親子剰余乗算器を用いた
左向きアレイ法の実装方式

櫻井 隆雄

1　はじめに

1.1　背景

近年、通信ネットワークを介しての重要な情報の流通が増大している。それに伴い外部への情報の流出、悪意ある第三者による改竄などの新たな種類の犯罪も増加している。そのため、それらの犯罪を未然に防ぎ、重要な情報を守るための暗号化技術の重要性はより高まっている。

その中で RSA 暗号 [1][2] に代表される公開鍵暗号は電子商取引や電子署名などに用いられる重要な暗号化技術のひとつである。公開鍵暗号は DES[4] や AES[5] に代表される秘密鍵暗号に比べて鍵の管理が容易で認証にも用いることが可能という利点がある一方で、処理に必要とする計算量が大きいという欠点がある。例えば RSA 暗号の場合は現在において主に用いられている鍵長 1024 ビットでの暗号化を行う場合、1024 ビットの数同士の剰余乗算を 1000 回以上行う必要がある。そのため公開鍵暗号を用いる際には専用のハードウェアを用いることが多く、その高速化手法が長年研究されている。

本研究は特に RSA 暗号に着目し、その専用ハードウェアの高速な実装手法を提案し、評価を行う。

1.2　研究の目的

RSA 暗号の演算は多ビットの剰余乗算とべき乗演算によって成り立っているが、現在用いられている手法でハイエンド向けの構成を考えた場合、鍵長を大きくすることによってべき乗演算回路の面積が急激に増大する。その一方で、剰余乗算回路は鍵長を大きくすることにより、ある程度回路面積は大きくなるがその増加率はべき乗演算回路と比べて小さな物である。

本研究は現在主流の方式には速度面でやや劣るが、並列性の高いべき乗演算方式を用いて、複数の剰余乗算器を並列に動作させることによって高速にべき乗演算を行うことのできる回路を実現することを目的とする。

1.3　論文の構成

本論文の構成は以下の通りである。2 章で公開鍵暗号と RSA 暗号について述べる。また、3 章で RSA 暗号をハードウェア実装する際によく用いられるアルゴリズムについて述べる。そして 4 章でそれらのアルゴリズムをハードウェア記述言語で記述して論理合成を行いその回路面積と処理速度を評価することで問題点を指摘する。それを受けて 5 章でその問題点を解決するための手法を提案し、6 章で提案手法の評価を行う。最後に 7 章でまとめる。

2　公開鍵暗号と RSA

2.1　公開鍵暗号

公開鍵暗号方式という概念は 1976 年に Whitfield Diffie と Martin E. Hellman によって考案された。2 人が示した公開鍵暗号の要求は以下のようなものである。

- あらかじめ 2 つの鍵を用意しておく
- 片方の鍵は他者に広く公開するため公開鍵と呼ぶ
- もう 1 つの鍵は厳重に秘匿されるため秘密鍵と呼ぶ
- 秘密鍵で暗号化されたデータは対応する公開鍵でしか復号できない
- 公開鍵で暗号化されたデータは対応する秘密鍵でしか復号できない
- 秘密鍵は公開鍵から類推することが不可能であるか、極めて困難である

それまでにも使用されていた共通鍵暗号方式は暗号化を行う通信に関わるすべての者が他者には秘密にしなければならない秘密鍵を共有する必要があったため、その秘密鍵の交換方法が問題となっていたが、公開鍵暗号方式では秘密鍵のみを秘匿すれば良いため、鍵の輸送方法を考慮する必要が無いという利点がある。

この公開鍵暗号の概念を受けて 1977 年に Rivest、Shamir、Adleman の 3 人が実際にその要求を満たす暗号方式を考案し、その方式は 3 人の名前の頭文字を取って RSA 暗号と名付けられた。その後に楕円曲線暗号 [3]、ElGamal 暗号などが考案され現在に至るまで使用されている。

2.2 RSA 暗号

公開鍵暗号として最も有名なものが Rivest、Shamir、Adleman の 3 人が考案した RSA 暗号 [1][2] である。これは非常に大きい数の素因数分解が極めて困難であることを利用したものである。その暗号化手順は以下の通りである。

まず受信者はある大きな素数 p と q を掛け合わせた数 N と、N と互いに素である数 e を公開鍵として送信者に送る。このとき素数 p と q は厳重に秘匿する。送信者は暗号化をおこなう平文を決められた法則（これは秘匿する必要は無い) により文字列から数字列に変換する。そしてその数字列をそれぞれが N よりも小さくなるように分割し、それぞれ M_0,M_1,M_2… とする。そのようにして出来た M_i を与えられた N と e を用いて以下の式で暗号文 c_i に変換して受信者に送る。

$$c_i = M_i^e \bmod N$$

c_i を受け取った受信者は復号化を行う必要があるが、復号に必要な秘密鍵 d を次のようにして生成する。

$$d = e^{-1} \bmod (p-1)(q-1)$$

生成された d を用いて以下の変換を行う。

$$M_i' = c_i^d \bmod N$$

このとき、

$$M_i' = M_i$$

となる。このようにして得られた数字列を送信者が行った法則と同じ物で文字列に戻せば平文が得られる。

M' が M と等しくなることは以下のように証明される。この証明にはオイラーの公式

$$a^{\varphi(N)} = 1 \bmod N$$

を用いる。$\varphi(N)$ はオイラー数と呼ぶ。これは N よりも小さい自然数の中で N と互いに素な数の個数で、RSA 暗号に用いる N の場合は

$$\varphi(N) = (p-1)(q-1)$$

となる。d の算出方法から

$$de = 1 \bmod (p-1)(q-1)$$

ということが分かり、

$$de = k(p-1)(q-1) + 1$$

となる (k は整数)。また、

$$\begin{aligned} M' &= c^d \bmod N \\ &= (M^e)^d \bmod N \\ &= M^{de} \bmod N \end{aligned}$$

であるから

$$\begin{aligned} M' &= M^{k(p-1)(q-1)+1} \bmod N \\ &= M^{k\varphi(N)+1} \bmod N \\ &= 1^k \cdot M \bmod N \\ &= M \end{aligned}$$

となり、M' が M と等しくなることが示された。

3 RSA 暗号の演算方式

3.1 RSA 暗号に必要な演算

RSA 暗号における公開鍵の 1 つである N のビット数 n はセキュリティパラメターと呼ばれ、RSA 暗号の強度と密接な関係がある。この n は RSA 暗号が考案された当時は 120 程度であったが、計算機性能の向上により現在において主に使われている値は 1024 となっており、既に 2048 を用いる製品もでてきている。ここで問題となるのが n に伴い増大する暗号化及び復号化処理の計算量である。RSA の暗号化処理は前述の通り、

$$c = M^e \bmod N$$

という演算である。この時に暗号化鍵（復号化鍵）は n ビットまでの値を取りうるため、非常に高い次数でのべき乗演算を必要とする。そして、べき乗演算を行う際の一回の乗算は $A \cdot B \bmod N$ のような形の剰余乗算となる。この剰余乗算は数千ビットで行うと膨大な計算コストがかかる。よって、RSA 暗号の処理を高速に実行するハードウェアの製作にはべき乗演算と剰余乗算を効率的に行うためのアルゴリズムが不可欠なものとなる。

次節からこれまでに考案されているアルゴリズムを紹介する。

3.2 剰余乗算の演算方式

3.2.1 モンゴメリ乗算

前述したように $A \cdot B \bmod N$ のような演算を剰余乗算と呼ぶ。剰余乗算において剰余は一般的に 1 ビットずつの減算の繰り返しにより求めることができる。しかし、この方法では演算に膨大な時間がかかるため、より良いアルゴリズムを用いる必要がある。一般に 1 回の除算の計算コストよりも 2、3 回の乗算、加算の計算コストの方がはるかに小さい。この事を利用したものがモンゴメリ乗算アルゴリズム (Montgomery's multiplication algorithm)[6][13] である。モンゴメリ乗算アルゴリズムは、剰余の性質を非常にうまく利用することで、本来 1 回の乗算と n 回の減算が必要である剰余乗算を、2 回の乗算と 1 回の加算、そして 1 回の減算で行うことができる演算方式である。

Input :	$Y = A \cdot B \cdot R^2 \bmod N$, N
Pre-calculation :	$R = 2^n (2^n > N > 2^{n-1})$, $V = -N^{-1} \bmod R$
Output :	$Y' = Y \cdot R^{-1} \bmod N$
Step1:	$U = Y \cdot V \bmod R$
Step2:	$W = Y + N \cdot U$
Step3:	$M = W/R$
Step4:	if$(M > N)$ $Y' = M - N$ else output $Y' = M$

図 1: モンゴメリ乗算アルゴリズム

図 1 にモンゴメリ乗算の演算手順を示す。モンゴメリ乗算は入力を A, B、法を N としたときに $A \cdot B \cdot R^{-1} \bmod N$ を求めるものである。これは $A \cdot B$ を R で割って N で剰余を求めることである。しかし、必ずしも $A \cdot B$ は R で割り切れるとは限らない。そこで剰余の性質に注目する。$A \cdot B$ は R で割り切れなくても、$0 < U < R$ となる整数 U を定義したとき、$A \cdot B + U \cdot N \bmod R = 0$ となるような U は必ず存在する。この U の値は $U = A \cdot B \cdot V \bmod R$

$(V = -N^{-1} \bmod R)$ という演算で与えられる。しかも、$A \cdot B \bmod N$ と $A \cdot B + U \cdot N \bmod N$ は同値である。ここで、$U < R$ であり、$A \cdot B < N^2 < N \cdot R$ となるような前提条件から、

$$(A \cdot B + U \cdot N)/R < (N \cdot R + N \cdot R)/R = 2N$$

となる。従って $(A \cdot B + U \cdot N)/R$ の計算を行った後で 0 回もしくは 1 回の減算を行えば求める $A \cdot B \cdot R^{-1} \bmod N$ の値を得ることができる。これらの手順を図で表すと図 2 のようになる。

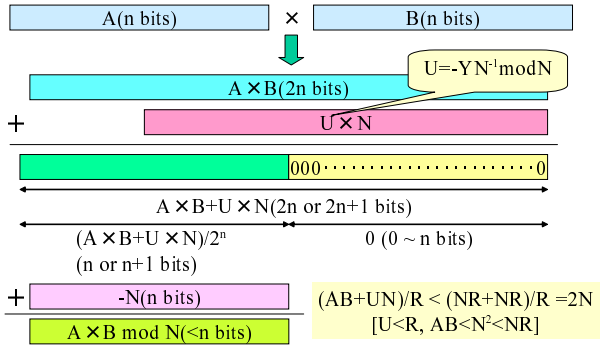


図 2: モンゴメリ乗算

モンゴメリ乗算アルゴリズムでは、$A \cdot B \bmod N$ の演算を行う際にまず N のビット数 n に対して 2^n となるモンゴメリ定数 R を定め、A、B いずれかに R^2、もしくは双方に R を掛けて、いずれも N で剰余をとった値を求める。得られた 2 数を掛け合わせて N で剰余を取った数を Y とし、モンゴメリ乗算と呼ばれる $Y \cdot R^{-1} \bmod N$ を求める計算を行う。これにより得られた値は $A \cdot B \cdot R \bmod N$ と同値である。この値を Y' とし、この値で再びモンゴメリ乗算 $Y'R^{-1} \bmod N$ を行えば、最終的な目的である $A \cdot B \bmod N$ の値が得られる。

3.2.2 部分積和算を用いたモンゴメリ乗算

RSA 暗号では数千ビットの数に対して演算を行う。その際に数千ビット入力の乗算器を直接作る事は非常に困難であるため、より小ビットの乗算器を複数用いる、もしくは繰り返し用いる事により数千ビットの乗算を実現する。その方式には数を適当な大きさに分割して部分積和算を用いる方式 [12] や RNS 表現を用いる方式 [7][8] などがある。本研究では部分積和算を用いる方式を使用するため、この方式について詳しく述べる。

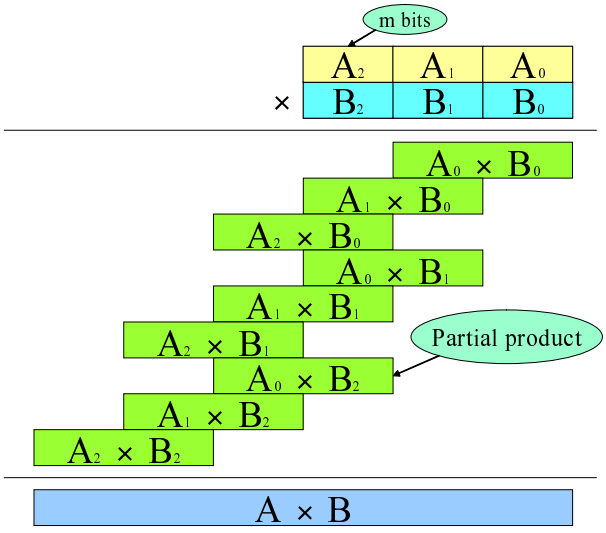


図 3: 部分積和算

部分積和算による乗算は図 3 のように入力された数を m ビットのブロックに分割する。これを 3.2.1 節で述べたモンゴメリ乗算アルゴリズムに適用したものが文献に述べられているアルゴリズムである。その内容を図 4 に示す。

