高速剰余算アルゴリズムとその実装についての研究

首都大学東京　理工学研究科　数理情報学専攻　山本　聡

平成 20 年 3 月 3 日

目次

図目次

表目次

1 序論

1.1 研究の背景

社会の情報化に伴い重要な情報もネットワーク上でやりとりされるようになり、情報に暗号化を施してからの情報通信が必要不可欠なものになった。現在使われている暗号技術の代表的なものとして、RSA 暗号やエルガマル暗号などが挙げられる。これらの暗号は暗号解読に必要な鍵を情報として公開していることから、公開鍵暗号と呼ばれている。

公開鍵暗号の多くは大きな整数（1024 ビット以上、10 進数で 308 桁以上）の剰余算演算を繰り返すことで計算を行う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、汎用コンピュータ上でのソフトウェアにおいても高速に計算を行う研究もなされているが、現在でもソフトウェアではハードウェア程のパフォーマンスを得ることができていない [10]。そこで暗号に必要な計算をより高速に行うことができる暗号処理専用のハードウェアの開発が必要になる。

現在、暗号システムに用いられる計算を高速に解くことのできる専用のハードウェアについての研究は頻繁に行われ、実際に暗号の安全性を評価することに用いられている [8]。ハードウェアによる実装の中でも書き換えが可能である FPGA（Field Programmable Gate Array）を用いて同程度の速度で計算を行うことができるとすれば、専用の回路を製作するよりもメリットが大きいといえる。高度成長を続ける情報化社会において、暗号の強度の信頼性はいつ解読されるかわからないという弱点を持ち合わせている。そのためその時々で安全性が保証できる鍵長や、新たな暗号に合わせてハードウェアを再構成できる FPGA は、暗号の研究に向いたデバイスといえるからである。

そこで本研究では FPGA を用いて公開鍵暗号などで用いられる剰余演算を高速に行うアルゴリズムについて実装しどのような工夫を施せば高速に構成できるかを示すことを目標にした。この研究を行うことで暗号分野の発展につながると考え本研究に着手した。

1.2 論文の概要と流れ

本論分ではまず研究を行った開発環境の解説を行い、続いて公開鍵暗号のシステム、剰余算の計算を高速に行うアルゴリズムについて言及する。さらにそれらのアルゴリズムをハードウェア上で実装し、その動作速度についての検証を行う。

2　開発環境について

この章では本研究で仕様したハードウェアである FPGA と、そこで用いたハードウェア記述言語（HDL:Hardware Discription Language）である Verilog について解説をしていく。

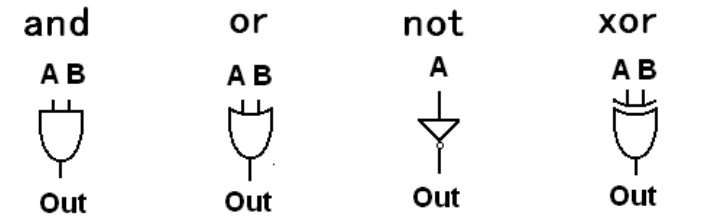
2.1　Verilog

Verilog とは、ゲートウェイ・デザイン・オートメーション社（現ケイデンス・デザイン・システムズ http://www.cadence.com/）により 1984 年に開発されたハードウェア記述言語、およびそのシミュレータの総称である。米国国防総省が開発した VHDL（Very High-Speed Integrated Circuit Hardware Description Language）に対抗して開発された。

エンジニアに受け入れられるよう C や Pascal の文法を取り入れて作られており、回路の入出力を実際に書くゲートレベルでの記述の他に、if 文での条件分岐や for 文での繰り返し文などの使用が可能なビヘイビア記述という方法が用意されており自由度の高い記述が可能となっている。

2.2　組み合わせ回路

ハードウェア記述言語で用いられる、現在の入力値のみで出力が決まる回路を組み合わせ回路と呼ぶ。組み合わせ回路には図 1 で示すものがある。



A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

A	Out
0	1
1	0

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

図 1　組み合わせ回路と真理値表

これらの回路を組み合わせて回路を設計できる。

2.3　ハードウェア記述言語における DFF

図 2 に DFF(D-FlipFlop と呼ばれる信号を 1 クロック遅延させる回路) の回路図と、ゲートレベル、ビヘイビアレベルそれぞれの Verilog での記述を載せる。

なお回路中の nand 回路の nand は not and の略であり、and 回路の結果をさらに反転させたものである。

【DFF のゲートレベル記述】

```
module dff_gate(q,qb,clk,d,rst);
  input clk,d,rst ;
  output q,qb ;

  nand n1(cf,dl,cbf), n2(cbf,clk,cf,rst),
       n3(dl,d,dbl,rst), n4(dbl,dl,clk,cbf),
       n5(q,cbf,qb), n6(qb,dbl,q,rst) ;
  endmodule
```

【DFF のビヘイビアレベル記述】

```
module dff_behavioral(q,qb,clk,d,rst) ;
  input clk,d,rst ;
  output q,qb ;

  always @(posedge clk)
  begin
     if(rst == 0)
     begin
        q = 0;
        qb = 1;
     end
     else
     begin
        q = d;
        qb =  d;
     end
  end
```

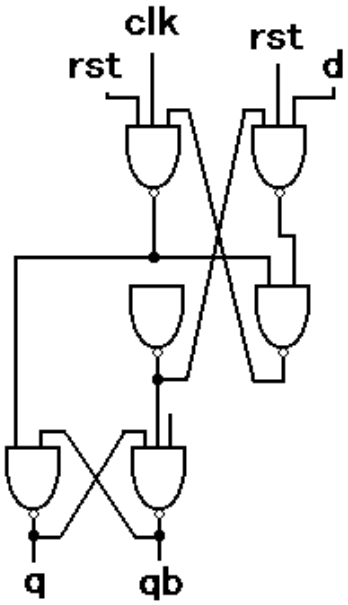


図 2　dff の回路図

2.4 FPGA

FPGA とは正式名称を Field Programmable Gate Array という、利用者が独自の論理回路を即座に実装できるハードウェアである PLD（Programmable Logic Device）の一種で、書き換えが可能な IC のことである。1985 年に米国ザイリンクス社（http://www.xilinx.com/）によって初めて製品化された。

本節では FPGA を用いた開発工程について述べる。

2.4.1 設計

設計は汎用コンピュータ上のソフトウェアで行う。その記述方法には HDL を用いる方法、C などのソフトウェア言語を用いる方法などがあるが、本研究では現在最も FPGA 設計に用いる手法である、HDL による記述を行った。HDL で記述した回路を各ベンダから提供されているソフトウェア（本研究では米国ザイリンクス社の ISE というソフト）を用いて論理合成する。

2.4.2 検証

検証段階では、設計段階で出来上がった回路の振る舞いをシミュレータを用いて検証する。HDL の記述のみでシミュレーションを行う場合と、この次の工程の配置配線を施し、ハードウェアで起こりうる遅延情報を踏まえて行うシミュレーションがある。このシミュレーションには ModelSim ザイリンクスエディションというソフトを用いた。

