C-016

Coarsely Integrated Operand Scanning アルゴリズムに基づくモンゴメリ乗算器の回路規模縮小手法の検討

A Method for Size Reduction of Montgomery Multiplier Based on Coarsely Integrated Operand Scanning Algorithm

田村 慧 † 山田 親稔 ‡ 市川 周一 §
Satoru Tamura Chikatoshi Yamada Shuichi Ichikawa

1. はじめに

近年，電子商取引や電子政府の推進，IC カードの普及により，高度な情報通信ネットワークが発展している．多様な情報端末がネットワークに接続されるユビキタス社会において，情報の秘匿化や通信の信頼性の保証は重要な課題となっている．この課題の解決法に，RSA 暗号 [1] を代表とする公開鍵暗号 [2] がある．公開鍵暗号の基本演算は剰余乗算計算であり，剰余乗算はモンゴメリ乗算 [3] で高速に計算できる．しかし，RSA 暗号では 512 bit から 2048 bit の鍵を用い，多倍長の剰余乗算計算を数千から数万回行う必要があるため計算量が膨大である．そのため，IC カードなどの計算リソースが限られた小型機器においては，ハードウェア実装が求められる．モンゴメリ乗算のハードウェア実装における先行研究として，基数 2 による構成 [4][5] と高基数による構成 [6][7] がある．基数 2 の構成では，乗算の片方のオペランドをビットごとに展開して扱うため，加算器を用いた単純な構成となる．しかし，オペランド長の大きな加算器を用いるため，加算器の段数が非常に多くなる．そのため，回路規模も大きくなり，配線遅延の増大によりスループットが低下してしまう．一方，高基数の構成では，乗算の片方，または両方のオペランドを任意のワード長で扱う．ワード長を任意に設定することで，要求されるリソース制約に応じて回路規模・処理速度のトレードオフが調整でき，スケーラブルな構成が可能となる．そのため近年では，高基数型モンゴメリ乗算に基づいたスケーラブルな回路構成の研究が多い．

高基数型モンゴメリ乗算アルゴリズムは複数提案されている．先行研究では，FIOS(Finely Integrated Operand Scanning) アルゴリズムに基づいた回路 [6] や CIOS(Coarsely Integrated Operand Scanning) アルゴリズムに基づいた回路 [7] が提案されている．しかし，先行研究 [7] の CIOS アルゴリズムに基づいた回路は，複数の加算器・乗算器から構成されており，回路規模が大きい．そこで本稿では，複数の演算器を一つの演算器に統合し，専用の演算器を設けた回路構成を提案する．専用の演算器を設けることで，統合に伴う内部遅延の増加を抑制しつつ，回路規模を縮小できる．さらに提案するアーキテクチャを FPGA に実装し，他の実装例との比較により機能評価を行う．

†沖縄工業高等専門学校 専攻科創造システム工学専攻
‡沖縄工業高等専門学校 情報通信システム工学コース
§豊橋技術科学大学 電気・電子情報工学系

2.RSA 暗号

RSA 暗号 [1] とは，1978 年に Ron Rivest らによって考案された公開鍵暗号である．従来の暗号方式であった共通鍵暗号では，暗号化・復号化に共通の鍵を用いており，鍵自体も情報として伝達する必要がある．そのため，情報伝達時に鍵自体の盗聴リスクがあり，情報の秘匿化を妨げる要因を内包している．しかし，RSA 暗号に代表される公開鍵暗号では，一対となる秘密鍵と公開鍵を生成し，暗号化を公開鍵で，復号化を秘密鍵で行うため，復号化用の秘密鍵を伝達する必要がなく，情報の高い秘匿化が可能である．RSA 暗号の暗号化式を式 (1) に，復号化式を式 (2) に示す．ここで a は平文，c は暗号文，$\{e, n\}$ は公開鍵，$\{d, n\}$ は秘密鍵である．

$$\textbf{Encryption} : c = a^e \bmod n \tag{1}$$

$$\textbf{Decryption} : a = c^d \bmod n \tag{2}$$

情報通信において，送信者は公開鍵 $\{e, n\}$ と秘密鍵 $\{d, n\}$ を生成し，受信者へ公開鍵 $\{e, n\}$ を公開する必要がある．公開鍵・秘密鍵の生成は次の手順で行われる．