図 4 にあげたアルゴリズムは A を m ビットのブロックに分割し、小さい桁のブロックから順番に B との間でモンゴメリ乗算を行う、というものである。図 1 の Step1 にあたるのが図 4 の Step2 であり、ここで $M + a_i B$ を 2^m で割り切る事のできるようにするためには N を何倍すればいいのかという数字を求める。そうして求められた Q を N と掛け合わせて、$M + a_i B$ と足し合わせる。このとき、下位の m ビットの値はすべて 0 であるので、剰余値を保存しながら m ビット右シフトすることができる。この手順を l 回繰り返す事によって n ビット同士のモンゴメリ乗算を効率的に行える。図 5 に A を 4 分割した場合の全体の計算の簡略図を示す。

Input :	$A = (a_{l-1} \cdots a_0)_{2^m}$,
	$B = (b_{l-1} \cdots b_0)_{2^m}$,
	N
Pre-calculation :	$R = 2^n > N > 2^{n-1}$,
	$V = -N^{-1} \bmod 2^m$
Output :	$Y' = A \cdot B \cdot R^{-1} \bmod N$
Step1:	$M = 0$, $i = 0$
Step2:	$Q = (M + a_i \cdot b_0) \cdot V \bmod 2^m$
Step3:	$M = (M + a_i \cdot B + Q \cdot N)/2^m$,
	$i = i + 1$
Step4:	if $(i < l)$ return Step2
Step5:	if $(M > N)$ $Y' = M - N$
	else $Y' = M$

図 4: 部分積和算を用いたモンゴメリ乗算アルゴリズム

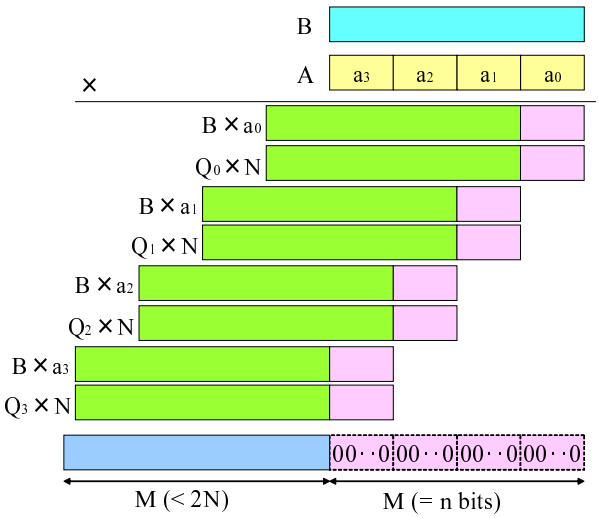


図 5: 4 分割した場合の剰余乗算

この m を小さくするほど必要な乗算器のサイズが小さくなり、延滞の回路面積が小さくなり、実装も容易となる。しかし、m を小さくすることで、1 回の剰余乗算において乗算器を用いる回数が増え、その結果、処理速度が低下する。そのため、求められる処理速度や回路規模によって適切な m を選択する必要がある。

3.3 べき乗演算のアルゴリズム

本節ではべき乗演算のアルゴリズムについて述べる。以降、$A \cdot B \bmod N$ の演算を $A \circ B$ と表す。

3.3.1 右向きバイナリ法

X^e を求める場合、単純に演算を行うのならば X を e 回掛ければよいが、X、e が数千ビットである場合、膨大な計算コストがかかる。しかし、そこで指数 e を $e = 2e_1 + e'_1$ (e'_1 は 0 または 1) と分割すると、X^e を

$$X^e = (X^{e_1})^2 \cdot X^{e'_1}$$

と表すことができる。子の式の右辺に従えば、X を e 回掛け合わせることなく e_1 回 X をかけることにより X^{e_1} を求め、それを自乗し、必要であるのならばさらに X をかけることによって X^e を求めることができる。これは、e_1 をさらに $e_1 = 2e_2 + e'_2$ (e'_2 は 0 または 1) と分割することで、より少ない乗算回数で X^e を求めることが可能となる。

Input:	$N, 0 < X < N, e$
Output:	$X^e \bmod N$
Step1:	$A = X$
Step2:	$i = \lceil log_2 e \rceil$
Step3:	$A = A \circ A$, $i = i - 1$
Step4:	if $(e_i = 1)$ $A = A \circ X$
Step5:	if $(i \neq 0)$ return Step3
	else output A

図 6: 右向きバイナリ法

このことを利用したものが図 6 に示す右向きバイナリ法 (Left-to-Right binary method) と呼ばれるべき乗演算方式である。右向きバイナリ法は $X^e \bmod N$ を求めるにあたり、まず計算結果を格納する変数 A に X を代入する。そして e を 2 進数で表現して各桁の値を上位から 1 桁ずつ読んでいき、その値に関係無くまず A に $A \circ A$ の演算で得られる値を代入し、読み取った e の桁の値が 0 であった場合はそのまま次の桁を読み、値が 1 であった場合は A に $A \circ X$ と

なる値を代入して次の桁を読む。この作業を e の最小桁まで繰り返し行う。

この方法による $X^e \bmod N$ を計算するときの乗算回数は e を 2 進数表記したときの桁数とそのときに現れる 1 の数の合計である。

3.3.2 左向きバイナリ法

右向きバイナリ法の右向きとは指数 e の上の桁から下の桁、つまり左の桁から右の桁の方に向かって処理を行うことに由来する。これとは逆に舌の桁から上の桁、つまり右の桁から左の桁に向かって処理する方式もあり、こちらは左向きバイナリ法 (Right-to-Left binary method) と呼ばれる。図 7 に計算方式を示す。

Input:	$N, 0 < X < N, e$
Output:	$X^e \bmod N$
Step1:	$A = X$
Step2:	$i = \lceil log_2 e \rceil$, $j = 0$
Step3:	if $(e_0 = 1)$ $B = X$
	else $B = identityelement$
Step4:	$A = A \circ A$, $j = j + 1$
Step5:	if $(e_j = 1)$ $B = B \circ A$
Step6:	if $(i \neq j)$ return Step4
	else output B

図 7: 左向きバイナリ法

左向きバイナリ法は A と B の 2 つの変数を用意しておく。計算の開始時に A には X を入れ、B には e の最小桁が 1 ならば X、0 ならば $identityelement$ を入れる。そして、e の 1 つ上の桁を読み、その値に関係無く A に $A \circ A$ を入れた後に、そのときの e の桁の値が 0 であるときはそのまま何もせずに 1 つ上の桁を読み、値が 1 のときは B に $B \circ A$ の結果を入れ、1 つ上の桁に進む。この作業を e の最大ビットまで繰り返し行うことで X^e を求めることができる。。

この方法による $X^e \bmod N$ を計算するときの乗算回数は右向きバイナリ法と同様に e を 2 進数表記したときの桁数とそのときに現れる 1 の数の合計である。

3.3.3 右向きアレイ法

Input:	$N, 0 < X < N,$
	$e = (e_{t-1} \cdots e_0)_{2^k}$
Output:	$X^e \bmod N$
Step1:	$X_0 = identityelement, X_1 = X$
Step2:	for i in 2 to 2^{k-1} , $X_i = X^i \bmod N$
Step3:	$A = X_{e_{t-1}}$, $i = t - 1$
Step4:	$A = A \circ A$ for k times, $i = i - 1$
Step5:	$A = A \circ X_{e_i}$
Step6:	if $(i \neq 0)$ return Step4
	else output A

図 8: 右向きアレイ法

前に述べた右向きバイナリ法からべき乗演算に必要となる乗算回数をさらに削減するために考案された手法が図 8 に示す右向きアレイ法 (Left-to-Right ary method)[9][12] である。このべき乗演算方式は、右向きバイナリ法では 1 桁に 1 回必要であった $A \circ X^{e_i}$ の演算を e の桁を読んだときに A と掛け合わせる値を X の他にあらかじめ X^2,$X^3 \cdots X^{2^k-1}$ と 2^k 個用意しておき、k 桁分まとめて行うことにより、k 桁に 1 回にまで削減する方式である。具体的な演算手順としては、まず e を k 桁ずつ l 個に分割し、それぞれ $e_{l-1}, e_{l-2}, \cdots, e_0$ とする。次に A に $X^{e_{l-1}}$ を入れる。そして、A に $A \circ A$ を入れる演算を k 回行い、その後に A に $A \circ X^{e_{l-2}}$ を入れる。そして、A に $A \circ A$ を入れる演算を k 回行い、その後に A に $A \circ X^{e_{l-3}}$ を入れる。このような手順を $l - 1$ 回繰り返すと X^e を求めることができる。このとき、k の値を大きくするほど $A \circ X^{e_i}$ という演算を行う回数は減少する。しかし、あらかじめ用意しなければならない X,X^2,$X^3 \cdots X^{2^k-1}$ のために $2^k - 2$ 回の前処理演算を必要とするため、あまり k を大きくするとかえって全体の乗算回数は増加することとなる。このアレイ法のパラメター k が 1 の場合のことをバイナリ法と考えることもできる。

3.3.4 左向きアレイ法

アレイ法にもバイナリ法同様に左向き方式が存在する。それが左向きアレイ法 (Right-to-Left ary method) と呼ばれる方式 [10] である。図 9 に計算方式を示す。

Input:	$N, 0 < X < N,$
	$e = (e_{t-1} \cdots e_0)_{2^k}$
Output:	$X^e \bmod N$
Step1:	for i in 0 to 2^{k-1} ,
	$B_i = identityelement$
Step2:	$A = X$, $B_{e_0} = X$, $i = 0$
Step3:	$A = A \circ A$ for k times, $i = i + 1$
Step4:	$B_{e_i} = B_{e_i} \circ A$
Step5:	if $(i < t - 1)$ return Step3
	else go to Step6
Step6:	for j in 2^{k-1} to 2 ,
	$B_{j-1} = B_j \circ B_{j-1}$, $B_1 = B_j \circ B_1$
Step7:	output B_1

図 9: 左向きアレイ法

左向きアレイ法における自乗演算の結果を格納する変数 A の扱いは左向きバイナリ法と同様であるが、e^i の値により A と乗算を行う変数 B を $B_0, B_1, B_2, \cdots, B_{2^k-1}$ と 2^k 個用意しておく。そして右向きアレイ法同様に e を k 桁のブロックに分割する。以上の前準備の後にまず A には X を入れ、全ての B には $identityelement$ を入れる。そして下の桁のブロックから値を読んでいき、A の自乗演算を k 回行うごとに 1 回、そのときの A の値と B_{e_i} の間で乗算を行う。それを一番上の桁のブロックまで繰り返し、得られた B で

$$\prod B_d^d (d = 1, \cdots, 2^k - 1)$$

のような演算を行った結果が X^e の値となる。このとき、図 9 の STEP6,7 にあげた手順 [11] を用いれば、この後処理演算は $2(2^k - 1)$ 回で行うことができる。

3.3.5 各べき乗演算の性能比較

RSA 暗号において $n = 1024$ としたとき、暗号化、復号化の指数 e、d は 1024 ビットまでの値を取り得る。実際に 1024 ビットの演算を行う際に、各べき乗演算方式により剰余乗算の回数がどのように変化するのかを示したのが図 10 である。この場合はワーストケースとして全てのビットを 1 としたとき、k の値を 1 から 8 まで変化させたときの乗算回数を示した。前述の通りアレイ法の $k = 1$ はバイナリ法とみなすことができるのでグラフの本数は 2 本である。

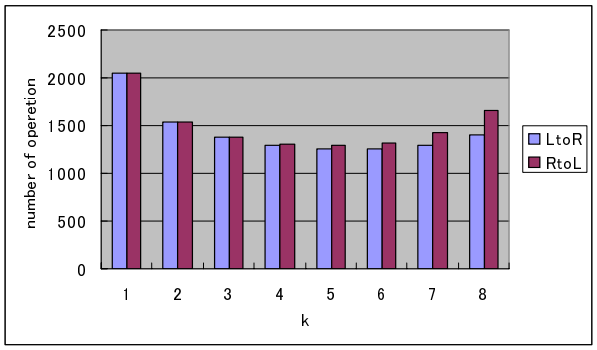


図 10: k の値と乗算回数の比較

全体的に右向き法に比べて左向き法の方が剰余乗算の回数が多い。これは、左向き法に必要な後処理演算の回数が右向き法に必要な前処理演算の回数の 2 倍となっているためである。

右向き法、左向き法共に k が 4 から 5 あたりまでは k が大きくなるにつれて演算回数は減少している。しかし、それ以降は逆に演算回数が増加する傾向にある。この理由は k の値を大きく取りすぎると前処理演算や後処理演算の剰余乗算の回数が大きくなり、その増加量が削減効果よりも大きくなってしまったからである。

また、ハードウェア実装を行う際には k の増加に伴い X をべき乗を格納するために必要なレジスタの数も増加し、その結果回路規模も大きくなる事から、$k = 3$ 辺りが実装を行う際には最適と考えられている [12]。