図 3 は ModelSim を用いての DFF 回路の出力信号を記録したものである。時間とともに様々な信号が 0,1 で変化していることがわかる。

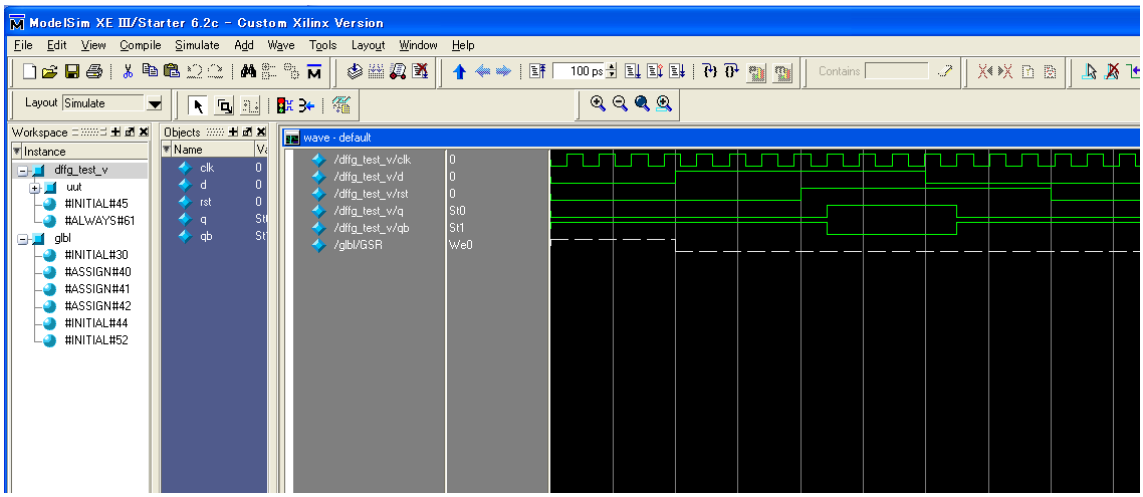


図 3　ModelSim での DFF の波形

2.4.3 配置配線

設計と検証を繰り返し、回路の仕様を固めた段階で、実際に FPGA に書き込むための回路データを作成する工程で、設計段階で得られた論理合成の結果から、FPGA 内部の素子への回路の割り当てとそれらの配置・配線を決定する。この工程も FPGA ベンダから提供されているソフトウェアを用いる。

2.4.4　書き込み（コンフィギュレーション）

配置配線を行った回路データを実際に FPGA に書き込む工程。FPGA ベンダから提供されているソフトウェアとダウンロードケーブルを用いて FPGA に流し込む。

以上のような開発工程を経て、FPGA 上で回路の実現が可能になる。

今回の研究では virtex の xc4vfx12 というモデルを使用する。Virtex 社から公表されているスペックは表 1 のようになる。今回は表の一番上に記述されているモデルを使用した。

デバイス	CLB アレイ：行 x 列	スライス数	LUT 数	最大分散 RAM またはシフト レジスタ (Kb)	フリップフロップ数
XC4VFX12	64 x 24	5,472	10,944	86	10,944
XC4VFX20	64 x 36	8,544	17,088	134	17,088
XC4VFX40	96 x 52	18,624	37,248	291	37,248
XC4VFX60	128 x 52	25,280	50,560	395	50,560
XC4VFX100	160 x 68	42,176	84,352	659	84,352
XC4VFX140	192 x 84	63,168	126,336	987	126,336

表 1　FPGA スペック表

各用語の説明はザイリンクスのホームページを参照のこと

(http://japan.xilinx.com/support/documentation/user_guides/j_ug070.pdf)

なお本研究で用いた FPGA には、データ検出用としてピンが用意されている。本研究ではピンから出力される波形を Logic Analyzer という装置を用いて計測した。

図 4 は DFF の出力波形を Logic Analyzer で観測した画面である。実際の回路からの出力信号を複数同時に観測し、互いのタイミングを調べることができる。

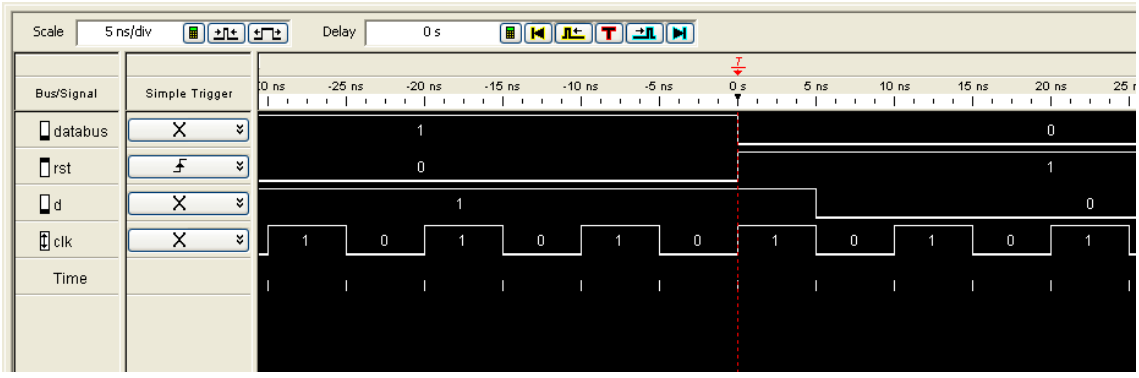


図 4　Logic Analyzer での DFF の波形図

また、本研究に用いた ISE というソフトでは IP（Intellectual property）コアと呼ばれる部分的な回路情報を提供している。IP コアには基本的な演算回路やメモリなど、様々なものが用意されており、入力のビット幅や動作の仕様など使用に則した形にカスタマイズができる状態で用意してある。

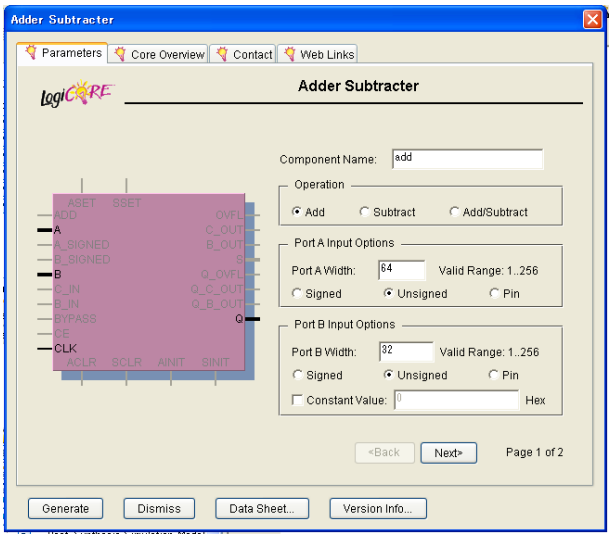


図 5　IP カスタマイズ画面（加算回路）

図 5 は加算回路の IP を選択した画面である。入出力のビット幅などを変更することができるため、構築する回路に合わせて使うことができる。

3　公開鍵暗号

3.1　公開鍵暗号の原理

暗号技術は一般に、共通鍵暗号技術と公開鍵暗号技術に分類される。共通鍵暗号技術は、特殊な「鍵」を使用してデータを暗号化するが、暗号化に使用する鍵と復号化に使用する鍵が同じであることから、データの送受信者の双方が、何らかの手段で安全に鍵を共有する必要がある。この鍵共有問題を解決したのが公開鍵暗号技術である。

公開鍵暗号技術を図 6 とともに解説していく。公開鍵暗号とは

1. 送信者が暗号化に必要な鍵 (暗号化鍵) とデータの復元に使用する鍵 (復号化鍵) を生成する。
2. 公開鍵を受信者に送信する。
3. 送信したいデータを、暗号化の鍵を用いて暗号文にする。
4. 暗号化された情報を受信者に送信する。
5. 受信した暗号文を受信した公開鍵を用いて平文に復号する。

というステップになっている。暗号化と複合化に用いる鍵が異なっており、データの受信者が復号化鍵を秘密にしておけば、暗号化鍵を公開しても安全性を確保できるという暗号方式である。

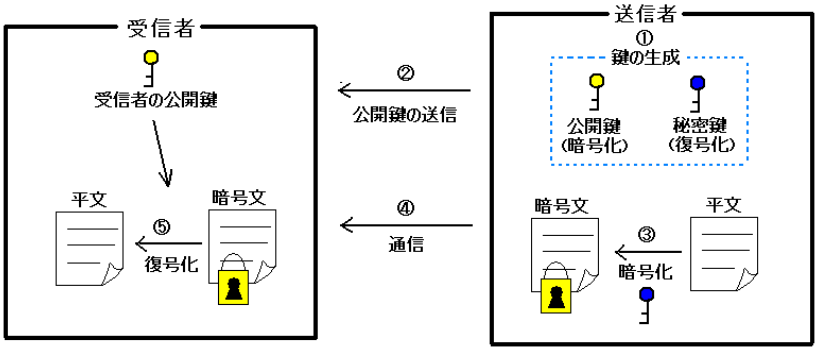


図 6　公開鍵暗号

公開鍵暗号では盗聴者が暗号化された情報と鍵を手に入れることができても、計算量的に複合化を行っての盗聴が事実上不可能とされている [5]。暗号に用いる鍵を盗聴されても安全性が保障されるため、公開鍵暗号は暗号の分野で幅広く使われている [10]。