1. 2 つの大きな素数 $\{p, q\}$ を生成し，それらの積 $n = pq$ を求める．
2. $p-1$ と $q-1$ の最小公倍数 (Least Common Multiple; LCM)L を求める．
$$L = \mathrm{LCM}(p-1, q-1) \tag{3}$$
3. L との最大公約数 (Greatest Common Divisor; GCD) が 1 となる公開鍵 e を選択する．
$$\mathrm{GCD}(L, e) = 1 \tag{4}$$
4. $(1 < d < L)$ となる秘密鍵 d を求める．
$$ed \equiv 1 (\mathrm{mod}\ L) \tag{5}$$

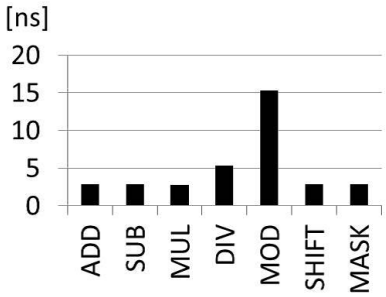
鍵 n の素因数 $\{p, q\}$ を得ることが出来れば，上記の生成手順により秘密鍵を入手することが可能となる．しかし，RSA 暗号では 512 bit から 2048 bit の鍵 n を使用しており，これだけ大きな数を現実的な時間で素因数分解できるアルゴリズム (離散対数問題の多項式時間による解法) は考案されていない．この NP 問題が RSA 暗号の安全性に直結している．

RSA 暗号の処理は剰余乗算計算が基本となるが，減算の繰り返し処理や除算を用いた剰余の計算は膨大な処理時間を要する．そこで，除算を行わずシフト演算やマスク演算などを用いて高速に剰余を求めることができる，モンゴメリ乗算 [3] が広く用いられている．

3. モンゴメリ乗算

3.1. モンゴメリ乗算の原理

モンゴメリ乗算は，1985 年に Montgomery によって考案された，剰余計算を行うためのアルゴリズムである．RSA 暗号では式 (1)，(2) に示すように，べき剰余計算のみで簡単に暗号化・復号化を行える．しかしながら，剰余計算は算術演算や論理演算の中でも最も時間のかかる処理である．各演算のソフトウェア処理時間比較を Fig. 1 に，測定環境を Table. 1 に示す．



*Average Value of Processing 1,000 Loop

Fig. 1: Comparison Software Processing Time of Each Operation

Table. 1: Measurement Environment Software Processing Time of Each Operation

Used Language	C Language
Compiler	Borland C++ Compiler 5.5
CPU	Intel Core 2 Duo T7500 @2.20 Ghz × 2
Memory	DDR2 4 GB

RSA 暗号では多倍長の剰余乗算計算を数千から数万回行う必要があり，剰余計算の高速化は重要な課題である．そのため，乗算・加減算・シフト演算・マスク演算などの比較的高速な演算のみで，剰余を求めることができるモンゴメリ乗算が広く用いられている．モンゴメリ乗算のアルゴリズムを Fig. 2 に示す．

Input : $X = \{x_{k-1},\ ...,\ x_1,\ x_0\}_2$
$Y = \{y_{k-1},\ ...,\ y_1,\ y_0\}_2$
$N = \{n_{k-1},\ ...,\ n_1,\ n_0\}_2$
$R = 2^k$

Output : $Z = XYR^{-1} \bmod N$

Before calculation : $N' = -N^{-1} \bmod R$

Algorithm :

1. $m = (XY \bmod R)N' \bmod R$;
2. $Z = (XY + mN)/R$;
3. if $(k \geq N)$ then $Z = Z - N$;
4. return Z ;