4 予備評価

4.1 評価した回路の構成

4.1.1 全体の構成

この節では評価に使用する回路において 3 章で述べた各演算の回路をどのように組み合わせるかについて述べる。

RSA 暗号回路は 3.1 節で剰余乗算部分とべき乗演算部分に分けられることを述べたが、剰余乗算部分には 3.2 節で述べたモンゴメリ乗算を用いる。

一般に n ビットの数同士のモンゴメリ乗算を行う場合、3.2.2 節で述べたように片方の数を m ビットのブロックに分割して $n \cdot m$ のモンゴメリ乗算を行う。そして $n \cdot m$ のモンゴメリ乗算を n/m 回繰り返すことにより 1 回の $n \cdot n$ の剰余乗算が行われる。そのため、本研究では $n \cdot m$ のモンゴメリ乗算を行う部分と $n \cdot n$ の剰余乗算を行う部分を階層構造とする実装を行った。

一方でべき乗演算部分は剰余乗算回路を繰り返し用いることにより演算を行うため、$n \cdot n$ の剰余乗算を行う部分のさらに上の階層におかれることとなる。よって全体の構成はまず RSA 暗号回路という一番上の層があり、そこから N から V を求めるような周辺回路を除いたすぐ下の層にべき乗演算回路がおかれる。べき乗演算回路は k の値に応じた数の n ビットのレジスタといくつかのセレクタ、そして剰余乗算回路によって構成される。そして剰余乗算回路はセレクタと減算器、$n \cdot m$ のモンゴメリ乗算回路によって構成される。そのモンゴメリ乗算回路は m ビットの乗算器や加算器、シフトレジスタなどによって構成される。これらの構成を図で表すと図 11 のようになる。

各々の演算器部分に関する詳しい構成を次節以降で述べる。

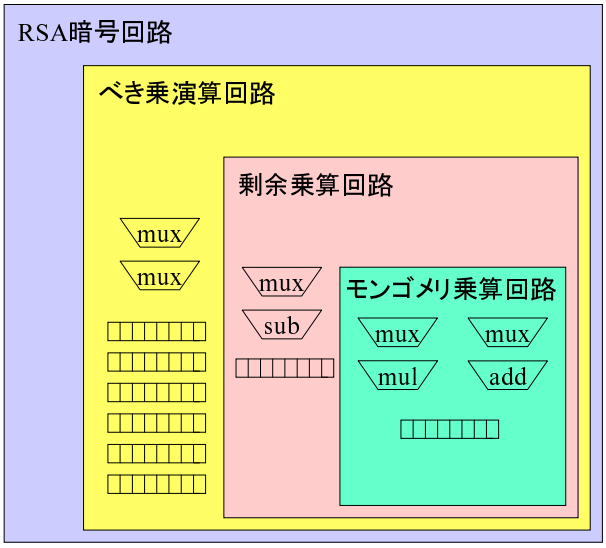


図 11: RSA 暗号回路の構造

Input :	a_i, V (m bits), N(n bits), W_i($n+1$ bits) $B = (b_{l-1} \cdots b_0)_{2^m}$ ($n+1$ bits),
Output :	W_{i+1}($n+1$ bits)
Step1:	$Q = (W_i + a_i b_0) V \bmod 2^m$
Step2:	$W_{i+1} = (W_i + a_i B + QN)/2^m$,

図 12: $n \cdot m$ のモンゴメリ乗算

4.1.2 $n \cdot m$ のモンゴメリ乗算回路

ここで行う演算は図 12 のようなものである。

図 12 では $a_i \cdot B$ のような $m \cdot n$ となる演算が行われているが、今回の評価では n ビットの数を m ビットのブロックに分割して $m \cdot m$ の乗算器を繰り返し使用することにより $m \cdot n$ の演算を行う方式を採用した。

また、図 12 の Step2 において、W_i と $a_i B$、QN の加算を行うが、このときに $m \cdot m$ の乗算器 1 つで $a_i \cdot B$ の演算を行い、W_i と足し合わせ、その後に Q を求めて $Q \cdot N$ の演算を行い、W_i と $a_i \cdot B$ の和に $Q \cdot N$ を足し合わせる方式と、あらかじめ Q を求めておき、$m \cdot m$ の乗算器を 2 つ用いて同時に $a_i \cdot B$ と $Q \cdot N$ を求め、W_i と足し合わせる方式の 2 つが存在する。後者の方が前者に比べて乗算器 1 つ分回路面積が増大するが、$n \cdot m$ のモンゴメリ乗算に必要なサイクル数を約半分まで削減することが出来る。

図 13 に乗算器 1 つの場合の回路構成、図 14 に乗算器 2 つの場合の回路構成を示した。W_i は図 12 で

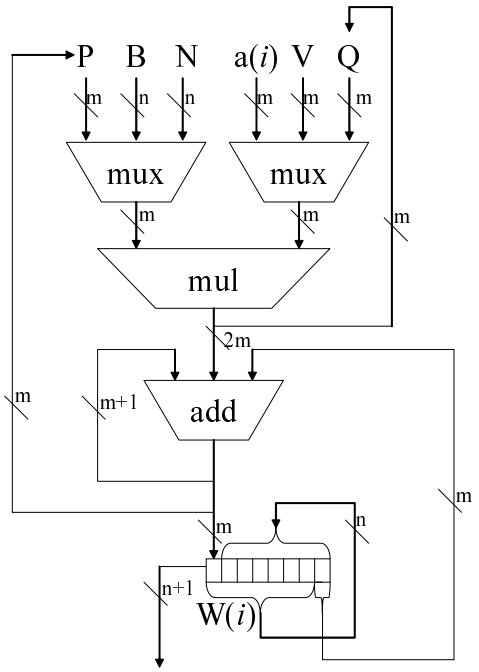


図 13: $n \cdot m$ のモンゴメリ乗算回路 1

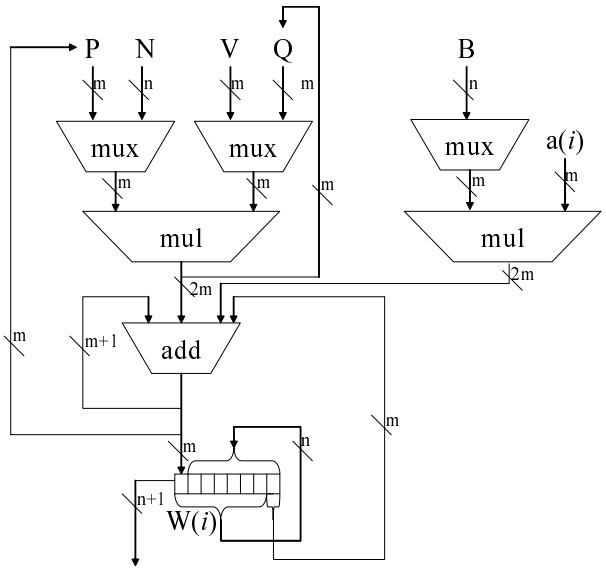


図 14: $n \cdot m$ のモンゴメリ乗算回路 2

は外部からの入力となっているが、実際には新しい a_i が来るたびに更新される値であるので、無駄な配線をなくすために内部のレジスタに格納して更新する方式を採用した。W_i を格納しているレジスタは、m ビットの右シフトし、追い出された m ビットと、乗算器の出力の和を取り、その下段 m ビットを空いた上段 m ビットに素の値を格納する、という動作をする。その和の上段 $m+1$ ビットは別のレジスタに格納しておき、次の乗算機の出力とともに新たに追い出された W_i の一部と足し合わせる。m ビットづつ処理を行っているため、最上段の乗算が終わった後に余計に 1 サイクル時間がかかるが、このようにすることで、Step2 の加算部分を乗算と同時に行うことができ、余計なサイクル数の増加を防ぐことができる。

これにより、乗算器を 1 つ用いる方式が、$a_i \cdot B$ を求め、W_i と足すために $n/m+1$ サイクル、Q を求めるために 1 サイクル、$Q \cdot N$ を求め、W_i と足すために $n/m+1$ サイクルの合計 $2n/m+3$ サイクルかかり、乗算器を 2 つ用いる方式が Q を求めるために $a_i \cdot B$ の最下段の乗算、その結果と W_i の下 m ビットの加算、その値と V との乗算の計 3 サイクル、その後に $a_i \cdot B$、$Q \cdot N$ を求め、W_i と足し合わせるために $n/m+1$ サイクルかかるため、合計で $n/m+4$ サイクルとなる。

4.1.3 $n \cdot n$ の剰余乗算回路

$n \cdot n$ の剰余乗算は a_i の値を更新しながら $n \cdot m$ のモンゴメリ乗算を n/m 回繰り返し、最後に W と N を比較し、W が N をより大きい場合には $W-N$ を出力し、そうでない場合はそのまま W を出力する。このとき、$W-N$ の演算回路をそのまま実装を行おうとすると、$n+1$ ビットの減算回路が必要となり、面積においてもクリティカルパスの長さにおいても有効ではないため、モンゴメリ乗算におけるの n ビットの演算と同様に m ビットのブロックに分割して演算を行う方式を採用した。この部分では n/m サイクルで減算を行い、1 サイクルでセレクタを通過する。

その回路構成を図 15 に示した。

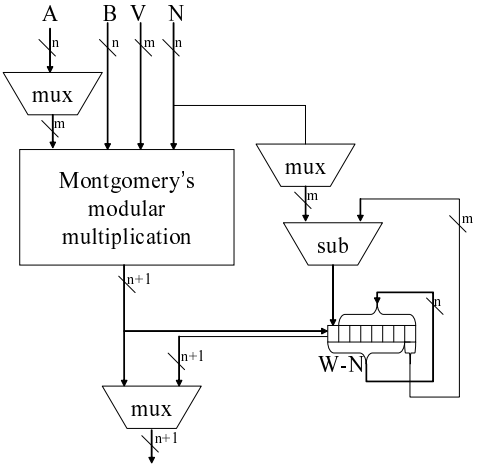


図 15: $n \cdot n$ の剰余乗算回路

4.1.4 べき乗演算回路

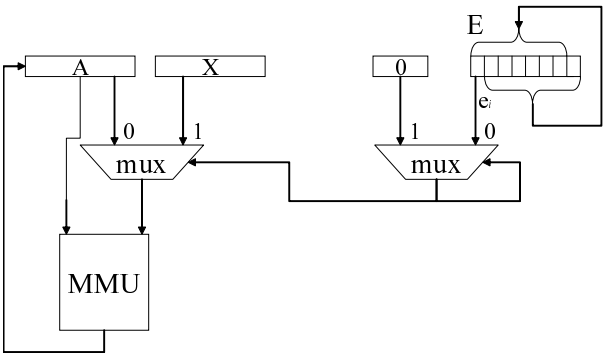


図 16: 右向きバイナリ法の回路構成

右向きバイナリ法、左向きバイナリ法、右向きアレイ法、左向きアレイ法それぞれの回路構成を図 16, 図 17, 図 18, 図 19 に示した。図中の MMU は剰余乗算回路をあらわす。

バイナリ法を $k = 1$ のアレイ法と考えたとき、べき乗演算回路は n ビットのレジスタを e を納めるシフトレジスタとは別に 2^k 個必要とする。このように並べて比較してみると、レジスタに納まる値と使用法が異なるのみで、右向き、左向き法における必要なレジスタや演算器に大きな差はないことがわかる。

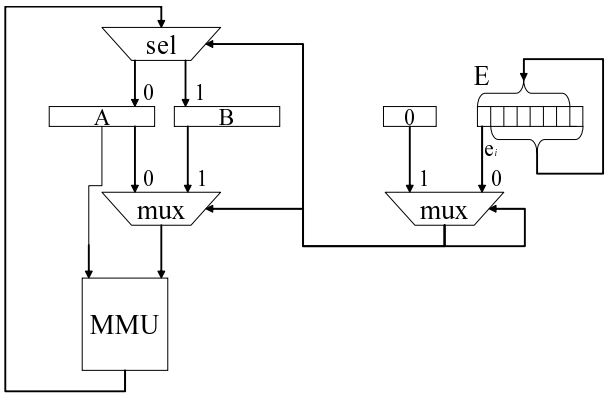


図 17: 左向きバイナリ法の回路構成

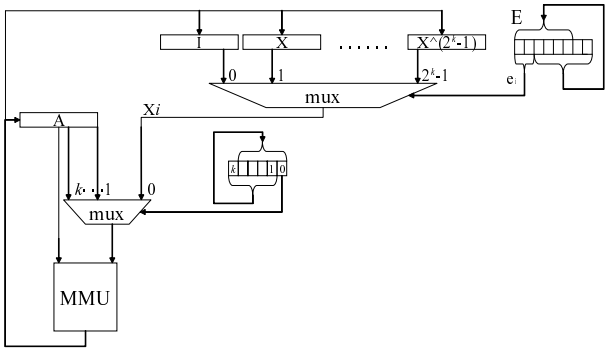


図 18: 右向きアレイ法の回路構成

4.2 評価環境

4.1 節で取り上げた $n \cdot n$ の剰余乗算回路において、用いる $m \cdot n$ のモンゴメリ乗算器に乗算器 1 つ用いたものと 2 つ用いたもの双方について m を 8,12,16,32,48,64,96,128 と変化させた場合の回路面積と処理にかかる時間を調べた。また、その中のいくつかの剰余乗算器を用いて、4.1.3 節であげたべき乗演算回路の構成で、アレイ法のパラメータ k を 1 から 5 までとしたときのべき乗演算回路全体の回路面積と処理速度がどのように変化するかを調べた。以上のことを n が 1024 と 2048 の場合において行った。

評価環境を表 (1) に示す。評価に用いたライブラリは日立製作所の仕様をもとに VDEC(VLSI Design and Education Center)[15] および京都大学が作成したものである。

