンパイルする.

今回ハードウェア化を行う場合に使用したのは以下のような環境である.

使用言語 VerilogHDL

ハードウェア（FPGA） Xilinx 社 XC2V3000(高速な 18bit 乗算器内蔵)

タイミング解析 Xilinx 社 ISE ロジックデザインツール

5.1.2　パラメータ

最近の RSA 暗号では安全性の問題で 1024bit 以上が必要といわれている. そこで, 今回の実装は, 入力に 1024bit の N, A, B, を用いた. また, 今回使用する FPGA が高速な 18bit

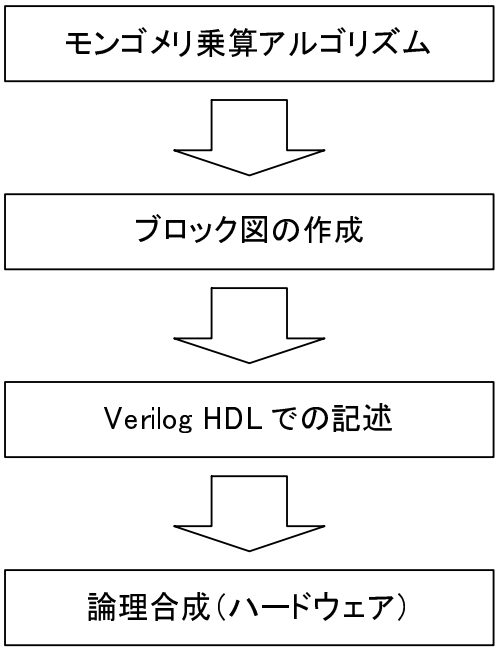

図 5.1: ハードウェア実装の流れ

乗算器を 96 個搭載しているので, 高速に動作させかつ, 内部セルの無駄使いを防ぐために, r を 16bit とした.

5.2 ハードウェアによる結果

今回は結果として, Xilinx 社 ISE ロジックデザインツールが吐き出す, 回路のいろいろな数値を用いた. [Device utilization summary] は FPGA 内のロジックセルの使用量を, [Timing Summary] は, どれくらいのクロック速度で実行可能であるかを表わしている.

また, 図 5.2 は, ModelSim というソフトで出力した波形である.

```
[Device utilization summary]
---------------------------
 Number of Slices:                     4154  out of  14336    28%
 Number of Slice Flip Flops:           4367  out of  28672    15%
 Number of 4 input LUTs:               7534  out of  28672    26%
 Number of bonded IOBs:                 151  out of    484    31%
 Number of MULT18X18s:                   62  out of     96    64%
 Number of GCLKs:                         1  out of     16     6%

[Timing Summary]
---------------
   Minimum period: 68.743ns (Maximum Frequency: 14.547MHz)
   Minimum input arrival time before clock: 5.194ns
   Maximum output required time after clock: 8.248ns
```

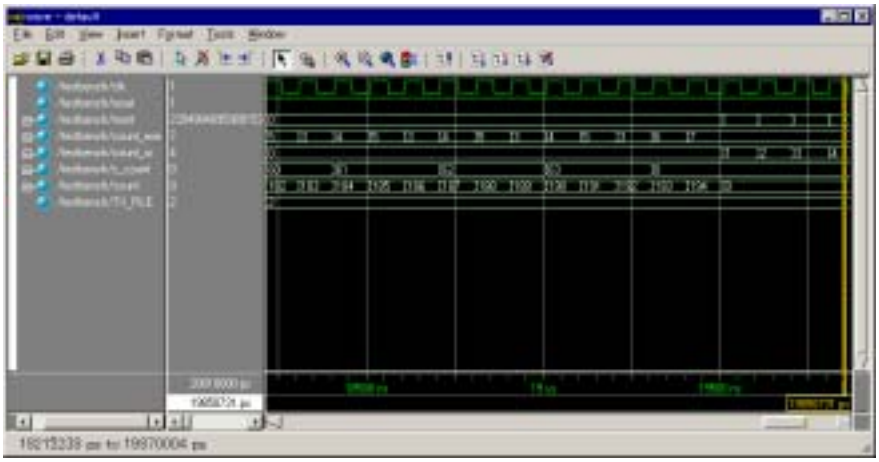

図 5.2: ModelSim による波形

結果として, 96 個の乗算器のうち 62 個を使用し, また 300 万ゲート中の 30%くらいの使用量で FPGA に実装することができた. またクロックサイクル数は 194 であった. 今回は, ハードウェアとしてのモンゴメリ乗算アルゴリズムの実行時間を, 最大遅延時間 × クロックサイクル数として算出した.

$$194(\text{クロックサイクル}) \times 68.743(ns) = 13.336(us)$$

5.3 ソフウェアによる実装

この節ではハードウェア実装との比較のために行った, ソフトウェアによる実装について述べる. プログラムは, 多倍長演算も簡単に実装可能なLEDAというC++のソフトウェアクラスライブラリを用いた. 以下にそのプログラムを示す.

```
 0:  #include <LEDA/integer.h>
 1:  main(int argc,char* argv[])
 2:  {
 3:       integer a, b, n;
 4:       integer x;
 5:       int C = atoi(argv[1]);
 6:
 7:       float TIME = 0.0;
 8:       for (int i=1; i<C; i++) {
 9:            a = integer::random(1024);
10:            b = integer::random(1024);
11:            n = integer::random(1024);
12:
13:            float T = used_time();
14:            x = (a * b) % n;
15:            TIME += used_time(T);
16:       }
17:       cout << TIME << endl;
18:  }
```

integer 型の宣言で多倍長の計算を可能にして (3,4 行目), 1024 ビットのランダムな整数 a, b, n を生成し (9,10,11 行目), $AB \bmod N$ を C 回計算し (14 行目) その時間を計測する (13,15 行目).