*N and R are relatively prime, **$X, Y < N$

Fig. 2: Montgomery Multiplication

モンゴメリ乗算のアルゴリズムについて説明する．アルゴリズムの 1，2 行目は Fig. 3 に示すように，XY の乗算結果に対し法 N の倍数 (mN) を加算 $(XY + mN)$ することで，$R(=2^k)$ で割り切れる数 $(XY + mN \equiv 0 \pmod R)$ に補正している．

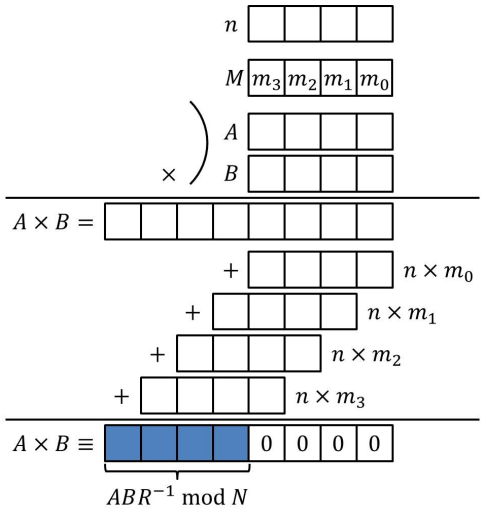


Fig. 3: Schematic Diagram of Montgomery Multiplication

- $XY + mN \equiv 0 \pmod R$

$$\begin{aligned} m &= (XY \bmod R)N' \bmod R \\ &\equiv XYN' \pmod R \\ XY + mN &\equiv XY + XYN'N \pmod R \\ &\equiv XY - XY \pmod R \\ &\therefore\ XY + mN \equiv 0 \pmod R \end{aligned}$$

この補正した数の上位 R bit$(Z = (XY + mN)/R)$ は，XY の剰余乗算結果に R^{-1} をかけた数 (XYR^{-1}) と法 N において合同 $(Z \equiv XYR^{-1} \pmod N)$ である．

- $Z \equiv XYR^{-1} \pmod N$

$$\begin{aligned} Z &= (XY + mN)/R \\ ZR &= XY + mN \\ &\equiv XY \pmod N \\ ZRR' &\equiv XYR' \pmod N \\ &\therefore\ Z \equiv XYR' \pmod N \end{aligned}$$

また，補正した数の上位 R bit$(Z = (XY + mN)/R)$ は，$Z < 2N$ の関係になるため，アルゴリズムの 3 行目で例外処理を行なっている．

- $Z \equiv XYR^{-1} \pmod N$

$$\begin{aligned} XY + mN &< N^2 + mN \\ ZR &= (N + m)N \\ &< (R + R)N \\ ZRR' &= 2RN \\ &\therefore\ Z = (XY + mN)/R < 2N \end{aligned}$$

$R(=2^k)$ は 2 のべき乗であるため，R での剰余計算は k bit マスク演算で置換でき，下位 k bit を取り出す処理となる．同様に R での除算は k bit 右シフト演算となり，上位 k bit を取り出す処理となる．このため，モンゴメリ乗算は乗算・加算・シフト演算・マスク演算のみで剰余乗算を求めることができる．

モンゴメリ乗算は，モンゴメリドメインと呼ばれる集合 $\boldsymbol{M}_D$ の乗算 $\otimes$(式 (6)) として解釈される．そのため，モンゴメリ乗算で剰余計算を行うには，入力 x，y を式 (7) の関数で上へ写像 ($x \to X$，$y \to Y$) して乗算 $Z = XY$ を行い，その結果 Z に対し，式 (8) の関数で逆写像 ($Z \to z$) を行う必要がある．モンゴメリ乗算での剰余計算式を式 (10) に示す．

$$X \otimes Y = XYR' \bmod N \quad (6)$$
$$F(x) = xR \bmod N \quad (7)$$
$$F^{-1}(x) = XR' \bmod N \quad (8)$$
$$= X \otimes 1 \quad (9)$$
$$z = F(x) \otimes F(y) \otimes 1 \quad (10)$$