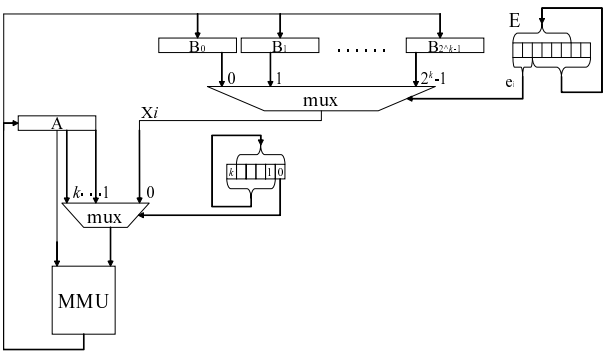


図 19: 左向きアレイ法の回路構成

表 1: 評価環境

実装言語	Verilog-HDL
論理合成系	DesignCompiler 2003.3 for sparcOS5
ライブラリ	日立製作所 CMOS 0.18μm

また、表 (1) にもあるが評価における加算器及び乗算器は DesignWare[16] を用いて合成を行った。オプションは加算器は cla(Carry Look ahead Adder)、乗算器は wall(wallace tree) を用いた。乗算器の wall オプションを用いた場合は桁上げ保存加算器を Wallace tree となるように並べて、部分積を 2 つになるまで加算し、最後に CLA 加算器を通して出力するという合成結果となる。図 20 に CSA と wallece 木を用いた 8 ビット乗算回路の構成を示す。

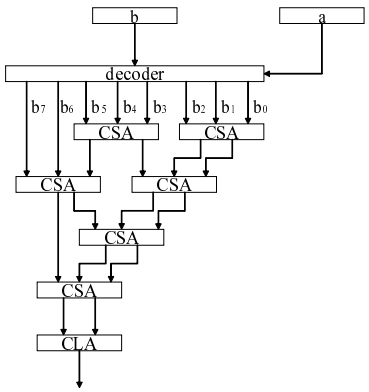


図 20: CSA と wallece 木を用いた 8 ビット乗算回路

なお、剰余乗算回路において 1 箇所減算が行われるが、そこでは補数表現を用いて加算器で演算を行わせた。

4.3 結果

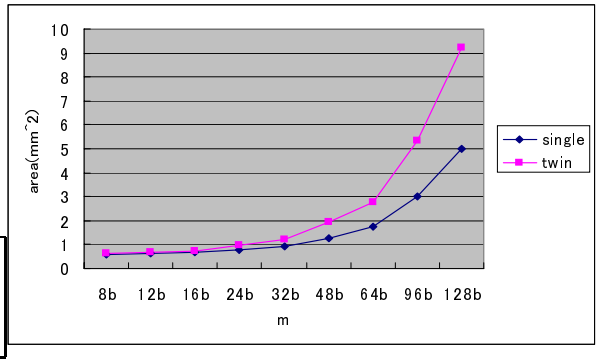


図 21: m による剰余乗算器の面積の変化 (n=1024)

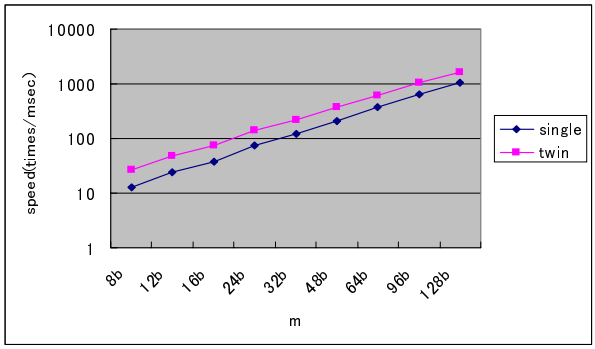


図 22: m による剰余乗算器の処理速度の変化 (n=1024)

m の値による剰余乗算器の面積と処理速度の変化の様子を図 21〜24 に示した。いずれの図も紺色の線で表したものが乗算器を 1 つで構成したもの、赤色の線で表したものが乗算器 2 つで構成したものである。面積は mm^2、処理速度については 1msec に何回剰余乗算を行えるかを表している。また、図 25,26 に m を倍にしたときの回路面積の処理速度の変化比

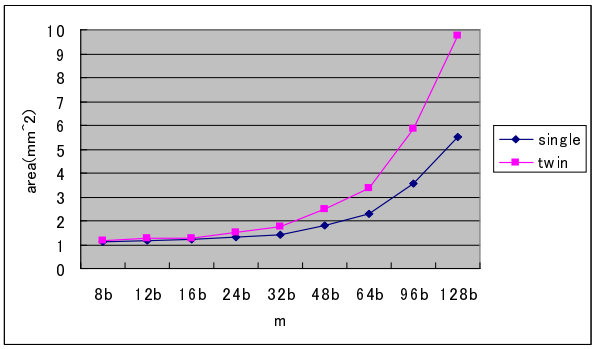


図 23: m による剰余乗算器の面積の変化 (n=2048)

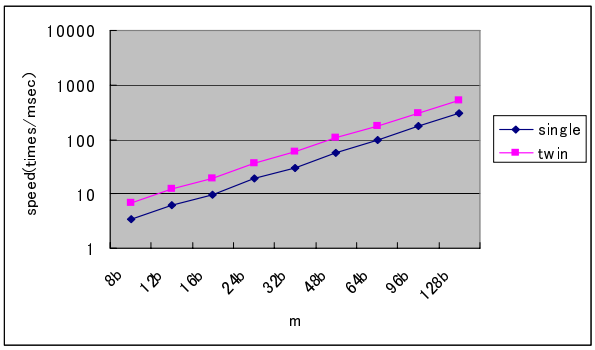


図 24: m による剰余乗算器の処理速度の変化 (n=2048)

を示した。

図 21,23 から m の値が小さいときは W_i を格納するレジスタや各セレクタ、$W - N$ を行う減算回路など周辺回路が乗算器の回路面積に比べて支配的であるために乗算器の数や m の値を大きくしてもさほど面積の増大は起こらない。周辺回路の面積は n の大きさによって変化するため、面積が大きく増大し始める m の値も変化する。$n = 1024$ の場合は m が 32 から 48 あたりで、$n = 2048$ の場合は m が 48 から 64 あたりから回路面積が目に見えて増加をはじめる。

一方、処理速度については全体的に乗算器を 1 つ使用したときと 2 つ使用したときでの速度比は約 2 倍、m の値を 2 倍としたときの速度比は約 3 倍となっている。ただし、処理速度においては回路面積とは

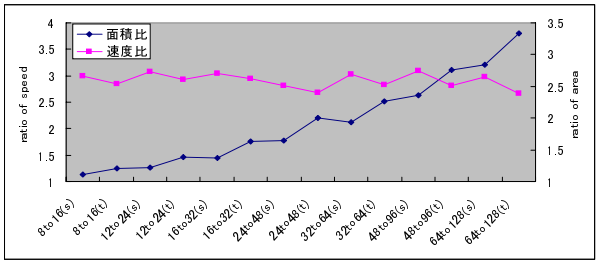


図 25: m を倍にしたときの面積と処理速度の変化比 (n=1024)

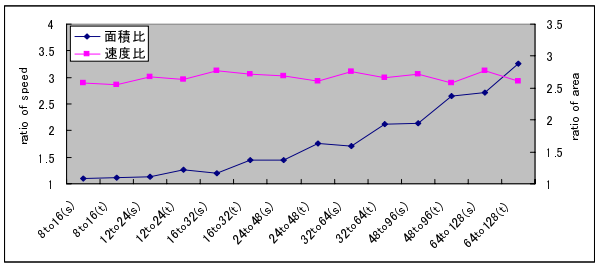


図 26: m を倍にしたときの面積と処理速度の変化比 (n=2048)

逆に m の値が大きくなったときに周辺回路での演算にかかるサイクル数が無視できなくなる。また、乗算器のビット数を大きくすることで、クリティカルパスが長くなり、1 サイクル当たりの時間も大きくなる。実際、$n = 1024$,$m = 64$〜128 のときにやや速度の伸びが低下していることが読み取れる。

以上のことから m が小さいときには回路において、大きいときには速度においてそれぞれ周辺回路の影響が大きくなることがわかった。

次に、べき乗演算回路の回路面積と処理速度のシミュレーション結果を図 27〜30 に示した。$m = 32, 64, 128$ の乗算器を 1 つ用いた構成の剰余乗算回路を用い、右向き法、左向き法双方で k を 1〜5 まで変化させた。図 27,29 から右向き法、左向き法における回路面積の差はどのようにパラメーターを変化させてもほとんど見られなかった。これは、右向き法、左向き法双方においてパラメーターが同一の場合、必要となる乗算器やレジスタの大きさや数が全く同一

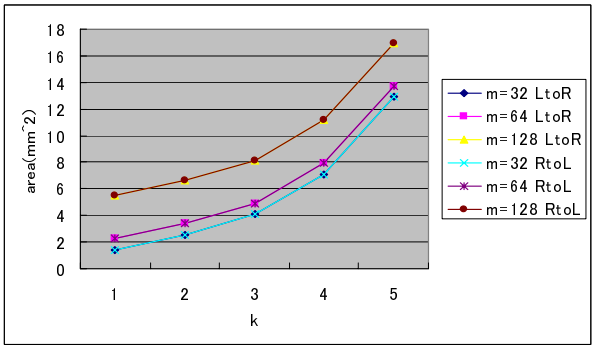


図 27: べき乗演算の回路面積 (n=1024)

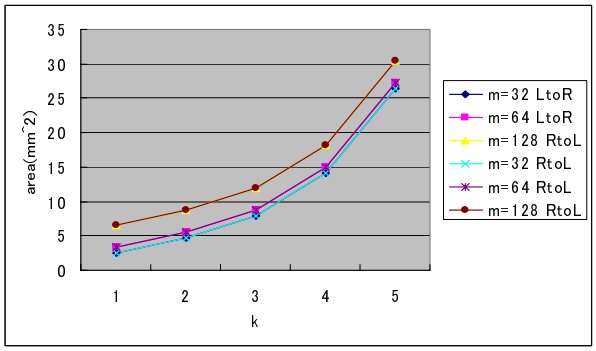


図 29: べき乗演算の回路面積 (n=2048)

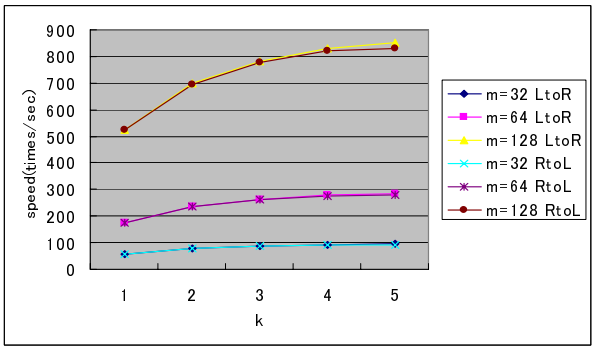


図 28: べき乗演算の処理速度 (n=1024)

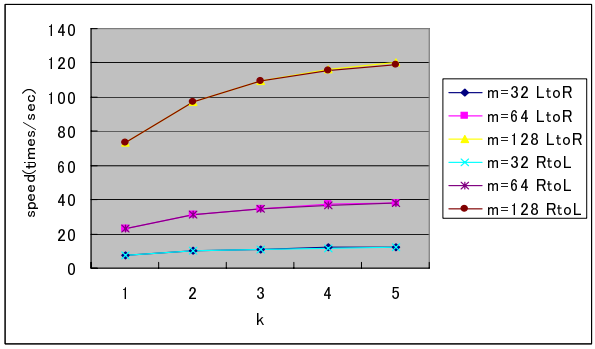


図 30: べき乗演算の処理速度 (n=2048)

であり、回路面積においてはこれらの割合が非常に大きいため、それ以外の細かい部分の違いが全体からみれば非常に小さなものとなり、目に見える形となって表れないためだと思われる。一方、処理速度については図 28,30 から k の値が大きくなるにつれてわずかではあるが、右向き法の方が優れていることが分かる。その理由は、3 章の最後で述べた通りに、左向き法の後処理演算に必要な乗算回数が右向き法の前処理演算に必要な乗算回数と比べて大きいため、その差が結果に現れたものと思われる。

次に、k を変化させたときの傾向をについて考える。k を大きくしたとき、処理速度の向上率は低下する一方で、回路面積の増加率は大きくなる。このときの面積増加は剰余乗算器とは相関が無く、k の値が大きくなるほど回路全体におけるべき乗演算回路の面積占有率は大きくなる。図 31,32 に各パラメターで回路全体での剰余乗算回路とべき乗演算回路の割合を示した。青い部分が剰余乗算回路で赤い部分がべき乗演算回路である。k の値が大きくなることによりべき乗演算回路部分の占有率が急激に増大することがわかる。また、図 31,32 の比較からべき乗演算回路における面積増加の原因は k の値によりその数が増大する n ビットのデータの記憶領域であるため、その面積占有率は n が大きくなることによりさらに増大することが読み取れる。

4.4 まとめ

剰余乗算回路の実験結果より、m が小さいときには回路において、大きいときには速度においてそれぞれ周辺回路の影響が大きくなることがわかった。ローエンド、ハイエンド向けに特化した回路を設計