しかし、暗号化および復号化に使用する鍵が異なることから、公開鍵暗号方式はその暗号・復号化プロセスが複雑であり、共通鍵暗号方式に比べると処理速度が遅いという問題がある。大容量のデータの暗号化は共通鍵暗号で行い、その暗号化鍵 (＝復号化鍵) を公開鍵暗号で暗号化して送信する方法が一般的である [12]。

公開鍵暗号の代表的なものに RSA 暗号や Differ-Hellman などがある。それらの暗号にはビット幅の大き

な数（1024 ビット以上、308 桁以上の数）のべき乗した後の剰余算の計算を多数回繰り返すものが用いられている [7]。

ハードウェア記述言語におけるべき乗剰余算の計算には、バイナリ法を用いての乗算剰余算の繰り返しが用いられている。つまり乗算剰余算の高速化を行うことができれば、情報の暗号化・複合化の高速化を図ることができるということになる。

4　HDL での回路設計

この章では「なぜ単純な剰余算を回路設計で求めるのか」を解説していく。

汎用コンピュータ上のソフトウェアでの演算は、全てハードウェア上で実行されることになる。ソフトウェアでのプログラミングの実行は CPU 内に用意されている命令だけで実行される。そのため、ソフトウェア上でどんなに効率の良いプログラミングを記述したとしても、CPU 上の冗長的な動作しかすることができない [3] [10]。

また汎用コンピュータ上でのソフトウェアでのプログラムでは、同じブロックに記述されたプログラムは逐次実行される。その点ハードウェアでの実行では、同じブロックの動作は並列に処理されるという特徴がある [1] [6]。

ハードウェアでの回路設計ならば、計算する数値のビット幅などを自分で決められることや計算に特化したアルゴリズムを考えながら回路設計することができることから、ソフトよりも高速にビット幅の大きな値を出力する回路を合理的に設計できるということになる。

実際に、最適化されたアルゴリズムを HDL を用いて記述することで、汎用コンピュータ上でのソフトウェア実装よりも高速に計算できるという結果を得られている [14]。

しかしハードウェア記述言語を搭載する FPGA には容量があり、それを超えないように回路を組まなければならない。つまり回路を組む際のメリット、デメリットとしては以下のようなことがいえる。

メリット：専用の回路を構成することができるので、細分化などを施すことで高速化が図れる可能性がある。
　　ソフトでは実現が難しい並列化を用いることができる。
デメリット：FPGA の回路容量によって、記述できる回路数に制限がある

しかし FPGA に記述できる回路容量も年々巨大化している。

4.1　HDL における加算回路

この節では例として、加算回路に用いられているアルゴリズム [6] を実装し、その動作を評価し HDL における回路設計の例を見ていく。ここでは 4 ビット同士の加算モジュールを作成するとする。信号としては A、B を各 4 ビットの入力、4 ビットの和 S、桁上がりとして 1 ビットの Carry を出力するとする。

2.2 でに述べたようにハードウェアで用いられる論理回路はすべて AND、OR、NOT ゲートから構成される（XOR 回路は and と or ゲートの組み合わせで構成されている）。これらを用いて加算回路を構成していく。

なお、論理式で使用される $*$、$+$、$\oplus$ の記号はそれぞれ and、or、xor 回路での計算を表し、各信号の添え字は、その信号におけるビットを表す。

4.1.1　順次桁上げ加算回路

この方法は、二進数同士の加算でおこる桁上げを伝播させ和を求める基本的なアルゴリズムである [14]。

この回路に用いられている hA と FA はそれぞれ HalfAdder、Full Adder と呼ばれる 1 ビットの加算回路である。hA（図 7、表 2）が 2 つの入力値から、FA（図 8、表 3）が 3 つの入力値から、加算結果（Sum）と伝播（Carry）を出力するという回路である。

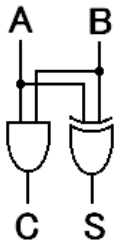


図 7　ha 回路図

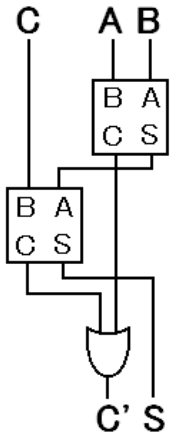


図 8　FA 回路図

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

表 2　ha 真理値表

A	B	C	C′	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

表 3　FA 真理値表

これらを各ビットに組み合わせたものが順次桁上げ加算である。
順次桁上げ加算回路のアルゴリズムは

$$C_0 = A_0 * B_0 + C_{-1} * (A_0 \oplus B_0)$$
$$C_1 = A_1 * B_1 + C_0 * (A_1 \oplus B_1)$$
$$C_2 = A_2 * B_2 + C_1 * (A_2 \oplus B_2)$$
$$C_3 = A_3 * B_3 + C_2 * (A_3 \oplus B_3)$$

のようになる。

順次桁上げ加算回路は、先の hA と FA を用いて図 9 のように書くことができる。

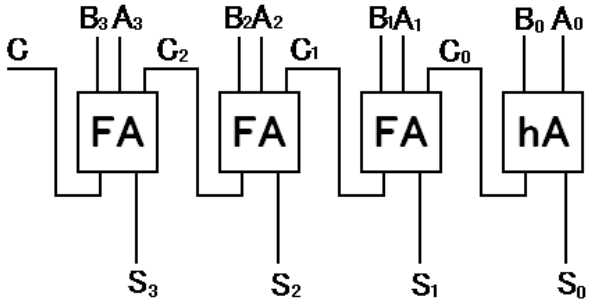


図 9　順次桁上げ加算回路図

この回路は容量としては FA という単純な回路を組み合わせているだけなので、FPGA の資源をそれほど使わずに実装が可能である。しかし上位ビットの計算は下位ビットの計算結果を待たなくてはならないため、桁上がりによる遅延が発生する。これはビット幅が大きくなればなるほど桁上がりによる遅延時間が大きくなってしまう。

4.1.2　桁上げ先見回路（Carry Lookahead Adder:CLA）

CLA アルゴリズムは下位ビットの値をあらかじめ考慮して上位ビットにあらかじめ伝達させておくことで桁上げの時間をなくすアルゴリズムである [14]。

先の順次桁上げ加算回路のアルゴリズムを考え直す。

先の計算式で C_0 を求める際の C_{-1}は 0 であり、また各式の $A_n * B_n$を G_n、$A_n \oplus B_n$を Q_n とおくと、それぞれの桁上げはそこまでの入力を考慮することですべて表すことができるので、それぞれ以下の式が成り立つ。

$$
\begin{aligned}
C_1 &= G_1 + (G_0 + C_{-1} * Q_0) * Q_1 \\
&= G_1 + G_0 * Q_1 + C_{-1} * Q_0 * Q_1 \\
C_2 &= G_2 + (G_1 + G_0 * Q_1 + C_{-1} * Q_0 * Q_1) * Q_2 \\
&= G_2 + G_1 * Q_2 + G_0 * Q_1 * Q_2 + C_{-1} * Q_0 * Q_1 * Q_2 \\
C_3 &= G_3 + (G_2 + G_1 * Q_2 + G_0 * Q_1 * Q_2 + C_{-1} * Q_0 * Q_1 * Q_2) * Q_3 \\
&= G_3 + G_2 * Q_3 + G_1 * Q_2 * Q_3 + G_0 * Q_1 * Q_2 * Q_3 + C_{-1} * Q_0 * Q_1 * Q_2 * Q_3
\end{aligned}
$$