3.2. 高基数型モンゴメリ乗算

モンゴメリ乗算には，基数 2 で行う方法と高基数で行う方法の 2 種類の方法がある．基数 2 でのモンゴメリ乗算は，乗算の片方のオペランドをビットごとに展開して扱うため，オペランド長は入力の数のビット数に依存する．そのため，多倍長演算を実行できる環境を必要とする．一方，高基数でのモンゴメリ乗算は，乗算の片方，または両方のオペランドを任意のワード長で扱う．ワード長を任意に設定することで，与えられるリソース制約の下での実行が容易になる．このため近年では，高基数型モンゴメリ乗算に基づく，スケーラブルな回路構成の研究が多い．Fig. 4 に高基数型モンゴメリ乗算アルゴリズムを示す．

Input : $X = \{x_{s-1},\ ...,\ x_1,\ x_0\}_{2^w}$
$Y = \{y_{s-1},\ ...,\ y_1,\ y_0\}_{2^w}$
$N = \{n_{s-1},\ ...,\ n_1,\ n_0\}_{2^w}$
$R = 2^w$
$k(= sw)$: Input number of bits
Output : $Z = XY2^{-s \cdot w} \bmod N$
Before calculation : $N' = -N^{-1} \bmod R$
Algorithm :
1. $Z = 0$;
2. for $i = 0$ to $s - 1$;
3. $C = 0$;
4. $z_i = (z_0 + x_i y_0) N' \bmod R$;
5. for $j = 0$ to $s - 1$;
6. $Q = z_j + x_i y_j + t_i n_j + C$;
7. if $(j \neq 0)$ then $z_{j-1} = Q \bmod R$;
8. $C = Q/R$;
9. $z_{s-1} = C$;
9. if $(Z > N)$ then $Z = Z - N$;

Fig. 4: High-Radix Montgomery Multiplicatione

高基数型モンゴメリ乗算の計算アルゴリズムは複数考案されているが，本研究では，Cetin Kaya Koc らによる分析結果 [8] において，最も処理時間が短いとされた CIOS アルゴリズムを取り扱う．Fig. 5 に CIOS アルゴリズムを示す．CIOS アルゴリズムは，高基数型モンゴメリ乗算 (Fig. 4) アルゴリズムの積和演算 ($Q = z_j + x_i y_j + t_i n_j + C$) を 2 つの内部ループに分割 (内部ループ 1 : $(C, S) = z_j + x_j y_i + C$，内部ループ 2 : $(C, S) = z_j + m n_j + C$) して処理している．積和演算 1 回あたりの項数を少なくすることで，積和演算器の回路規模を抑制することが可能となる．

Input : $X = \{x_{s-1},\ ...,\ x_1,\ x_0\}_{2^w}$
$Y = \{y_{s-1},\ ...,\ y_1,\ y_0\}_{2^w}$
$N = \{n_{s-1},\ ...,\ n_1,\ n_0\}_{2^w}$
$R = 2^k$
$W = 2^w$
$k(= sw)$: Input number of bits
Output : $Z = XYR^{-1} \bmod N$
Before calculation : ${n_0}' = -{n_0}^{-1} \bmod R$
Algorithm :
1. for $i = 0$ to $s - 1$;
2. $C = 0$;
3. for $j = 0$ to $s - 1$;
4. $(C, S) = z_j + x_j * y_i + C$;
5. $z_j = S$;
6. $(C, S) = z_s + C$;
7. $z_s = S$;
8. $z_{s+1} = C$;
9. $C = 0$;
10. $m = z_0 * {n_0}' \bmod W$;
11. $(C, S) = z_0 + m * n_0$;
12. for $j = 1$ to $s - 1$;
13. $(C, S) = z_j + m * n_j + C$;
14. $z_{j-1} = S$;
15. $(C, S) = z_s + C$;
16. $z_{s-1} = S$;
17. $z_s = z_{s+1} + C$;
18. if $(Z \geq N)$ then $Z = Z - N$;
19. return Z ;