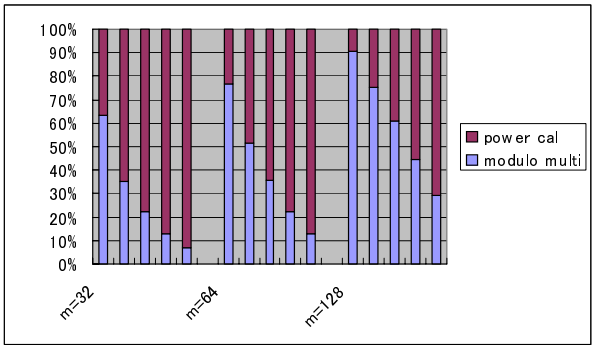


図 31: 剰余乗算回路とべき乗演算回路の面積比 (n=1024)

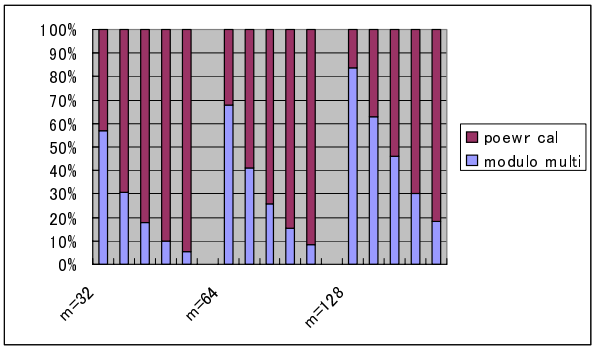


図 32: 剰余乗算回路とべき乗演算回路の面積比 (n=2048)

する際には、それぞれの設計思想に応じて周辺回路部分も小規模/低速なものや高速/大規模なものを用意する必要があるが、扱う数が 1024 ビット、2048 ビットと大きな数であるため、演算結果を格納するレジスタなど特化しきれない部分が存在する。特にハイエンド向けの構成を考慮する際は、周辺回路の問題以外にも、m が大きくなることで、クリティカルパスが長くなることによる速度向上率の低下などの問題も生じる。

一方、べき乗演算回路については右向き法と左向き法を比較し、本実験で用いた構成では回路面積はほぼ同等、処理速度においては右向き法がわずかに上回る、という結果が得られ、現状では左向き法に有用性が見られないことが分かった。また、k を大きくすることによる速度向上は大きいものの、回路面積の向上はそれ以上に大きなものであり、ある程度の速度を得ようとするとべき乗演算回路の面積が剰余乗算回路の面積と比べてかなり大きくなることも分かり、さらに、n が大きくなることによってべき乗演算回路の面積はさらに大きなものとなり、n を大きくすることにより暗号強度を上げる RSA の将来において深刻な問題となると考えられる。

5 親子剰余乗算器を用いた実装方式

5.1 n が大きくなる事の問題点

4 章の評価結果からハイエンド向けの構成を行う際、べき乗演算のパラメター k や剰余乗算器に用いる乗算器の数とビット数 m を大きい値とすることで高速化が実現できる。しかし、k は大きくしていくことで処理速度の向上率は低下し、面積における占有率が非常に大きな物となることがわかった。さらに、べき乗演算部分の回路面積は n を大きくすることでより剰余乗算回路の面積と比べて大きくなるため、将来におけるこの問題はさらに深刻なものとなる。

そこで、本研究ではべき乗演算回路部分の面積占有率の高さに着目し、1 つの暗号回路内に 2 つの剰余乗算回路を組み込むことで効率的にべき乗演算を行う方式を提案する。

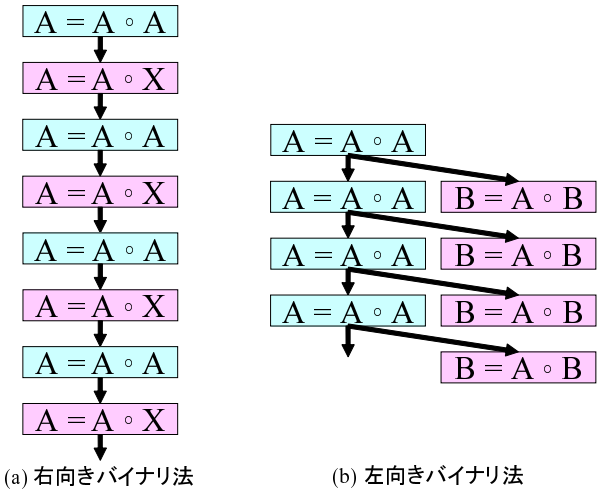
次節からその具体的な実装方法について検討を行う。

5.2 親子剰余乗算器を用いたべき乗演算

5.2.1 用いるべき乗演算方式

2 つの剰余乗算器を用いてべき乗演算を行う際、2 つの剰余乗算器を無駄なく並列に動作させるためにべき乗演算での剰余乗算同士の依存関係が少ないことが望ましい。

そこで、3.3 節で取り上げた右向き法、左向き法の



(a) 右向きバイナリ法　　(b) 左向きバイナリ法

図 33: 剰余乗算の依存関係

演算手順を調べる。まず、右向き法については、前処理の後に図 33(a) に示したように k 回の自乗演算 $(A \circ A)$ の後に 1 回の $A \circ X^{e_i}$ を行うという手順を n/i 回繰り返すが、このとき全ての剰余乗算において $A \circ A$、$A \circ X^{e_i}$ のどちらも直前の剰余乗算の結果を必要とする。一方、左向き法に付いては、最後に後処理演算を行い、図 33(b) に示したように k 回の自乗演算の後に 1 回の $A \circ B_{e_i}$ を行うという手順は X^{e_i} が B_{e_i} に変わったのみであるが、右向き法においては k 回の自乗演算に $A \circ X^{e_i}$ の結果を用いる必要があるのに対して、左向き法は k 回の自乗演算に $A \circ B_{e_i}$ の結果を必要としない。

よって、剰余乗算器を 2 つ用いて無駄なく並列に動作させるためには、左向き法を用いて片方を自乗演算に使い、もう片方を $A \circ B_{e_i}$ に使えばよい。このとき、べき乗演算を行うために必要な時間は、自乗演算を n 回分と、アレイ法の場合はそれにさらに後処理演算を $2 \cdot (2^k - 2)$ 回分の合計である。図 34 に k の値による本手法の演算時間の変化を示した。既存のべき乗演算の実装方式では k の値を大きくすることで演算時間が短くなるが、本手法においては既存方式で削減される部分である $A \circ B_{e_i}$ の演算を行う時間は削減されない部分である自乗演算を行う時間に隠されてしまうため、後処理演算の回数が増えることもあり、演算時間が逆に長くなる。一方で、k の値を大きくした場合、$A \circ B_{e_i}$ は自乗演算を k 回行う間に 1 回行えばよいので、$A \circ B_{e_i}$ を行う剰余乗算器の処理速度は自乗演算を行う剰余乗算器の $1/k$ の速度でよいこととなる。このことから $A \circ B_{e_i}$ を行う剰余乗算器は自乗演算を行う剰余乗算器よりも用いる乗算器の数を減らすことや、m の値を小さくして面積を削減することが出来る。m が 48,64 ビットあたりから剰余乗算器の回路面積が急激に上昇するため、親剰余乗算器が 64 ビット、128 ビットの乗算器を用いている場合は k を大きくすることによるべき乗演算回路部分の面積増加よりも剰余乗算器の面積削減効果の方が大きくなることが期待できる。

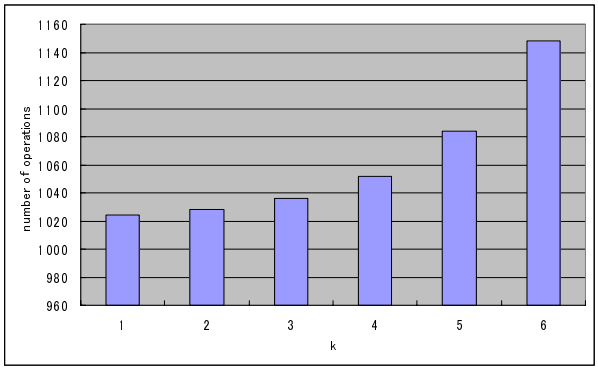


図 34: k の値による提案手法の演算時間の変化 (n=1024)

このように条件次第で 2 つの剰余乗算器の大きさに差が出ることから、本論文では自乗演算を行う剰余乗算器を親剰余乗算器、$A \circ B_{e_i}$ を行う剰余乗算器を子剰余乗算器と呼ぶこととする。

5.2.2 回路構成

図 35,36 に剰余乗算器を 2 つ用いた左向き法の回路構成を示す。アレイ法、バイナリ法共に左が親剰余乗算器、右が子剰余乗算器である。$A \circ B_{e_i}$ を行うために子剰余乗算器には B_d を格納するレジスタからの入力が必要なことに対して、親剰余乗算器は自乗演算のみを行うため、それらのレジスタからの

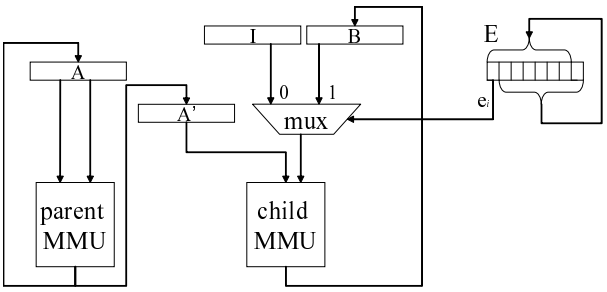


図 35: 剰余乗算器を 2 つ用いた左向きバイナリ法

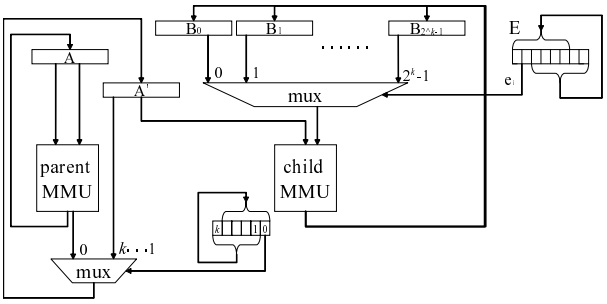


図 36: 剰余乗算器を 2 つ用いた左向きアレイ法

入力の必要が無いため、後処理演算のみのために配線を行うことが無駄であること、親剰余乗算器は自乗演算以外を行わないことで本来 2 つ必要な乗数と被乗数を格納するレジスタを 1 つに押さえられること、後処理にかかる剰余乗算の回数が $n = 1024$ の場合で $k = 4$ の場合においても 3%未満であることなどの理由から、左向きアレイ法の最後に必要な後処理演算

$$\prod B_d^d (d = 1, \cdots, 2^k - 1)$$

は子剰余乗算器で行う。

5.3 親剰余乗算器の自乗演算特化手法

左向きアレイ法を用いる場合は基本的に k 回の自乗演算を行う親剰余乗算器の演算時間が全体の演算時間を決定する。そこで、親剰余乗算器は自乗演算のみを行うことに着目し、自乗演算に特化した剰余乗算回路の実装方式を親剰余乗算器に適用する。

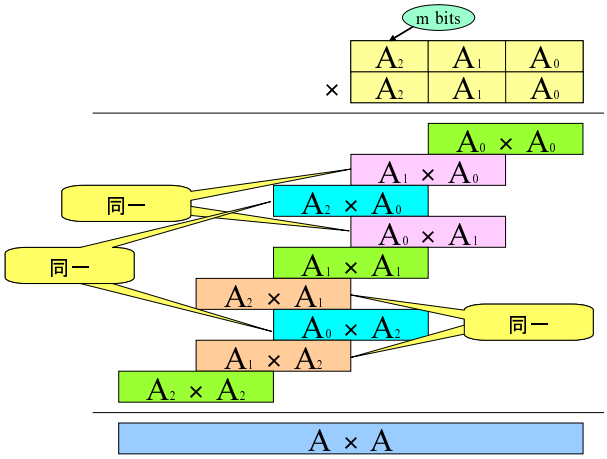


図 37: n ビットの自乗演算

図 37 に示したように m ビットのブロックに分割して n ビットの自乗演算を行う際に、$a_i \cdot a_j$ と $a_j \cdot a_i$ の結果が一致することを利用して、図 38 のような手法を用いることで、$m \cdot m$ ビットの乗算器を使用する回数を $(n/m)^2$ 回から $m + \{(n/m)^2 - m\}/2$ 回に削減することが出来る [14]。

Input :$a = (a_{l-1}, a_{l-2}, \cdots, a_0)_{2^m}$ Output :$m = a^2$
$m = 0$ for $i = 0, i < l, i++$ ($m = m + a_i \cdot a_i$ for $j = i + 1, j < l, j++$ ($m = m + 2 \cdot a_i \cdot a_j$))

図 38: 自乗演算

本研究での親剰余乗算器は自乗演算のみを行うため、この手法を利用して剰余乗算に必要なサイクル数の削減が行えないかの検討を行う。

乗算器を 1 つ使用した構成の場合、$a_i \cdot B$ と w_i の加算を行い、その後に Q を求め、$Q \cdot N$ と w_i の加算を行う。この際、$a_i \cdot B$ の部分に自乗演算の特化手法を適用できる。しかし、その値を使用する w_i の加算