この計算式に則した回路を図 10 に示す。

この回路は桁上げの伝播時間を短縮できるが、その分回路が複雑になる。これは入力ビットを大きくするほど大きくなってしまうため、普通 4 ビットの CLA を複数個結合させるなどの工夫をすることで、高速化と縮小化のバランスを取って用いられている [14]。

4.1.3　冗長二進数を用いた加算回路

冗長二進数とは、各ビットに符号ビットを入れる（各ビットを二ビットにする）ことで数値の表示方法を変え、伝播の時間をなくすアルゴリズムである [3] [10]。

通常、二進数の n ビット目は 2^{n-1} の数を表している。そのため二進数の 0111 は

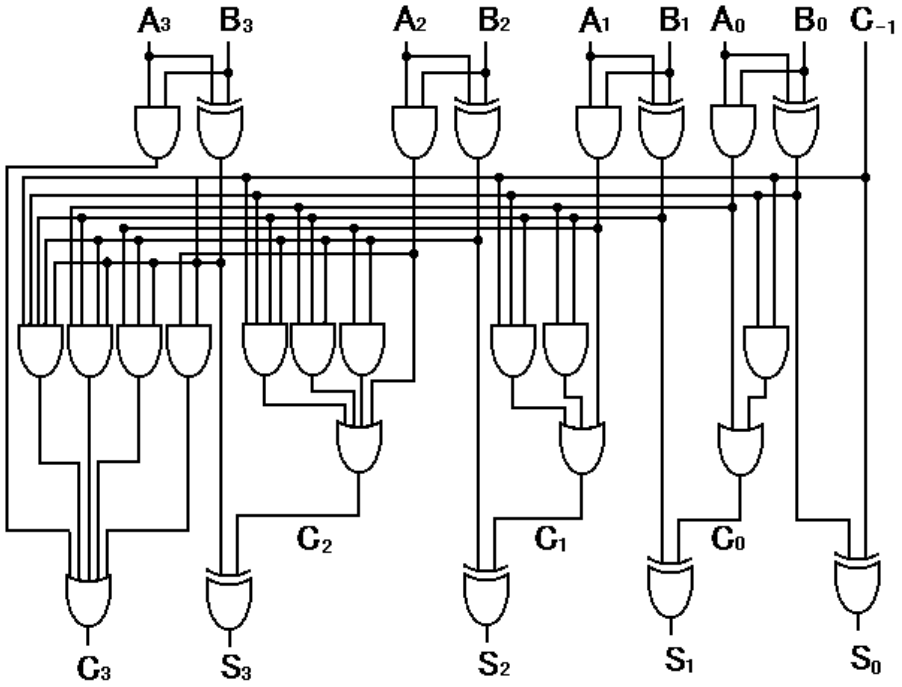


図 10　CLA 回路図

$$2^3 * 0 + 2^2 * 1 + 2^1 * 1 + 2^0 * 1 = 0 + 4 + 2 + 1 = 7$$

と一意に表すことができる。

冗長二進数は各ビットに符号ビットを付加させ、00 を 0、01 を 1、10 を-1 として計算することで数を表現する。すると先の数 7 は例えば $8 = 7 - 1$ と表現することができるので、7 は冗長二進数においては

01 00 00 10

と表すことができる。

この状態で 7+1 を考えてみると、図で示したように加算の桁上がりの伝播時間が省略できることがわかる。

二進数での表現が一意に決まるのに対し、冗長二進数では数を一意に決めることができない。例えば 7 は $8 - 4 + 2 + 1$ として 01 10 01 01 とも表すことができる。

この冗長性を取り入れることで、桁上がりが起こらないような数で表現することで、桁上がりの伝播時間をなくすことができる。

冗長二進数のアルゴリズムは 2 つのステップで行う。

i ビット目を考えるとすると、第一ステップでは

$$x_i + y_i = 2 * c_{i+1} + d_i$$

を満たすように、中間桁上げ c_{i+1}と中間和 d_i を求める。

その値は一桁前の x_{i-1}, y_{i-1} のインプットによって値が決められる。（表 4 参照）

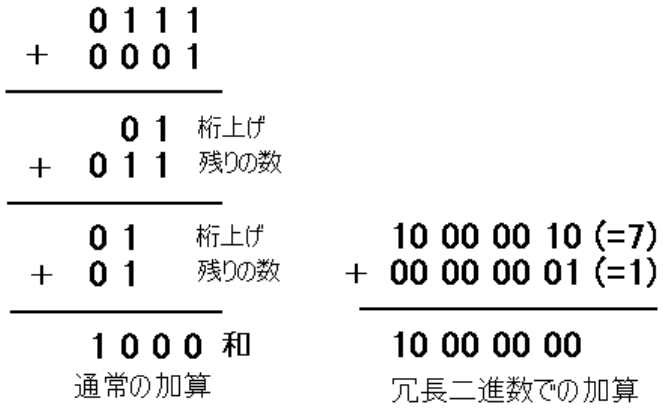


図 11　冗長二進数で桁上がりが無くなる例

c_{i+1} , d_i

$x_i \backslash y_i$	$\bar{1}$	0	1
$\bar{1}$	$\bar{1}$,0	*0,$\bar{1}$/$\bar{1}$,1	0,0
0	*0,$\bar{1}$/$\bar{1}$,1	0,0	*1,$\bar{1}$/0,1
1	0,0	*1,$\bar{1}$/0,1	1,0

*: x_{i-1},y_{i-1} 両方とも非負 / 少なくとも一方が負

表 4　ステップ 1 の真理値表

第二ステップでは中間和 d_iとひとつ下位の桁からの中間桁上げ c_iを加え合わせ、和 s_iを得るというステップをとる。加算結果の真理値表は表 5 のように表される。

s_i

$d_i \backslash c_i$	$\bar{1}$	0	1
$\bar{1}$	×	$\bar{1}$	0
0	$\bar{1}$	0	1
1	0	1	×

× : 起こらない

表 5　ステップ 2 の真理値表

ここから冗長二進数の計算の例を示す。

加数と被加数が図 12 のように与えられたとすると、d_i, c_i は図のようになり、桁上げのステップが省略できていることがわかる。

冗長二進数のアルゴリズムを実現する回路は図 13 のようになる。

図 13 で示した加算回路は、一つにつき 1 ビット（扱うのは 2 ビット）の計算を行うため、実際の回路に用いる場合は計算する数のビット数個分だけ用意すればよいことになる。

この回路も作成することで桁上げで起こる遅延時間を完全に無視することができるが、回路規模としては大きなものになってしまう。

被加数		1 0 $\bar{1}$ 0 $\bar{1}$ 0 0 $\bar{1}$	
加数		+ 1 $\bar{1}$ 1 0 0 1 1 $\bar{1}$	ステップ1
中間和	d_i	0 1 0 0 $\bar{1}$ $\bar{1}$ 1 0	
中間桁上げ	c_i	+ 1 $\bar{1}$ 0 0 0 1 0 $\bar{1}$	ステップ2
和		1 $\bar{1}$ 1 0 0 0 $\bar{1}$ 0 0	