このプログラムを用いて, $A \cdot B \bmod N$ の計算時間を 3 台の異なる環境のパソコンで測定した.

5.4 ソフトウェアによる結果

結果は以下のようである. 時間は 10 万回の計算を行い, 1 回あたりの時間に直したものである.

	Sun Solaris5	linux1	linux2
time(us)	275.7	109.7	54.7
cpu	UltraSPARC-IIi 270MHz	Intel Xeon 1.7GHz	Intel Xeon 2.8GHz
memory	256MB	1.5GB	2GB

第6章　評価と考察

今回, モンゴメリ乗算回路をハードウェアとソフトウェア両方により実装してきた. その結果, 本研究で実装したハードウェアは, RSA 暗号で実用的であるとされている 1024bit の剰余演算を, 13.336(us) で実行できた. それは, 1 秒間では, 74985 回の処理が可能ということになる. また, ソフトウェアとくらべて, 4.1 倍から 20.7 倍に高速化することができた.

高速化については, 動作周波数が１４ MH ｚと低いため, 最適化されたとはいえない結果となってしまった. 今よりも高速化するためには, データの流れを整理してパイプライン化するべきであろう. その一つの方法として, 論文 [10] のように, 中国剰余定理を用いる方法などがあげられる. また加算回路を高速化ができると知られている, キャリースキップや Wallace ツリー回路にすることでさらに高速化が望めるだろう.

回路規模については約 100 万ゲートくらいの回路であり, スマートカードのようなローエンドマシンには載せることができないが, 当初の目的の一つである鍵長が倍になった場合に, 回路を拡大してもこの FPGA には収まる大きさである. しかし, 今回は r を 16bit としたが, これが 32bit となることでも回路は倍増してしまうので, 鍵長を伸ばした場合には $r$bit とのトレードオフは必要であるだろう.

しかし, まだまだ高速化が望めそうなことからも, 当初の目的である暗号化用のハードウェアとして FPGA を用いることは, 有効な手段であるといえる.

第7章　まとめ

本論文では, 近年欠くことができない技術の一つである公開鍵暗号システムにおいて, 繰り返し行われる剰余演算に注目した. その演算を高速に計算するために, 剰余演算の一つのアルゴリズムであるモンゴメリ乗算を取り上げた. そしてそのアルゴリズムをハードウェア用に組み立て, ハードウェア記述言語 VerilogHDL により記述した. また, ハードウェアには, 18 ビットの高速な乗算器を搭載した FPGA, XC2V3000 を使用した.

RSA 暗号で実用的であるとされている 1024bit の剰余演算を回路化し, 13.336(us) で動作させることができた. また, 比較用に測定したソフトウェアによる実行時間によると, 本研究で作成したハードウェアは. ソフトウェアよりも 4.1 倍から 20.7 倍高速に動作した. 今回作ったハードウェアも, まだ高速化が期待できることから, 鍵長や暗号システム自体の変化に対応できるような暗号化用チップとして, ＦＰＧＡを用いることは有効であるといえる.

謝辞

最後になりましたが, 本研究を行うにあたり, 中野浩嗣助教授には, 種々の親切なる御指導, 御助言をたまわり, 深く謝意を表する次第であります. また, 当講座の浅野哲夫教授をはじめ, 助手の小保方幸次氏や元木光雄氏をはじめ, 同研究室の皆様にも, 種々の親切なる御助言を頂き, 深く感謝いたします.

参考文献

[1] R.L. Rivest, A. Shamir, and L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Comm. ACM, vol. 21, no. 2, pp. 120-126, 1978.

[2] P. L. Montgomery, Modular Multiprication without Trial Division, Mathematics of Computation, vol. 44, no.170, pp.519-521, Apr. 1985.

[3] E.F. Brickell, A Survey of Hardware Implementations of RSA, Advances in Cryptology, Advances in Cryptology, -CRYPTO '89, G. Brassard, ed., pp. 368-370. Spring, 1990.

[4] S.E. Eldridge, A faster modular multiplication algorithm, Intern. J. Computation. Math, vol. 40, pp. 63-68, 1991.

[5] C. D. Walter, Faster modular multiplication by operand scaling, Advances in Cryptology, -CRYPTO '91, vol. 576, J. Feigenbaum, ed., pp. 313-323, 1992.

[6] S.E. Eldridge and C.D. Walter, Hardware implementation of Montgomery's modular multiplication algorithm, IEEE Transactions on Computers, vol. 42, no. 6, pp. 693-699, 1993.

[7] N. Takagi, Modular Multiplication Algorithm with Triangle Addition, Proc. 11th IEEE Symp. Computer Arithmetic, M.J. Irwin, E. Swartzlander, and G. Jullien, eds., pp. 272-276, 1993.

[8] M. Shand and J. Vuillemin, Fast Implementations of RSA Cryptography, Proc. 11th IEEE Symp. Computer Arithmetic, M.J. Irwin, E. Swartzlander, and G. Jullien, eds., pp. 252-259, 1993.

[9] C. K. Koc, T. Acar, and B. Kaliski, Analyzing and comparing Montgomery multiplication algorithms IEEE Micro, vol. 16, pp. 26-33, June 1996.

[10] 新保 淳, 野崎 華恵, 川村 信一, 高速 RSA 暗号 LSI 東芝レビュー, vol.56, no.7, pp. 10-13, 2001.

[11] 佐藤 証, 高野 光司, 高速モンゴメリ演算回路The 2001 symposium on Cryptography and Information Security Oiso, Japan, January. 2001.