において、特化手法は $a_i \cdot b_i$(ここでは $a_i = b_i$) という w_i の下から $i+1$ 番目のブロックと加算を行う部分積を処理するのに対し、w_i はシフトレジスタに格納されており、モンゴメリ乗算が始まる際には最下段の m ビットから処理を行おうとするため、いきなり途中の部分積を途中のブロックと加算を行うことはできない。そこで、i が 1 より大きい場合は w_i の下 m ビットにおいてすでに $a_i \cdot b_0$ が足されている状態となっていることから、それと V を掛けて Q を求め、下から i 番目のブロックまでは $Q \cdot N_j$ とのみ加算を行い、それ以上のブロックにおいては w_i のシフトに合わせて $a_i \cdot b_j$ もしくは $2 \cdot a_i \cdot b_j$ と $Q \cdot N_j$ の加算を交互に行う方式を採用した。

このようにして構成した回路が図 39 である。このモンゴメリ乗算回路を用いた剰余乗算器を自乗演算に特化した剰余乗算器とする。

乗算器を 2 つ使用した構成の場合は $a_i \cdot B$ と $Q \cdot N$ が同時に行われるため、$a_i \cdot B$ に必要なサイクル数を削減を行っても $Q \cdot N$ の演算にかかるサイクル数が削減できないため、高速化が行えない。

また、バイナリ法では子剰余乗算器が親剰余乗算器と同等の処理速度が求められるため子剰余乗算器の方が親剰余乗算器の足を引っ張る形となり、高速化手法が無意味なものとなる。

このように剰余乗算器は内部のモンゴメリ乗算器において乗算器を 1 つもちいたもので、かつ、べき乗演算方式は $k = 2$ 以上のアレイ法でのみ有効、というかなり限定的な条件の元でのみの手法となるが、適用できた場合はさほど面積を増やすことなく 2 割程度の処理速度の向上を見込むことができる。

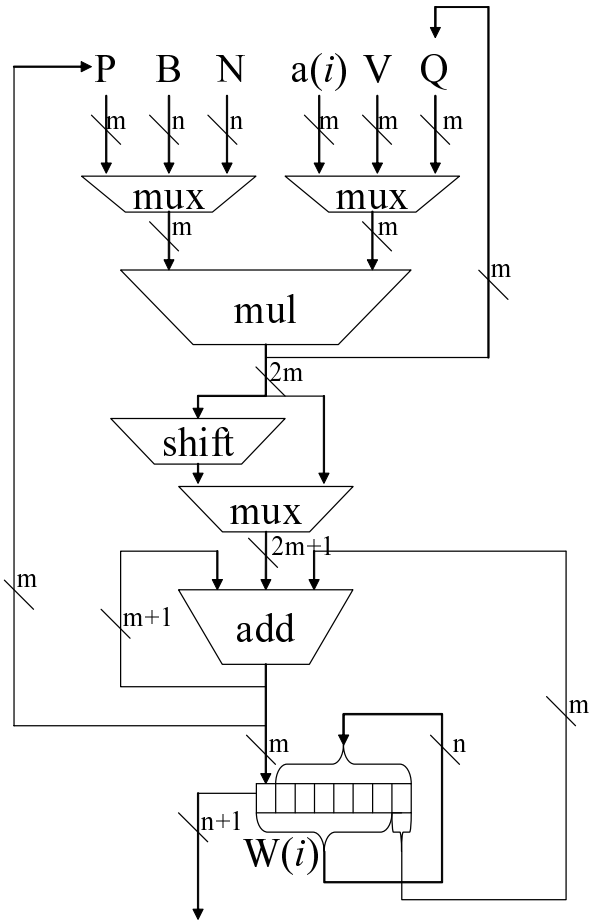


図 39: 自乗演算に特化したモンゴメリ乗算回路

バイナリ法の回路面積と処理速度の比較を行った。その後に、親子剰余乗算器を用いた左向きアレイ法において k を 1 から 4 まで変化させたときの回路面積と処理速度の変化を調べた。また、親剰余乗算器に自乗演算の特化した回路を実装し、その処理速度向上率の評価を行った。

パラメターとして暗号強度を示すパラメター n を予備評価同様に 1024,2048 とし、右向きアレイ法のパラメター k を 1 から 4 までとした。右向き法及び提案手法の親剰余乗算器に用いる乗算器のビット数 m は 8,16,32,64,128 としたが、子剰余乗算器は 1 回の剰余乗算に必要なサイクル数が親剰余乗算器の k 倍以下でよいとするため、それとは別に m =12,24,48,96

6 評価

6.1 評価環境

提案手法である親子剰余乗算器を用いた左向き法を実装し、予備評価においてより優れた性能を示した右向き法と比較を行った。

まず、右向き法と親子剰余乗算器を用いた左向き

の剰余乗算器も用意し、必要に応じて適用する。乗算器の数は 1 つの場合と 2 つの場合の両方を用いる。

評価環境は 4 章の予備評価と同じもの (表 (1)) を用いた。

6.2 評価

6.2.1 親子剰余乗算器を用いた左向きバイナリ法の評価

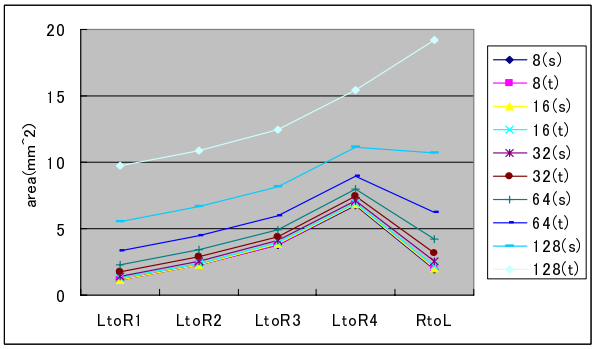


図 40: 親子左向きバイナリ法と右向き法の比較 (area,$n = 1024$)

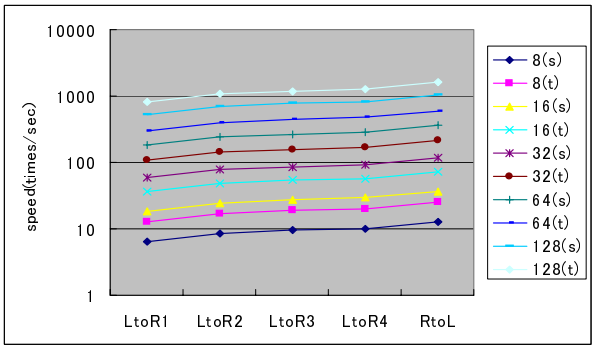


図 41: 親子左向きバイナリ法と右向き法の比較 (speed,$n = 1024$)

親子剰余乗算器を用いた左向きバイナリ法の実装を行い、その処理速度と回路面積を測定した。その結果と 4 章で測定した右向き法の結果と比較したグラフを図 40,41,42,43 に示した。

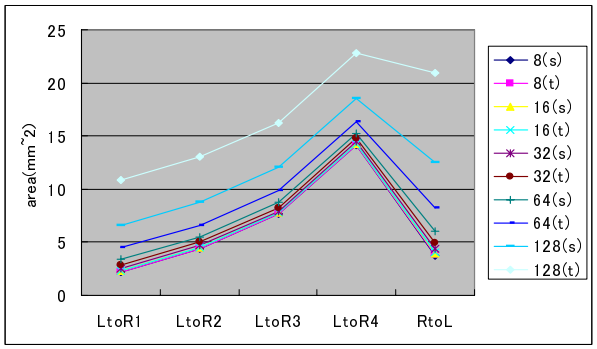


図 42: 親子左向きバイナリ法と右向き法の比較 (area,$n = 2048$)

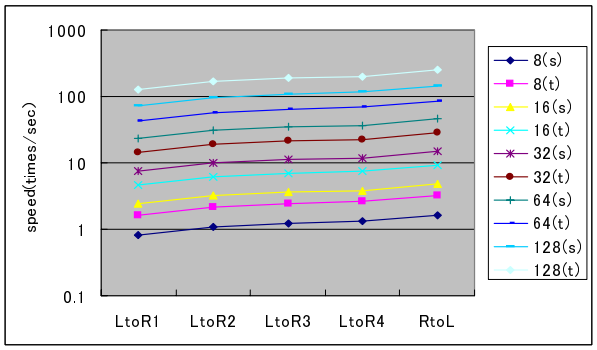


図 43: 親子左向きバイナリ法と右向き法の比較 (speed,$n = 2048$)

n=1024 の場合、図 41 から左向きバイナリ法の処理速度は n や m の大きさと無関係に右向き法の $k = 4$ の場合と比べて 25%程度向上している。n=2048 の場合も図 43 から同じ剰余乗算器を使用したときの処理速度が n=1024 の場合と比べて約 4 分の 1 となるものの、全体の傾向はまったく同じであることがわかる。よって、処理速度は n,m に関係無く右向き法の $k = 4$ を比べて 25%程度上昇することがわかった。

一方で、図 40 から m が 8 や 16 の場合は剰余乗算器が k を大きくすることによるレジスタの面積と比べて小さいため、左向きバイナリ法の回路面積は $k = 2$ よりも小さなものとなるが、m が 32 や 64 となると剰余乗算器を 2 つ使用することによる面積増加が大きなものとなり、$k = 2$ から 3 の間あたりと

同等となる。さらに、m が 128 となると、剰余乗算器の大きさはかなりの大きさとなるため、特に乗算器を 2 つ試用した構成の場合、その面積は $k = 4$ の場合よりも 20%ほど大きなものとなる。こちらの場合、n を大きくすると図 42 と図 40 の比較からわかるように左向きバイナリ法の面積が右向き法の k を大きくした場合に比べて小さくなる。この理由は次のように考えられる。n を大きくしたときに剰余乗算器の面積は乗算器以外の周辺回路が 2 倍弱となるのみであり、m を大きくした場合にはその影響があまり大きなものではなくなるのに対し、べき乗演算部分はその面積のほとんどが n ビットの数字を収めるためのレジスタであるため、n の大きさが倍になると、べき乗演算部分の回路面積もほぼ倍になる。よって n を大きくした場合の影響は剰余乗算器を 2 つ使うことで速度を向上させる左向きバイナリ法よりもレジスタを増やすことで速度を向上させる右向き法の方が大きく受けることとなる。以上のことから、親子剰余乗算器を用いた左向きバイナリ法は、回路面積において剰余乗算器が大きなものになるほど右向き法に対する優位性は小さな物になるが、一方で n を大きくするほど優位性が増し、将来において有利となる手法であることがわかった。

ここまでの課題として考えられることは、$n = 1024, m = 128(t)$ において回路面積が $k = 4$ の場合と比べて 20%増加していることである。処理速度が 25%向上していることを考慮すればこれは決して不利な数字ではないが、剰余乗算器の回路面積が大きくなると優位性がなくなるということは親子剰余乗算器を使用した方式がハイエンド向けの構成用である以上好ましいものではなく、この問題を解決するために高い処理速度を維持したままで面積を削減できる手法が必要となる。

6.2.2 左向きアレイ法による回路面積の削減効果

本節では親子剰余乗算器を用いる手法に左向きアレイ法を適用することにより親剰余乗算器が大きくなったときに高い処理速度を維持しつつ回路面積を削減することができるか評価を行った。

まず、表 (2) に剰余乗算器の面積とサイクル数についてまとめた。左向きアレイ法を適用した場合、子剰余乗算器に用いる剰余乗算器は、サイクル数が親剰余乗算器の k 倍以下であれば親剰余乗算器が k 回自乗演算を行う前に処理を終わらせることができる。よって、子剰余乗算器に用いる剰余乗算器のパラメターは、サイクル数の条件を満たすものの中で、最も面積の小さいものとする。以上の条件で選んだ結果、表 (3) のようになった。表の中で (s) となっているものが乗算器を 1 つ用いたもの、(t) となっているものが乗算器を 2 つ用いたものである。

このとき、2 つの剰余乗算器のみの回路面積を測定し、その結果を図 44.45 に示した。$n = 1024$ の場合、2048 の場合で共に m が 32 あたりまではほとんど面積は変わらないが、それよりも大きくなると明らかに回路面積が削減されている。

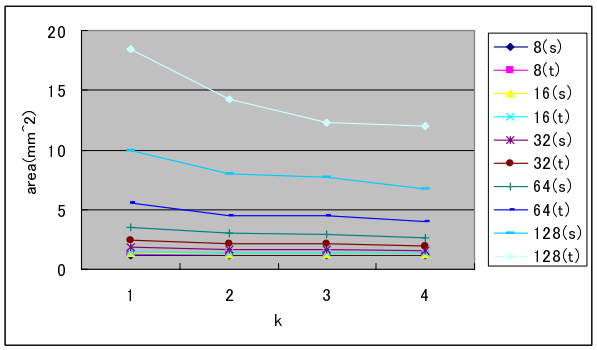


図 44: k の値による親子剰余乗算器の面積の変化 $(n = 1024)$

以上の条件での親子剰余乗算器を用いた左向きアレイ法のパラメター k と回路面積、処理速度の関係を図 46,47,48,49 に示した。

図 44,45 から予測できたが、m が 32 よりも小さい範囲ではアレイ法を用いることによる回路面積の削減効果はまったく見られず、逆に増加する結果となった。m が 64 の場合でも、剰余乗算器そのものでは面積が目に見えて削減されていたが、それ以上にレジスタを増やしたことによる面積の増加が大きく、