図 12　計算例

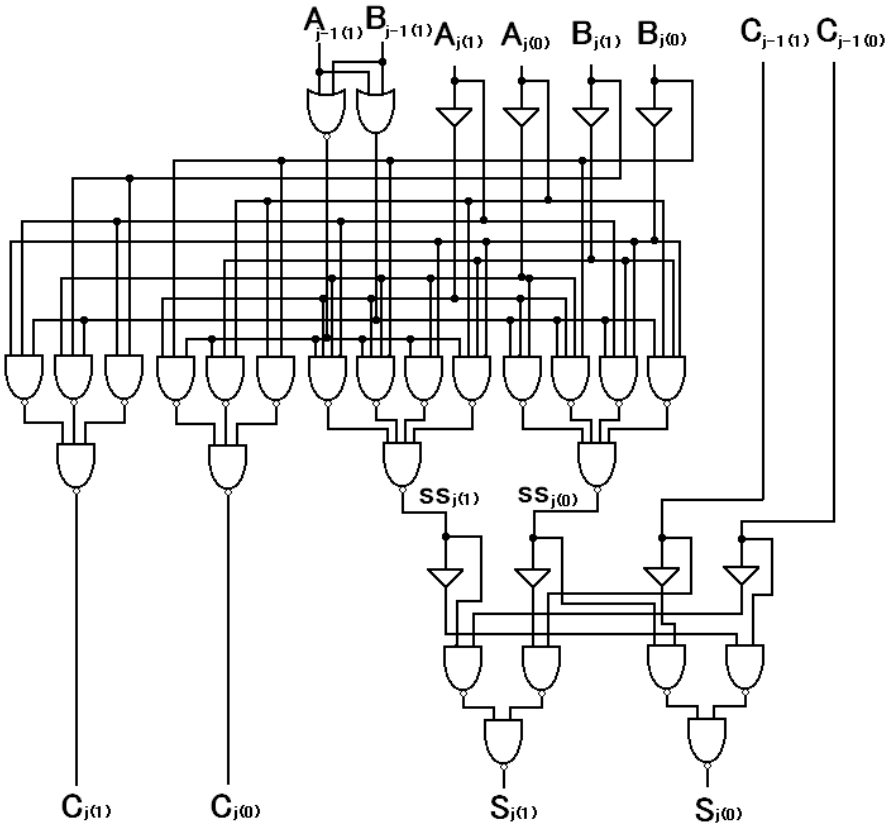


図 13　冗長二進数を用いた加算回路

4.2　計算時間比較

以上のアルゴリズムの計算時間と使用する回路容量を比較してみる。

図 14、図 15 で見たように、アルゴリズムによってはモジュールの規模が大きくなることや演算の遅延などが起こってしまう。しかし冗長二進加算のようにリソースを使用するアルゴリズムであっても、加算を多数回用いる場合などで十分な高速化が図れるようなアルゴリズムに対して用いられている [3]。

そのためハードウェア記述言語における回路設計には、求めるアルゴリズムと使用できる FPGA の資源との兼ね合いで最適な仕様を考えることが必要不可欠である。

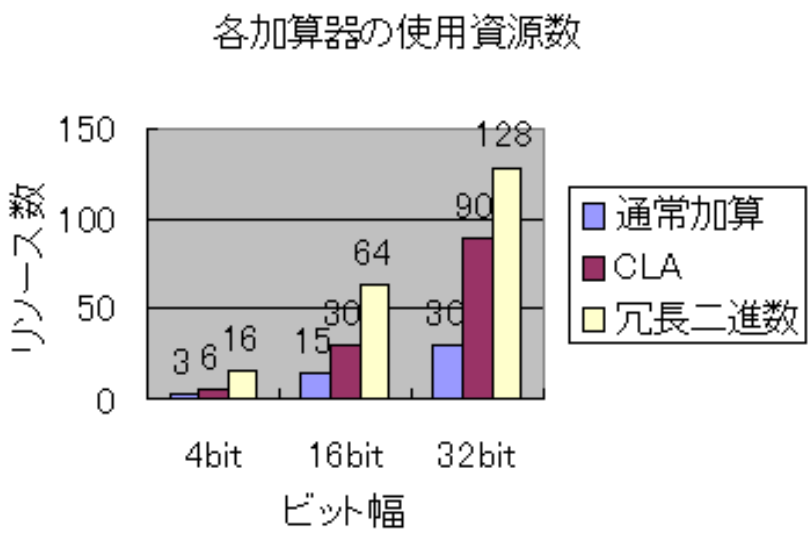

各加算器の使用資源数
150
100
50
0
リソース数
3
6
16
15
30
64
30
90
128
4bit
16bit
32bit
ビット幅
通常加算
CLA
冗長二進数

図 14

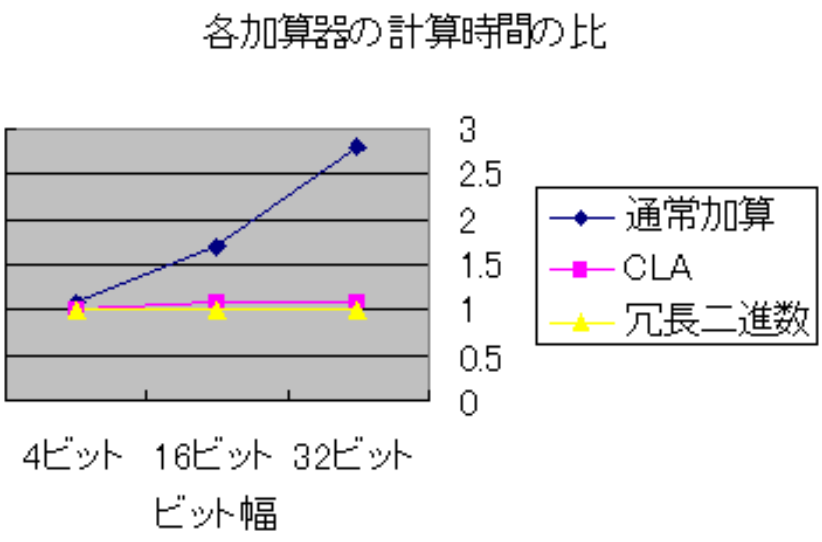

各加算器の計算時間の比
3
2.5
2
1.5
1
0.5
0
4ビット
16ビット
32ビット
ビット幅
通常加算
CLA
冗長二進数

図 15

5　剰余算アルゴリズム

この章ではハードウェア記述言語における剰余算アルゴリズムの代表的なものとして回復法、インターリーブ法、モンゴメリ剰余算、二分乗算剰余算のアルゴリズムについて延べ、さらにそれぞれを回路化したものの詳細ついて述べる。

5.1　回復法

シフト型除算は二進数での除算回路において、基本的なアルゴリズムとして知られているものである [14]。そのアルゴリズムとしては割り算を筆算で行う要領で、被除数の上位から、除数が引けるかどうかを計算し、引けるようなら商に 1 を書き込むという計算をビット数分だけ繰り返すというものである。

5.1.1　回復法のアルゴリズム

入力値を A、B、N とすると

$Input$ ： $N, 0 < A < N, 0 < B < N$　（それぞれ n ビットとする）
$Output$ ： $M = A * B \bmod N$

$STEP1$: $A * B$ を計算し M に保存する、$i = 0$
$STEP2$: M の上位 n ビットと N を比較し、減算を行う
$STEP3$: $STEP2$ の結果が負なら減算を行う前、正なら減算後の M の値を M の上位に保存する
$STEP4$: M の値を 1 ビット左シフトする、$i = i + 1$
$STEP5$: $i < n$ なら $STEP2$ へ
$STEP6$: M を出力

5.1.2　回復法の回路図

回路での計算も筆算と同様のアルゴリズムで行われる。図 16 にそのブロック図を載せた。また図 17 に二進数の筆算の例を載せる。

積の上位ビットと法を比較し、商に 1 が立つかどうかを判定しながら最下位ビットまで計算を行うものである。構成する回路が減算、シフト、mux（値を選択する回路）から成り立っているので、回路の容量は少なくて済む。しかし順次桁上げの加算回路のように、入力された数のビット幅分だけシフトを繰り返すステップが存在するため、計算時間としては高速ではない。