Fig. 5: CIOS Algorithm

4. モンゴメリ乗算のハードウェア実装

4.1. 回路アーキテクチャ

先行研究 [7] の CIOS アルゴリズムに基づいた回路は，複数の加算器・乗算器から構成されている．演算器を複数設けることで，積和演算や加算の並列処理が可能になるが，回路規模が増大する．そこで本稿では，複数の演算器を一つの演算器に統合し，専用の演算器 (MAC ユニットおよび MM ユニット) を設けた回路アーキテクチャを提案する．提案する回路アーキテクチャを Fig. 6 に，提案手法と先行研究の手法との比較を Fig. 7 に示す．提案手法では専用の演算器を設けることで，統合に伴う内部遅延の増加を抑制しつつ，回路規模を縮

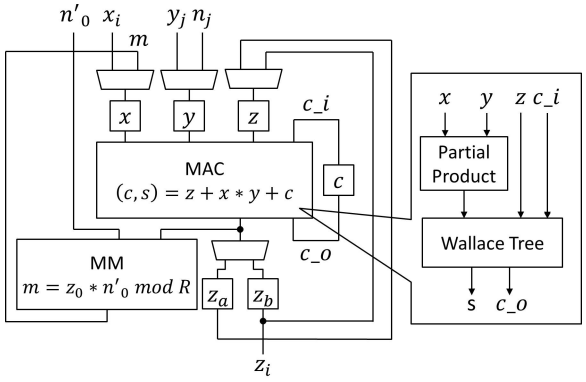


Fig. 6: CIOS Architecture

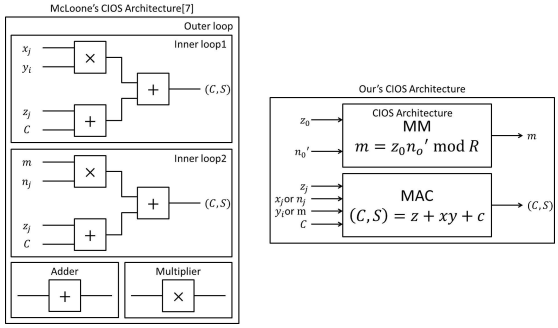


Fig. 7: Calculator Comparison of Internal Circuit of Our's and McLoone's Method[7]

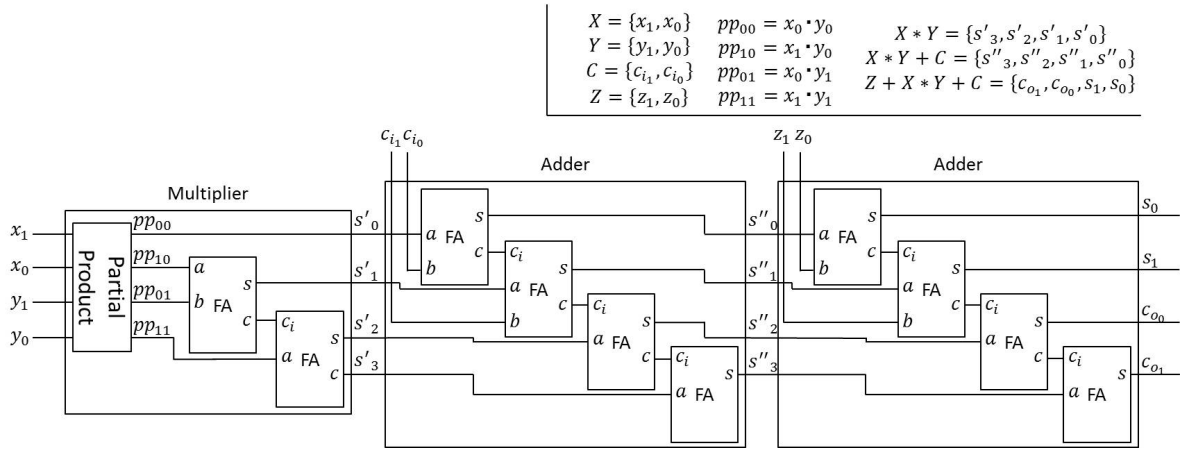


Fig. 8: Series Connection of The Adder and Multiplier ($w = 2$ bit)