表 2: 剰余乗算器の面積とサイクル数

m	$n = 1024$				$n = 2048$			
	single mul		twin mul		single mul		twin mul	
	cycle	area(mm^2)	cycle	area(mm^2)	cycle	area(mm^2)	cycle	area(mm^2)
8	33281	0.596	17025	0.627	132097	1.135	66817	1.165
12	15137	0.640	7827	0.698	59167	1.188	30097	1.245
16	8449	0.661	4417	0.752	33281	1.229	17025	1.280
24	3871	0.780	2065	0.967	15137	1.323	7827	1.510
32	2177	0.905	1185	1.224	8449	1.432	4417	1.749
48	1057	1.280	595	1.934	3871	1.821	2065	2.473
64	577	1.749	337	2.774	2177	2.271	1185	3.384
96	287	3.013	177	5.336	1057	3.554	595	5.875
128	161	4.981	105	9.235	577	5.515	337	9.755

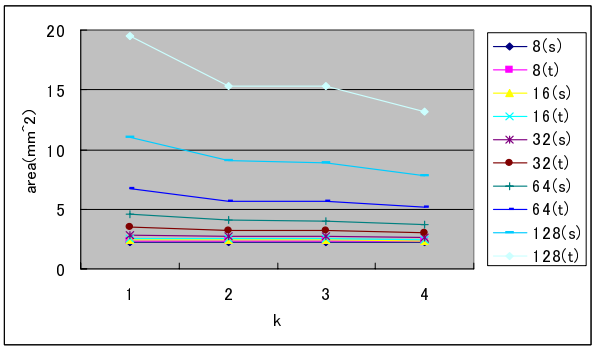


図 45: k の値による親子剰余乗算器の面積の変化 ($n = 2048$)

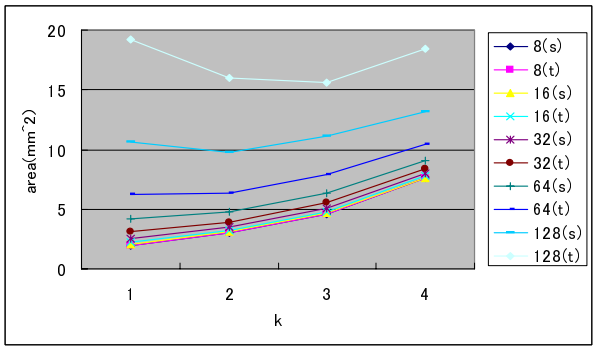


図 46: 親子剰余乗算器を用いた左向きアレイ法の回路面積 ($n = 1024$)

逆に面積が増える結果となった。m が 128 の場合のみ回路面積の削減効果が見られた。ただし、n が大きくなるほどレジスタ 1 つあたりの面積が大きくなるため、k を大きくしたことによるべき乗演算部分の回路面積の増加量が大きくなるため、n=1024 と比べて 2048 の場合は回路面積の削減効果は小さなものとなった。

処理速度については図 47,49 からわかるように n,m に関係無くほとんど低下しない。図 47,49 は対数グラフであるため細かい数字がわかりにくので、回路面積の削減効果の大きかった n=1024, 親剰余乗算器の m が 128(t) の場合での k の値による処理速度の変化を 50 に示した。$k = 2$ のときに $k = 1$ と比べて 1%以下、$k = 3$ のときに 2%程度の低下で面積の削減効果が得られていることがわかる。他のパラメターの場合は細かく示さないが、基準となる処理速度が違うだけで k を大きくすることによる処理速度の変化の傾向はほぼ同一である。

以上のことから、左向きアレイ法を用いることによってパラメター次第では高々2%程度の処理速度の低下で最大 20%程度の面積の削減効果が得られることがわかった。削減効果を得られる条件は n に比し

表 3: 子剰余乗算器の乗算器の数とビット数

親剰余乗算器	$n = 1024$				$n = 2048$			
	$k = 1$	2	3	4	$k = 1$	2	3	4
8(s)	8(s)	8(s)	8(s)	8(s)	8(s)	8(s)	8(s)	8(s)
8(t)	8(t)	8(s)	8(s)	8(s)	8(t)	8(s)	8(s)	8(s)
16(s)	16(s)	12(s)	8(t)	8(s)	16(s)	12(s)	8(t)	8(s)
16(t)	16(t)	16(s)	16(s)	8(t)	16(t)	16(s)	16(s)	8(t)
32(s)	32(s)	24(s)	16(t)	16(s)	32(s)	24(s)	16(t)	16(s)
32(t)	32(t)	32(s)	32(s)	16(t)	32(t)	32(s)	32(s)	16(t)
64(s)	64(s)	48(s)	32(t)	32(s)	64(s)	48(s)	32(t)	32(s)
64(t)	64(t)	64(s)	64(s)	32(t)	64(t)	64(s)	64(s)	32(t)
128(s)	128(s)	96(s)	64(t)	64(s)	128(s)	96(s)	64(t)	64(s)
128(t)	128(t)	128(s)	96(s)	64(t)	128(t)	128(s)	128(s)	64(t)

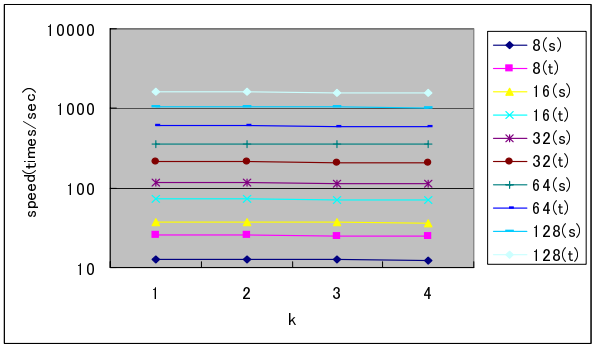


図 47: 親子剰余乗算器を用いた左向きアレイ法の処理速度 ($n = 1024$)

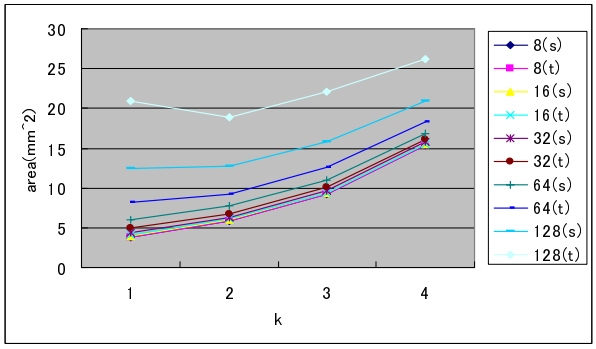


図 48: 親子剰余乗算器を用いた左向きアレイ法の回路面積 ($n = 2048$)

て親剰余乗算器の m が大きくなった場合という限られたものであるが、左向きバイナリ法によって回路面積で問題となるのがまさにその条件であるため m が小さい場合は左向きバイナリ法を用い、m が大きくなったときにアレイ法を用いて面積の削減を行えば良い。

6.2.3 親剰余乗算器に対する自乗演算特化手法の適用

親剰余乗算器は自乗演算のみを行えば良いため、それに特化した手法を適用することが出来る。本節ではその評価を行う。

まず、既存の剰余乗算器に自乗演算特化手法を適用し、その回路面積と処理速度の変化を評価した。その結果を図 51,52,53,54 に示した。

特化手法を適用した回路はシフタとセレクタがそれぞれ 1 つづつ増えているが、その面積はさほど大

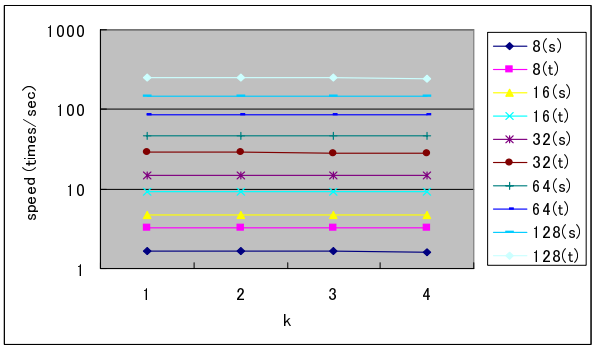


図 49: 親子剰余乗算器を用いた左向きアレイ法の処理速度 ($n = 2048$)

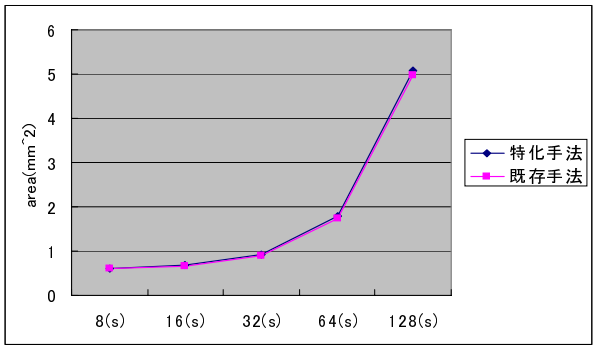


図 51: 自乗演算特化手法による回路面積の変化 ($n = 1024$)

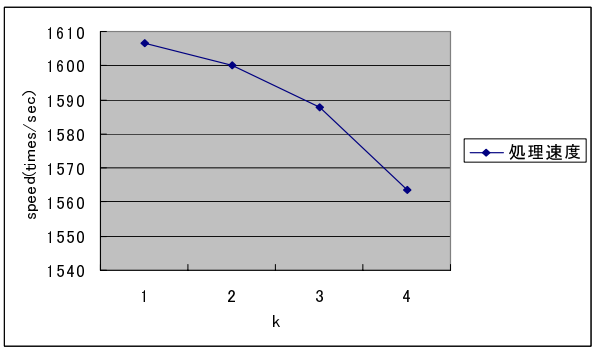


図 50: k の増加による処理速度の変化 ($n = 2048,128(s)$)

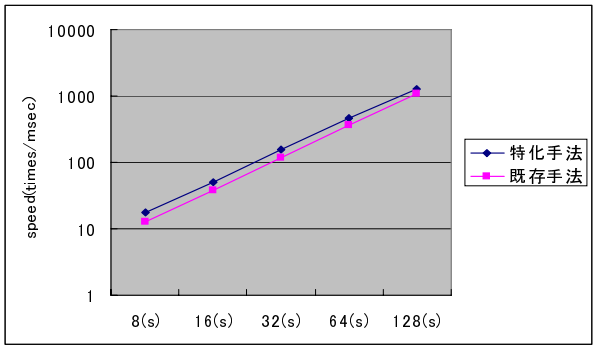


図 52: 自乗演算特化手法による処理速度の変化 ($n = 1024$)

きな物ではなく、全体の回路面積はさほど増加をしていない。

一方、パラメターに関係無く 1 回の剰余乗算に必要なサイクル数が 2 割程度減少するため、それに伴って処理速度も 2 割程度向上する。

親剰余乗算器はこの剰余乗算器を用いてもう一度左向きアレイ法の評価を行う。このとき、親剰余乗算器のサイクル数は表 (2) と比べて削減されているため、表 (3) の組み合わせは使えない。そこで改めて最適な剰余乗算器の組み合わせを調べ、その結果得られた表 (4) の組み合わせで評価を行った。

特化手法を適用したアレイ法の回路面積と処理速度を前節の結果と比較したグラフを図 55,56,57,58 に示した。

剰余乗算器が 2 割程度処理速度が向上していることにより、それがそのまま回路全体の処理速度に影響し、全体的に従来手法の回路と比べて 20%程度向上が見られる。

一方、回路面積については $n = 1024,m = 128$ において 12%、上昇率の最も大きい $n = 2048,m = 8$ において 55%であったが、$n = 2048,m = 8$ においては右向きアレイ法の $k = 3$ の場合の面積よりも小さく、速度向上には十分見合うと考えられる。。

6.3 まとめ

最後に本章で取り上げた各手法によって回路面積と処理速度がどのように変化するのかを図 59,60,61,62

表 4: 子剰余乗算器の乗算器の数とビット数（自乗演算特価方式）

	$n = 1024$			$n = 2048$		
親剰余乗算器	$k = 2$	3	4	$k = 2$	3	4
8(s)	8(s)	8(s)	8(s)	8(s)	8(s)	8(s)
16(s)	16(s)	8(t)	8(t)	16(s)	8(t)	8(t)
32(s)	32(s)	16(t)	16(t)	32(s)	16(t)	16(t)
64(s)	64(s)	32(t)	32(t)	64(s)	32(t)	32(t)
128(s)	128(s)	64(t)	64(t)	128(s)	64(t)	64(t)

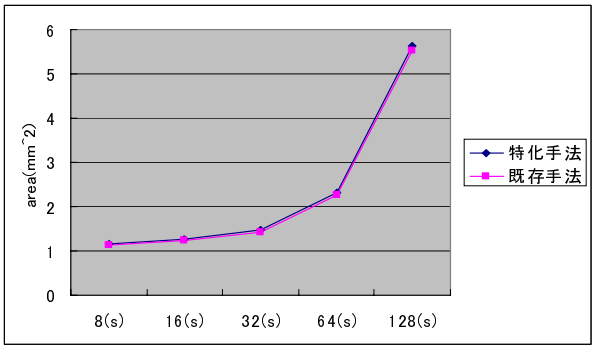


図 53: 自乗演算特化手法による回路面積の変化 ($n = 2048$)