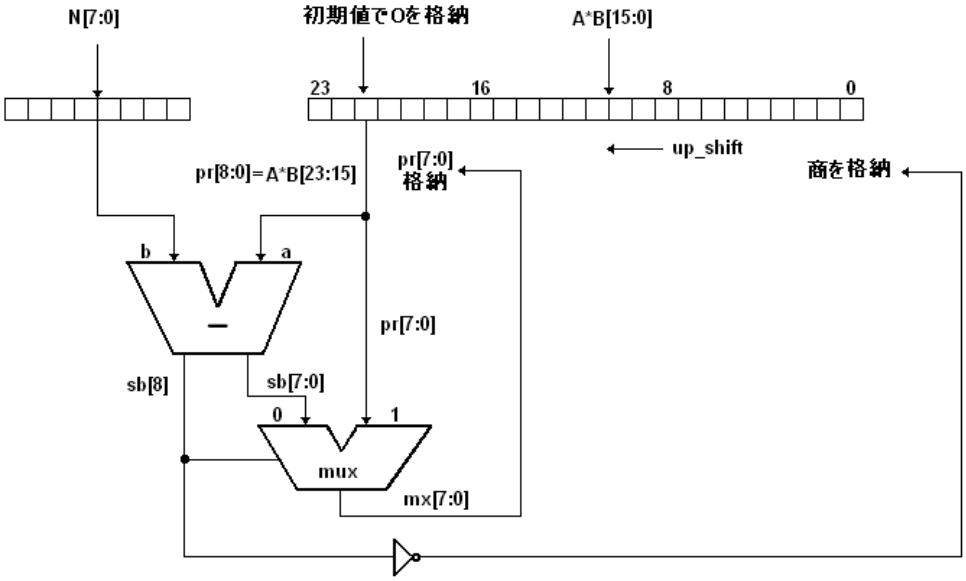


図 16　回復法の回路図

```
   0 2 2
 7 1 5 5
 - 0
   1 5
 - 1 4
     1 5
   - 1 4
       1
```

```
                 1 0 1 1 0
 0 1 1 1 ) 1 0 0 1 1 0 1 1
         - 0 1 1 1
             0 1 0 1
           - 0 0 0 0
               1 0 1 0
             - 0 1 1 1
                 0 1 1 1
               - 0 1 1 1
                     0 1
```

図 17　回復法の計算例（155 mod 7）

5.2 インターリーブ法

インターリーブ法とは、計算する数値データをを効率よく記録し計算の中間結果が大きくならないように計算することで、回復法よりも容量を使わずに剰余の計算を行うことができるアルゴリズムである [10]。

5.2.1 インターリーブ法のアルゴリズム

入力値を A、B ($= [b_{n-1}, b_{n-2}, \ldots, b_0]$)、$N$ とする。

$Input : N, 0 < A < N, 0 < B < N$
$Output : M = A * B \bmod N$

$STEP1 : i = 0, j = 0, P_j = 0$
$STEP2 : P_{j+1} = 2 * P_j + b_{n-j-1} * A$
$STEP3 : i = i + 1, j = j + 1, P_{j+1} < N$ になるまで $P_{j+1} = P_{j+1} - N$
$STEP4 : i < n$ なら $STEP2$ へ
$STEP5 : M$ を出力

5.2.2 インターリーブ法の回路図

図 18 にインターリーブ法の回路のブロック図を載せる。

次に 法 211、被乗数 $A = 79$、乗数 $B = 108(= 01101100)_2$でのインターリーブ法での計算フロー図 19 に示す

計算例では最大で 9 桁の数しか扱っていないため、回路の容量は回復法よりも小さくできる。しかし回復法と同様にビット幅の分だけシフトを繰り返さなくてはならないため、大きなビット幅の数になると計算時間がかかってしまう。そこで法の数を大きくする（一度に 2 ビットや 4 ビットずつ計算する）などして計算のステップ数を少なくし、高速化を図る研究がなされている [2]。

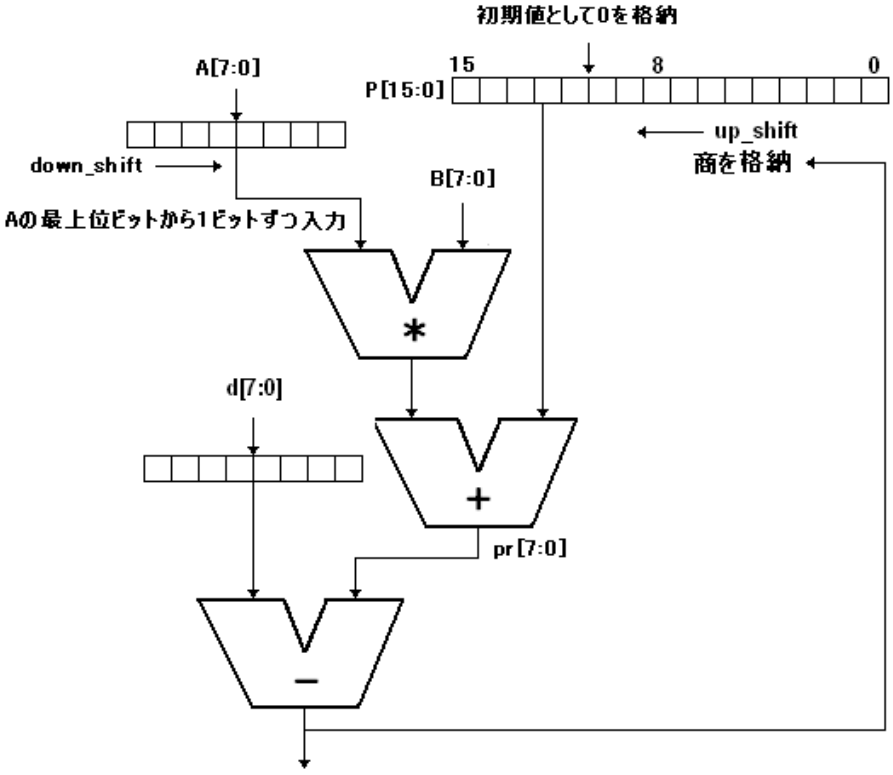


図 18　インターリーブ法の回路図

P_0		0	0	0	0	0	0	0	0
$2P_0$	0	0	0	0	0	0	0	0	
b_7A	+	0	0	0	0	0	0	0	0
mod $N(=P_1)$		0	0	0	0	0	0	0	0
$2P_1$	0	0	0	0	0	0	0	0	
b_6A	+	0	1	0	0	1	1	1	1
mod $N(=P_2)$		0	1	0	0	1	1	1	1
$2P_2$	0	1	0	0	1	1	1	1	
b_5A	+	0	1	0	0	1	1	1	1
mod $N(=P_3)$		0	0	0	1	1	0	1	0
$2P_3$	0	0	0	1	1	0	1	0	
b_4A	+	0	0	0	0	0	0	0	0
mod $N(=P_4)$		0	0	1	1	0	1	0	0
$2P_4$	0	0	1	1	0	1	0	0	
b_3A	+	0	1	0	0	1	1	1	1
mod $N(=P_5)$		1	0	1	1	0	1	1	1
$2P_5$	1	0	1	1	0	1	1	1	
b_2A	+	0	1	0	0	1	1	1	1
mod $N(=P_6)$		0	0	0	1	0	1	1	1
$2P_6$	0	0	0	1	0	1	1	1	
b_1A	+	0	0	0	0	0	0	0	0
mod $N(=P_7)$		0	0	1	0	1	1	1	0
$2P_7$	0	0	1	0	1	1	1	0	
b_0A	+	0	0	0	0	0	0	0	0
		0	1	0	1	1	1	0	0

図 19　インターリーブ法の計算例

5.3 モンゴメリ剰余算

モンゴメリ剰余算とは、ハードウェアで作成する剰余算回路において、高速化を実現でき、様々な分野で用いられているアルゴリズムである [4] [13]。

5.3.1 モンゴメリ法のアルゴリズム

入力として N ($< R$)、(ここで $R = 2^{r*m}$)、$A(= [a_{m-1}, \ldots, a_0]_R)$、$B(= [b_{m-1}, \ldots, b_0]_R)$ をとるとする。

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

$STEP1 : M = 0, i = 0$
$STEP2 : Q = (M + A_i * B_0)V \bmod 2^r$
$STEP3 : M = M + A_i * B + Q * N$
$STEP4 : M = \frac{M}{2^r}, i = i + 1$
$STEP5$: もし $i < m$ なら $STEP2$ へ
$STEP6$: もし $N < M$ なら $M = M - N$
$STEP7$: M を出力