小できる．

提案するアーキテクチャでは，クリティカルパスとなる積和演算の演算器が回路全体へ与える影響を考慮し，Wallace Tree[9] を用いた積和演算器 (MAC ユニット) を設けた．Wallace Tree は，乗算器を高速化する手法の 1 つで，部分積の加算において桁が同じなら入力項，和，キャリーなどの加算の順番は考えなくてもよい事から，次の 3 つのルールに則り加算の段数を減らすことができる．

1. 同じ桁にビット項が 3 項以上あれば，FA(Full-Adder) を適用して加算する．
2. 加算可能な最下位の桁にビット項が 2 項あれば，HA(Half-Adder) を適用して加算する．
3. すべての桁のビット項が 3 以下になったときは，ビット項が 2 項の桁が加算可能な最下位桁でなくとも HA を適用する．

CIOS アルゴリズム内での演算処理は，基本的にこの MAC ユニットで行う．例外として，Fig. 4 の CIOS アルゴリズム 10 行目の剰余乗算

$$10. \quad m = z_0 * {n_0}' \bmod W$$

は，後述する MM ユニットで処理する．また，z_j の入出力処理において，専用メモリを用意するとアドレス制御が必要となり，制御時間分の遅延が余分に発生する．そのため，2 個のシフトレジスタを FIFO(First In, First Out) バッファとして使用し，入出力をキューとして処理することで，アドレス制御を不要としている．

4.1.1.MAC（Multiply and AC cumulation）ユニット

MAC ユニットは 4 項からなる積和演算を行うユニットで，Fig. 5 の CIOS アルゴリズムにおける 4，6，11，13，15，17 行目の処理

$$\begin{aligned}
4. &\quad (C,S) = z_j + x_j * y_i + C \\
6. &\quad (C,S) = z_s + C \\
11. &\quad (C,S) = z_0 + m * n_0 \\
13. &\quad (C,S) = z_j + m * n_j + C \\
15. &\quad (C,S) = z_s + C \\
17. &\quad z_s = z_{s+1} + C
\end{aligned}$$

を行う．これら処理では最大で加算 2 回と乗算 1 回の計算を行う．しかし，Fig. 8 のように加算器，乗算器

を直列に接続すると FA アレイの段数が多く，最下位ビットから最上位ビットへのキャリー伝搬遅延が大きい．そこで，同じ桁同士をまとめ，並列に加算を行う Wallace Tree に基づき専用の積和演算器である MAC ユニットを使用することで，加算段数を減らし，全体の遅延時間を短縮している．また，別途に加算器を設けることによる回路規模の増加を防ぐため，通常の加算も同ユニットで処理している．Fig. 9 は，Fig. 8 の演算器に Wallace Tree を適用した結果を示している．

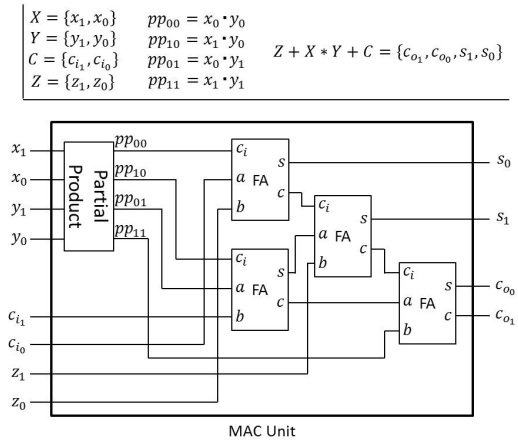


Fig. 9: MAC Unit ($w = 2$ bit)