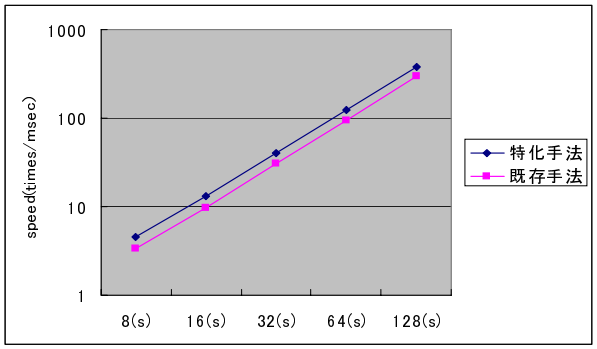


図 54: 自乗演算特化手法による処理速度の変化 ($n = 2048$)

に示した。LtoR の右の数字は右向きアレイ法の k、RtoL_b は左向きバイナリ法、RtoL_a は左向きアレイ法 $k = 1$ から 4 の中で最も面積の小さかったもの、RtoL_s は左向きアレイ法に自乗演算特化方式を適用したものを表す。いずれのグラフも右向きバイナリ法の値を 1 としたときの相対値で表した。

回路面積はほとんどのパラメターにおいて左向きバイナリ法を用いて剰余乗算器を 2 つに増やした場合でも、右向きアレイ法の $k = 3$ の場合の面積よりも小さく、それより大きくなる場合は左向きアレイ法を用いて子剰余乗算器の面積を削減することで全体の面積を削減できることがわかる。また、親剰余乗算器を自乗演算に特化させるために $k = 1$ が最も面積の小さい場合でも $k = 2$ 以上にする必要があるが、その場合でもその面積は右向きアレイ法の $k = 3$ の場合と同じか小さくなることがわかる。

一方、処理速度については全体的に左向き法を用いることにより、右向きバイナリ法の 2 倍、右向き法の $k = 4$ と比較して 25%向上していることがわかる。さらに、自乗演算特化手法を用いることにより、さらに最大 33%の速度向上が得られる。

以上の結果から、処理速度においては提案手法が従来用いられている右向きアレイ法と比べて 25%から 65%向上することがわかった。特に今回の評価のように、アレイ法に必要な X^{e_i}, B_{e_i} を格納する記憶領域まで回路面積に含めた場合、面積においても提案手法が有効となることがわかった。

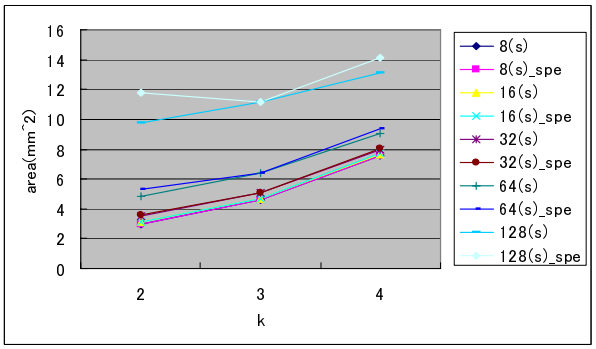


図 55: 特化手法を適用したアレイ法の回路面積 ($n = 1024$)

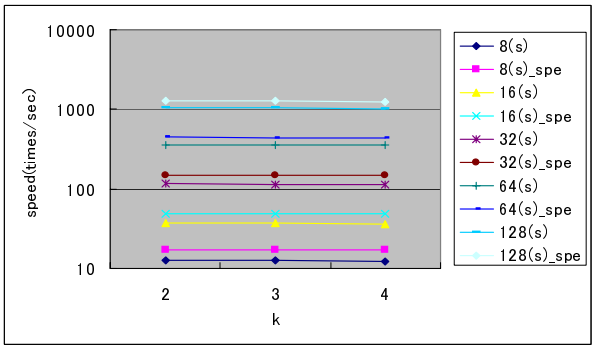


図 56: 特化手法を適用したアレイ法の処理速度 ($n = 1024$)

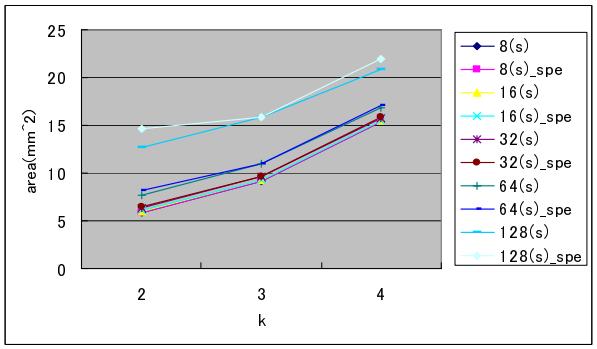


図 57: 特化手法を適用したアレイ法の回路面積 ($n = 2048$)

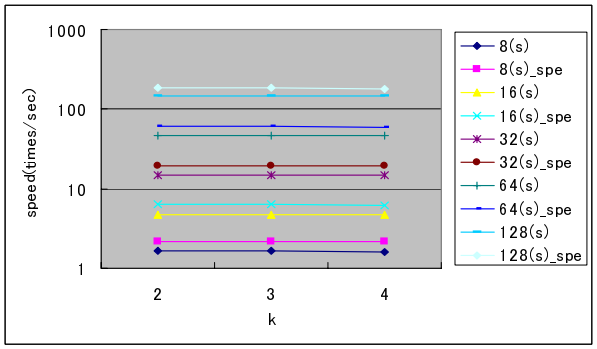


図 58: 特化手法を適用したアレイ法の処理速度 ($n = 2048$)

7 おわりに

7.1 まとめ

本論文では RSA 暗号のハードウェア実装におけるべき乗演算方式についての検討を行った。RSA 暗号のハードウェア実装を行う際、高速なべき乗演算とされている右向きアレイ法を行うためは膨大な記憶領域が必要である。また、その記憶領域はセキュリティパラメターである鍵長を大きくすることによりさらに大きくなり、その記憶領域を確保するために回路全体の面積は非常に大きなものとなる。また右向きアレイ法は高速化のパラメター k が前処理演算にかかる時間にも影響するため、ある程度大きくなると k を大きくすることによる処理速度の増加よりも前処理演算にかかる時間が増えることによる処理速度の低下の方が大きくなり、その高速化には限界がある。本研究は剰余乗算器を 2 つ用いて並列性の高い左向きアレイ法を用いることにより、回路面積は右向きアレイ法の一般的に用いられる k=2 あるいは 3 とほぼ等しい値に抑えつつ、処理速度は右向きアレイ法で最速となる $k = 4$ から 25%程度の速度向上が得られた。また、自乗演算を行う親剰余乗算器に自乗演算の特化手法を適用することで、回路面積はパラメターにより 12%から 55%ほど増えるものの、さらに 30%程度の速度向上を得ることができた。

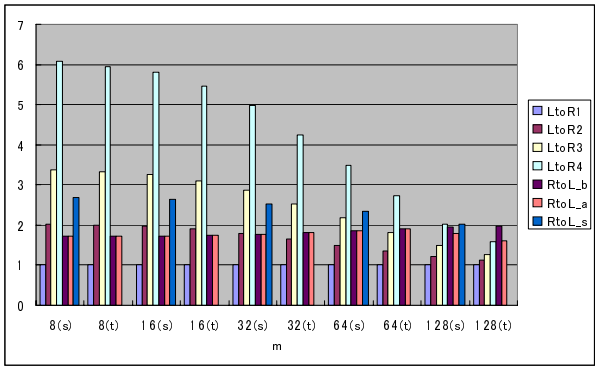


図 59: 各手法による回路面積の変化 ($n = 1024$)

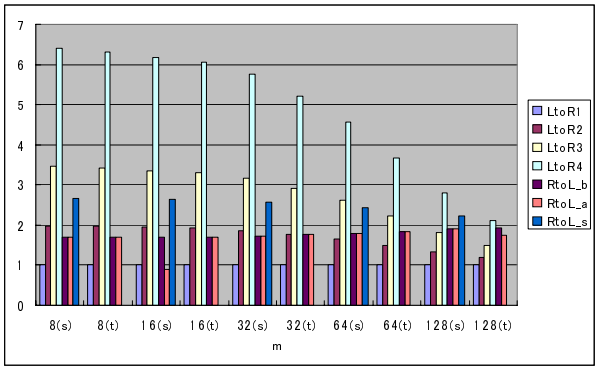


図 61: 各手法による回路面積の変化 ($n = 2048$)

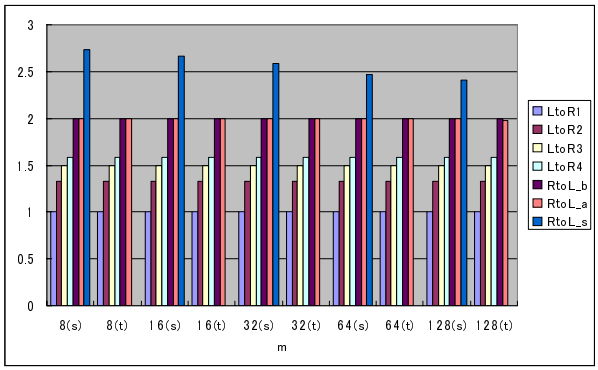


図 60: 各手法による処理速度の変化 ($n = 1024$)

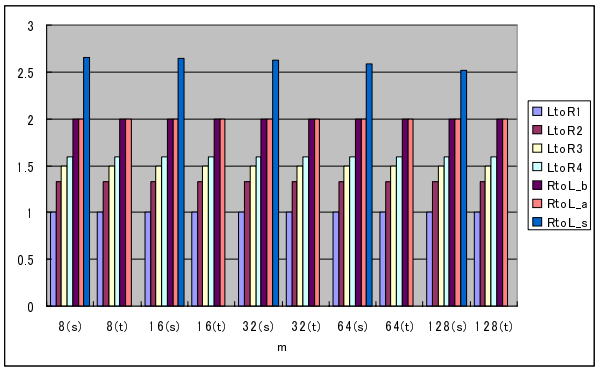


図 62: 各手法による処理速度の変化 ($n = 2048$)

7.2 今後の課題

今後の課題としては以下のことが考えられる。

- サイクル数から考慮した最適な子剰余乗算器の選定

今回の評価で親剰余乗算器から子剰余乗算器を決める際に、あらかじめ用意した剰余乗算器の中から一番適切と思われるものを用いたが、これは必ずしも最適と言えるものではなかった。剰余乗算器に用いる乗算器のビット数を 1 ずつ変化させた場合の 1 回の剰余乗算にかかるサイクル数の変化を調べ、その結果から最適な子剰余乗算器を求めて評価を行うことでさらに回路全体の面積を削減できると考えられる。

- 記憶領域を外部においた場合の面積評価

本研究ではアレイ法による記憶領域の拡大を調べるために回路の中に X^{e_i} や B_{e_i} を格納するためのレジスタを組み込んだ。しかし、実際にハードウェア実装を行う場合は、記憶領域を他の回路と共有するために外部に置くことが多い。本当の意味で本研究で提案した方式の有用性を示すにはこのような実装を行う際の記憶領域を除いた部分での右向きアレイ法と本研究で提案した方式の回路面積の比較を行う必要がある。

参考文献

[1] R. L. Rivest, A. Shamir, and L. Adliman: "A Method for Obtainning Digital Signatures and Public-key Cryptosystems," Communi-

cations of the ACM, 21(2), pp.120-126, Feb. 1978.

[2] “サルにもわかる RSA 暗号,” http://www.maitou.gr.jp/rsa/

[3] A. J. Menezes: “Elliptic Curve Cryptosystems,” Boston:Kluwer Academic, 1993.

[4] “Data Encryption Standard(DES),” http://www.itl.nist.gov/fipspubs/fip46-2.htm.

[5] “Advanced Encryption Standard(AES),” Fedral Information Processing Standards Publication 197, Nov. 2001.

[6] P. L. Montgomery: “Moduler Multiplication Without Trial Division,” Mathematics of Computation, 44(170), pp.519-521, Apr. 1985.

[7] 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.

[8] K. C. Posch and R. Posch: “Modulo Reduction in Residue Number Systems,” IEEE Tr. Parallel and Distrbuted Systems. 6, 5, pp.449-454, 1995.

[9] A. J. Menezes, P. C. Oorschot, and S. A. Vanstone: “HANDBOOK of APPLIED CRYPTOGRAPHY,” CRC Press, 1997.

[10] Knuth, D. E: “The Art of Computer Programming Vol. 2: Seminumerical Algorithms (2nd ed.),” Addison-Wesley, 1981.

[11] Yao, A. C.-C: “On the evaluation of powers,” SIAM Journal on Computing5, pp.100-103, 1976.

[12] 梶原 祐輝, 永田 真, 瀧 和男: “RSA 暗号用高速べき乗剰余演算器の設計,” 電子情報通信学会信学技報 VLD2002-108, pp.157-162, Nov, 2002.

[13] 高木 直史: “初等関数計算回路のアルゴリズム,” 情報処理, vol.37, no.4, pp.362-368, Apr. 1996.

[14] Anand Krishnamurthy, Yiyan Tang, Cathy Xu and Yuke Wang: “An Efficient Implementation of Multi-Prime RSA on DSP Processor,” ICASSP 2003, Jul, 2003.

[15] VLSI Design and Education Center http://www.vdec.u-tokyo.ac.jp/

[16] DesignWare http://www.synopsys.com/