先に述べたように暗号には 1000 ビット以上の大きな数に対しての演算を扱う。その際大きなビット幅を計算する乗算回路を用いることは困難であるため、小さいビットの乗算器を複数用いる、もしくは繰り返し用いることにより大きな数に対応できるようにしたアルゴリズムである。

それぞれ m 分割した入力値 A、B の部分積を STEP2、STEP3 で求める。その際、前計算で得た V を用いることで、STEP3 において下位ビットを 0 にするような Q を、法 N から求めることができる。これは STEP3 の式が $M + A_i * B + Q * N = 0 (\bmod 2^r)$ となるように Q を $Q = (M + A_i * B)(-N^{-1}) (\bmod 2^r)$ と変形できることから出る。

その要領で下位に 0 を詰めていくことで求める値を計算していく。

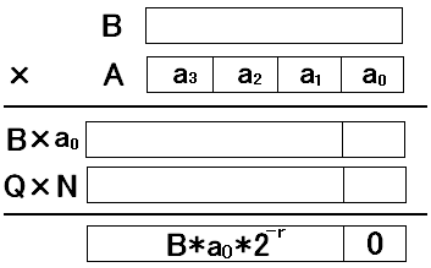


図 20　モンゴメリ法での STEP2、STEP3 を一回行った図

図 20 はモンゴメリ法アルゴリズムの STEP2 と STEP3 を一度行った概略図である。V によって Q を変形することで、下位ビットが 0 になる。

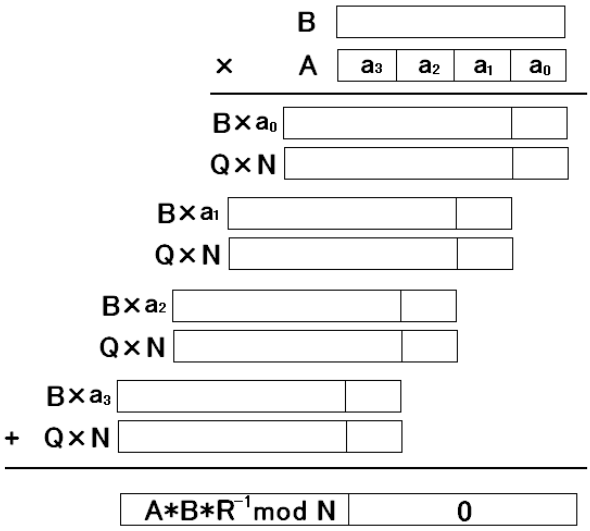


図 21　モンゴメリ法最終段階

これを分割した分だけ計算させたものが図 21 である。これで上位ビットに求めたい値を出力することができることになる。

5.3.2　モンゴメリ法の回路図

モンゴメリ法の回路図は図 22 のようになる。

モンゴメリ法は高速に演算を行うことができることから、暗号の分野で用いられている [13]。

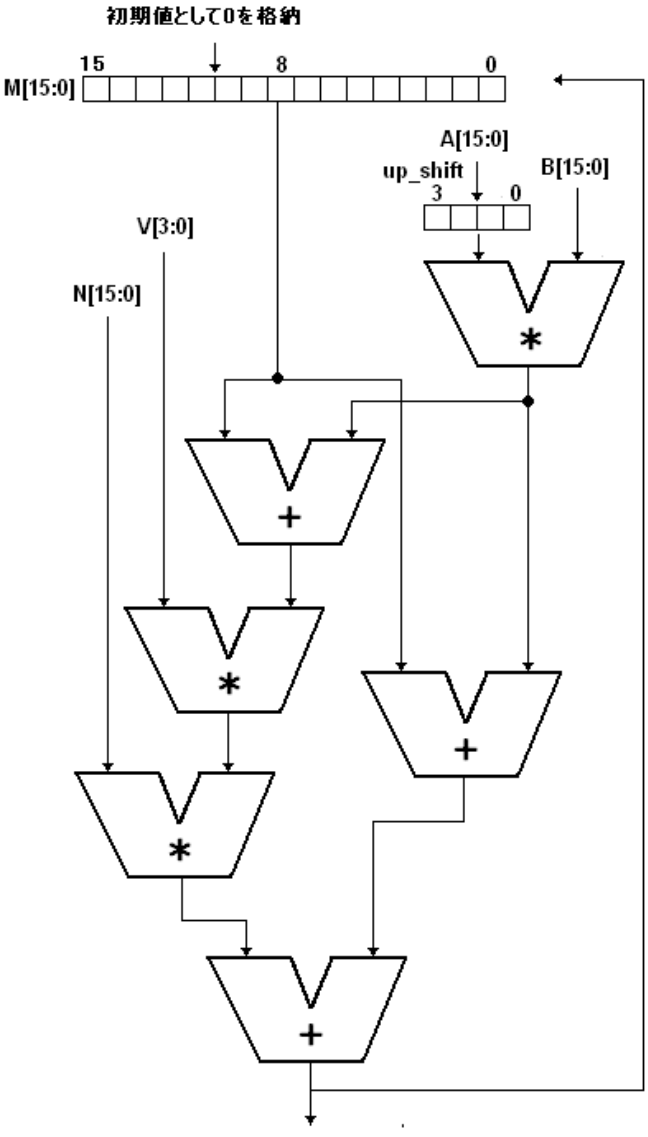


図 22　モンゴメリ法の回路図

5.4 二分乗算剰余算

二分乗算剰余算 [9] とは、近年開発されたモンゴメリ法よりも高速に乗算剰余算を解くことができるとされているアルゴリズムである。

5.4.1 二分乗算剰余算のアルゴリズム

前述のアルゴリズムのうち、インターリーブ法が上位ビットから、モンゴメリ法が下位ビットから計算を行うということから、それぞれ独立させて計算を行い、高速化を施すというアルゴリズムである。

アルゴリズムとしては、

$STEP1$: 乗数の一方を上位と下位に分割する。
$STEP2$: 上位をインターリーブ法、下位をモンゴメリ法で計算する。
$STEP3$: 出力した値を加算し、もし $M < N$ なら $M = M - N$ とする
$STEP4$: M を出力する

となる。

このアルゴリズムはモンゴメリ法とインターリーブ法の計算速度に応じて乗数の分割サイズを決定することで最適化できる。両方のアルゴリズムが同様のスピードで計算できるとすれば、最大で二倍に高速化できるということになる。