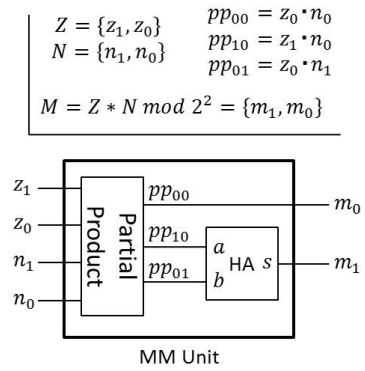


Fig. 10: MM Unit ($w = 2$ bit)

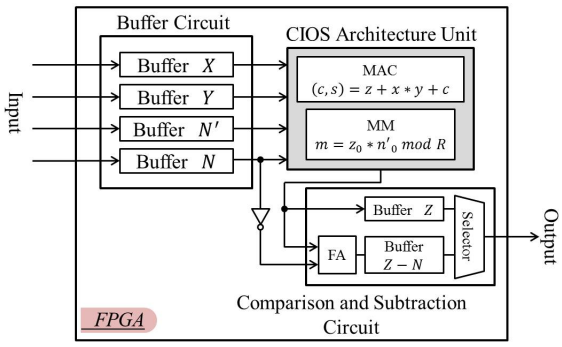


Fig. 11: System Configuration

4.1.2.MM（Multiply and Modulo）ユニット

MM ユニットは Fig. 5 の CIOS アルゴリズムにおける 10 行目の剰余乗算

$$10. \quad m = z_0 * n_0{}' \bmod W$$

を行うユニットである．$W(= 2^w)$ が 2 のべき乗であり，剰余計算がマスク演算で計算可能であることから，本処理の計算結果の上位 w bit はマスク演算により消失する．この特徴を利用し本ユニットでは，上位 w bit 分の乗算を行わず，下位 w bit 分のみの乗算を Wallace Tree を用い計算している．そのため通常の乗算の半分の計算量での乗算・剰余計算が可能である．Fig. 10 は $w = 2$ bit での MM ユニットを示している．

4.2. システム構成

提案する CIOS アーキテクチャを用い，高基数型モンゴメリ乗算器の設計を行った．システム構成を Fig. 11 に示す．

本構成の全ての入出力は w bit 単位で行う．バッファ回路では回路全体の処理終了まで入力を保持し，CIOS アーキテクチャユニット，および比較減算回路への出力を内部カウンタにより制御する．CIOS アーキテクチャユニットでは積和演算と乗算・剰余計算を繰り返し，Fig. 2 のモンゴメリ乗算における 1，2 行目の処理

$$1. \quad m = (XY \bmod R)N' \bmod R$$
$$2. \quad Z = (XY + mN)/R$$

を行う．CIOS アーキテクチャ内部のセレクタは，それぞれ専用の制御用パルスジェネレータによって制御される．その後，比較減算回路にて，2 の補数表現で加算による減算を行い，Fig. 2 のモンゴメリ乗算における 3 行目の処理

$$3. \quad if(k \geq N) then Z = Z - N$$

を行う．

4.3. システム評価

Fig. 11 のシステムを基数 2(ワード長 $w = 2$ bit)，および 2^{32}(ワード長 $w = 32$ bit) の 2 つの場合にて設計した．開発環境を Table. 2 に示す．また，回路の論理合成結果を Table. 3 に，先行研究 [7] との比較を Table. 4 に示す．Table. 3 の () 内の数字は資源の利用率 (Used/Available*100 %) を表している．

Table. 2: System Development Environment

開発ツール	Xilinx ISE Design Suite 13.3
FPGA	Spartan-6 XC6SLX45T
言語	Verilog-HDL

資源の利用率は，基数 2^{32} で設計した場合レジスタ：0.84%，LUT：10.56% となり，より高い基数で設計しても十分実装可能な回路規模である．また，提案手法では処理時間が 0.797 μs となり，先行研究 [7] の処理時間 0.6 μs と比較して，約 1.33 倍に増加した．しか