5.4.2 二分乗算剰余算の回路図

図 23 に二分乗算剰余算の回路図を示す。

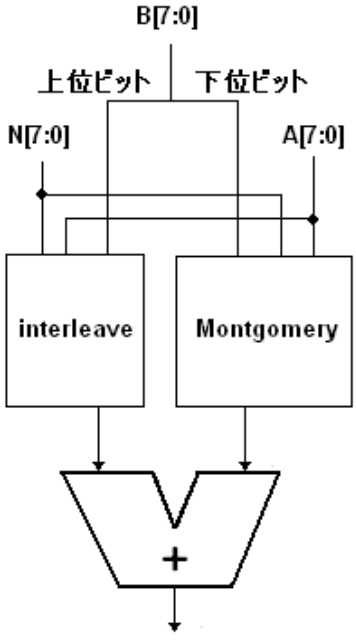


図 23　二分乗算剰余算の回路図

5.5　実装評価

以上で示したアルゴリズムを実装した結果は以下のようになる

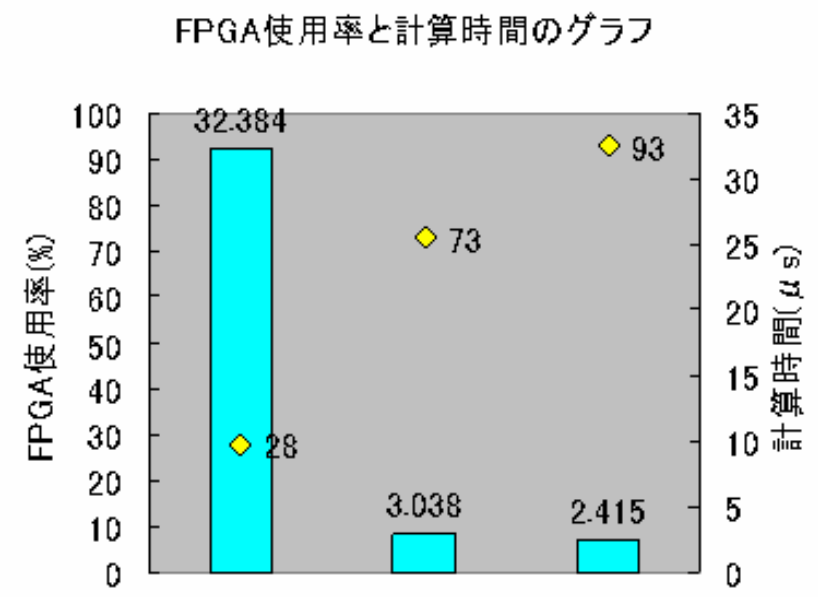


図 24　各アルゴリズムの計算時間と資源占有率

図 24 は 352 ビット同士の積を取った数を 352 ビットの数で剰余を取る計算を行う回路の計算速度と資源占有率のグラフである。

インターリーブ法は資源を使わない代わりに速度が遅い。またモンゴメリ法は高速化を図ることができている代わりに資源を使う。モンゴメリ法のアルゴリズムでの計算をインターリーブ法で手助けをすることで、モンゴメリ法よりも高速に計算を行えることを実証できた。

5.6　二分乗算剰余算回路作成で工夫した点

今回は FPGA の資源に応じた回路設計をしたため、インターリーブ法の回路部分の高速化がうまく図れなかった。その分モンゴメリ法の計算部を大きくすることを試みた。

モンゴメリ法の回路の中で分割した A_i と B の積を求める際には、ビット幅の大きな乗算回路を二つ用いなくてはならなかった (図 25)。そこで共通の乗算回路をひとつにまとめることで回路の共有化を行い、回路の縮小化を図った (図 26)。

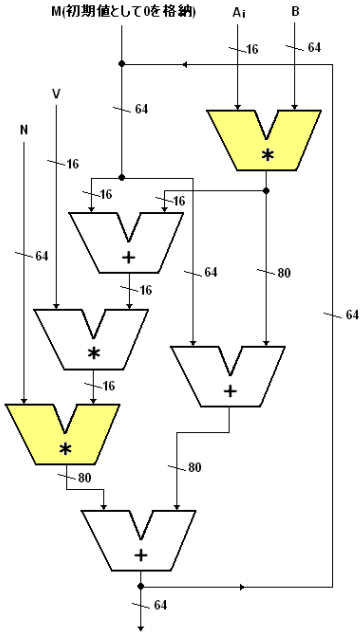


図 25　モンゴメリ法の共通の乗算回路

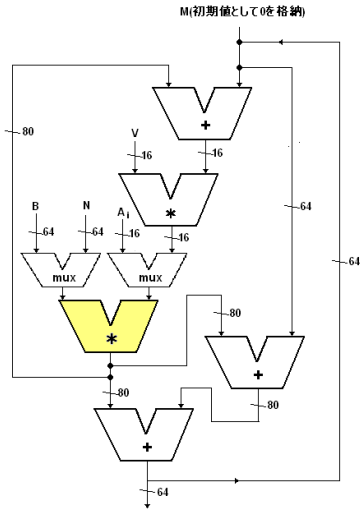


図 26　共通の乗算回路を一つにまとめた回路

この工夫の結果として素子数を抑えることができ、省略をしない回路よりも大きなビット幅を計算できる回路を作成することができた。

5.7 結果

今回の研究で、ハードウェア記述言語における剰余算モジュールの性能の評価を行い、352 ビット× 352 ビットの数を 352 ビットで剰余をとる計算を 2415ns で実行する回路を作成することができた。これは RSA 暗合に用いるビット幅には足りないが、他のアルゴリズムであれば十分暗号化・復号化に用いることができるビット幅である。また現在暗合処理に用いられているモンゴメリ法よりも高速な計算ができることが実証できた。

5.8 今後の課題

今回の研究では諸所のアルゴリズムを用いることで、乗算剰余算の計算速度を高速にできるということを実証することができた。しかし容量の問題で、冗長二進数などのさらに高速化できる概念を取り入れることなどができなかった。容量の大きな FPGA を用いることで、それらの概念を取り入れさらに高速化するアルゴリズムについて研究・検証すること。さらに、より大きなビット幅の数にも対応できる回路設計をし計算速度を検証するということが必要である。

また現在の回路などをさらに細分化し動作を独立化させることでパイプライン化を施すなど、高速化を図る可能性はあると考えられる。

6　まとめ

本論文では、現在欠くことの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、暗号化・複合化をする際に繰り返し用いられる剰余演算に注目した。演算を高速に計算するために、ハードウェアにおいて有効とされる諸アルゴリズムに関して考察を行った。近年開発された二分乗算剰余算のアルゴリズムのハードウェア実装を行うことで、現在高速に剰余算を計算できるとされているモンゴメリ法と比較して 1.12 倍ほどの高速化を図ることができた。またさらに最適化を施せることから、二分乗算剰余算のアルゴリズムが剰余算の計算に有効であるということを実証できた。

7　謝辞

本研究を通じ、ハードウェア記述言語での効果的なモジュール作成、有効な並列アルゴリズムなど、多くのことを学ぶことができました。

本研究に適切なご指導と助言を与えてくださった指導教官福永力教授に深く感謝いたします。研究の発端を与えてくださった中村憲教授、内山成憲准教授、西本啓一郎氏にも深く感謝します。

並列処理でのアルゴリズム研究に助言を下さった田中和人氏、坂本達哉氏にも感謝いたします。また諸手伝いをして下さった研究室の方、友人、家族に感謝の意を表します。

参考文献

[1] D E.Thomas 、P R .Moordy 共著
『設計言語　Verilog-HDL 入門』(培風館、1995)

[2] 葛　毅、櫻井　隆雄、L D.Hung、阿部　公輝、酒井　修一　：　電子情報通信学会論文誌
『インターリーブ型剰余乗算回路の評価』

[3] 松村　暢也　：　高知工科大学　電子・光システム工学　修士論文
『冗長二進数を用いた RSA 用乗算器の高速化』(2004)

[4] P.L.Montgomery
『Modular Multiprication without Trial Division』(Mathmatics of Computation vol44 pp519-521,Apr1985)

[5] R.L.Rivest,A.Shamir,and　L.Adleman
『A Method for Obtaining Digital Signatures and Public-Key Cryptosystems』(Comm.ACM, vol21 ,pp.120-126 1978)

[6] 桜井　至
『HDL 設計入門』(株式会社テクノプレス、1996)

[7] 櫻井　隆雄　：　東京大学　修士論文
『親子剰余乗算器を用いた左向きアレイ法の実装方式』(2004)

[8] 新保　敦、野崎　華恵、川村　信一
『高速 RSA 暗号 LSI』(東芝レビュー、Vol56、No7、2001)

[9] 高木　直史
『算術演算の VLSI アルゴリズム』(コロナ社、2001)

[10] 高木　直史、高木　一義
『ハードウェアアルゴリズムの性能評価に関する研究』(京都大学　成果報告書、2005)

[11] 渡波　郁
『CPU の創りかた』(株式会社毎日コミュニケーションズ、2003)

[12] 山口　健輔　：　東京大学　新領域創成科学研究科　修士論文
『アドホックネットワークにおける (k,n) 閾値法を用いた認証方式に関する研究』(2005)

[13] 吉原　智明　：　北陸先端科学技術大学　情報科学研究科　修士論文
『高速モンゴメリ乗算回路に関する研究』(2003)

[14] 鈴木　昌治
『割り算回路設計、あの手この手』(Design Wave Magagine、2005-8〜)