Table. 3: Logic Utilization of Our's Method

	ワード長 32 bit	ワード長 2 bit	Available
Occupied Slice	695 (10.19%)	146 (2.14%)	6822
Slice Registers	563 (1.03%)	461 (0.84%)	54576
Slice LUTs	2882 (10.56%)	685 (2.51%)	27288
Operating Frequency	40.1 MHz	213.022 MHz	
Processing time	0.797 μs	41.47 μs	

Table. 4: Performance Comparison of Montgomery Multipliers

	Architecture	
	Our's	McLoone's Method[7]
Area	695 slices	1522 slices 11 multipliers
Operating frequency	40.1 MHz	70.7 MHz
Processing time	0.797 μs	0.6 μs
Radix	$2^2(w=2)$	$2^4(w=4)$
FPGA	XC6SLX45T (1 slice = four 6-input LUT + eight FF)	XC2VP50 (1 slice = one 4-input LUT + one FF)

し，回路規模としては提案手法が 695 slices，先行研究 [7] が 1522 slices + 11 multipliers と，スライス数だけ考えた場合回路規模は約 55% 縮小している．先行研究 [7] では FPGA 内部の乗算器も使用していることを考慮すれば，回路規模は更に小さくなっていると考えられる．これらより，提案手法は回路規模の縮小化において有効であることを確認した．

5. まとめ

本稿では，IC カードなどの小型暗号機器向けに，高速かつ小型な剰余乗算回路を設計することを目的とする．先行研究では，剰余を高速に求めることのできる高基数型モンゴメリ乗算に基づいた回路構成が提案されていたが，複数の演算器が必要であり，回路規模が大きい．そこで本研究では，複数の演算器を乗算器の高速化手法である Wallace Tree を用いて 1 つの積和演算器に統合することで，回路規模の縮小化を図った．結果，提案するアーキテクチャは処理速度が約 1.33 倍に増加するものの，回路規模は約 55% 縮小し，回路規模縮小手法として有効であることを確認した．

謝辞

本研究は JSPS 科研費 25871048 の助成を受けたものである．

参考文献

[1] R.L. Rivest, A. Shamir & L. Adleman: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, Vol.21, No.2 (1978)

[2] Whitfield Diffie & Martin E. Hellman: New Directions in Cryptography, IEEE TRANSACTION ON INFORMATION THEORY, Vol.IT-22, No.6 (1976)

[3] Pater L. Montgomery: Modular Multiplication Without Trial Division, MATHEMATICS OF COMPUTATION, Vol.44 (1985)

[4] A.F. Tenca & C.K. Koc: A scalable architecture for modular multiplication based on Montgomery's algorithm, IEEE Trans. Comput., vol52, No.9 (2000)

[5] D. Harris, R Krishnamurthy, S. Mathew & S. Hsu.: An improved unified scalable radix-2 Montgomery multiplier, In ARITH'05:Proceedings of the 17th IEEE Symposium on Computer (2005)

[6] 馬場裕一，宮本篤志，本間尚文，青木孝文，佐藤証：「RSA 暗号プロセッサジェネレータの設計と評価」，情報科学技術フォーラム講演論文集，Vol.8(1) (2009)

[7] Maire McLoone, Ciaran Mclvor & John V McCanny: Coarsely Integrated Operand Scanning (CIOS) Architecture For High-Speed Montgomery Modular Multiplication, 2004 IEEE International Conference on Field-Programmabled Technology (2004)

[8] Cetin Kaya Koc, Tolga Acar & Burton S. Kaliski Jr.: Analyzing and Comparing Montgomery Multiplication Algorithms, IEEE Micro, 16(3) (1996)

[9] 鈴木昌治：「ディジタル数値演算回路の実用設計」，CQ 出版社（2006）