A-04

多倍長整数演算ライブラリをリンクしたバイナリコードからの RSA 暗号回路の高位合成

High Level Synthesis from Binary Code Linked with Multiple Precision Arithmetic Library to RSA Encrypt/Decrypt Circuit

伊藤 直也 † Naoya Ito　竹林 陽 † Hinata Takebayashi　神原 弘之 ‡ Hiroyuki Kanbara　石浦 菜岐佐 † Nagisa Ishiura

1 はじめに

近年, スマートメーターに代表される M2M (Machine to Machine) 機器が爆発的に増大している. M2M 向け組込み機器ではセキュリティ確保のため, これまで IC カードとカードリーダ間で行われてきた, 相互認証あるいは無線通信の暗号化機能が必要とされている [1].

DES, AES 等の共通鍵ブロック暗号については, データの受信, 処理, および出力をストリーム処理する専用ハードウェアを搭載することにより, 大幅な性能改善を図ることができる. 一方, 機器間の相互認証や共通鍵の受渡しに用いられる RSA [2], 楕円暗号等の公開鍵暗号は, 演算処理が複雑であり, データをメモリ上に保存して更新するプロセッサ型の処理として実装せざるを得ない.

M2M 機器ではバッテリ容量の制限により, プロセッサの動作周波数の向上による処理時間の短縮が困難である. 鍵長や通信処理時間の制約もアプリケーションにより異なるため, 消費電力とレイテンシ制約を満たす暗号化ハードウェアを, 短期間で設計する手法が課題となっている.

高位合成技術の一つであるバイナリ合成 [3] は, アセンブリや機械語プログラムを入力として, レジスタ転送レベル (RTL) のハードウェア回路を自動生成するコンパイラ技術である. 公開鍵暗号等の複雑な演算処理を, 消費電力やレイテンシの制約を満たす専用ハードウェアとして, 短期間に実現することがバイナリ合成により可能になりつつある.

本稿では, 2048 ビットの RSA 暗号化/復号回路をバイナリ合成を用いて設計した. RSA 暗号化/復号プログラムに, 多倍長精度ライブラリ GMP [4] を簡約化したものをリンクして得られる MIPS のバイナリコードから, バイナリ合成システム ACAP [5] により回路を合成した. PC および MIPS R3000 互換プロセッサ [6] 上でプログラムの動作検証を行った後, 合成した回路を RTL シミュレーションおよび FPGA 上で実行し, そのメモリダンプが PC および MIPS のものと一致することを確認した. MIPS R3000 と比較して, 合成した回路は約 1.6 倍の回路規模で, 暗号化, 復号共に処理を 7.6 倍高速化できた.

† 関西学院大学, Kwansei Gakuin University
‡ 京都高度技術研究所, ASTEM RI

表 1　RSA 暗号に用いる鍵データ

鍵	説明
P, Q	ランダムに生成された素数
N	PQ
E	$gcd(E,(P-1)(Q-1))=1$ $(E<(P-1)(Q-1))$
D	$E^{-1} \pmod{(P-1)(Q-1)}$
$DMP1$	$D \bmod (P-1)$
$DMQ1$	$D \bmod (Q-1)$
$IQMP$	$Q^{-1} \bmod P$

2 準備

2.1 RSA 暗号

RSA の暗号化/復号 [2] に用いる鍵データを表 1 に示す. N と E は公開鍵として公開し, D は秘密鍵として保管する.

平文を M, 暗号文を C としたとき, 暗号化は次の演算により行える.

$$C = M^E \bmod N$$

復号は, 中国人の剰余定理を利用することにより高速に実行できる. 素数 P, Q に加え, $DMP1$, $DMQ1$, $IQMP$ を秘密鍵とし, 以下の演算を順に実行する.

1. $Mp = C^{DMP1} \bmod P$
2. $Mq = C^{DMQ1} \bmod Q$
3. $v = (Mp - Mq)IQMP$
4. $M = Mq + Qv \bmod N$

2.2 多倍長精度演算ライブラリ GMP

GMP (GNU Multi-Precision Library) [4] は, GNU プロジェクトが提供する算術ライブラリであり, 任意精度の符号付き整数, 有理数, 浮動小数点数の演算を扱うことができる. GMP には, 互換インタフェースを 1 ファイルで実装した mini-gmp というサブセットがある. mini-gmp のコードは約 4,500 行であり, GMP 本体に比べて速度が落ちる反面, ビルドやコードの修正を容易に行うことができる.

GMP は, 図 1 に示す構造体 `mpz_t` によって多倍長整数を表現する. `_mp_d` は, 多倍長整数の絶対値を表現する `mp_limb_t` (`unsigned long int`) 型動的配列を指す

ポインタである．_mp_alloc は _mp_d が指す動的配列の要素数を，_mp_size は実際に使用されている要素数を保持する．多倍長整数が負の値であれば，_mp_size は負数となる．図 2 は，mpz_t を用いて，72 ビットの符号なし整数 0x21fedcba9876543210 を表現した例である．ここでは，unsigned long int のサイズは 32 ビットとする．多倍長整数の下位バイトから 32 ビットずつ，_mp_d の先頭から格納されていき，使用した要素数 3 が _mp_size に保存される．_mp_alloc の値は，多倍長整数の代入や演算によって必要な要素数が増える度，動的配列と共に更新される．

```
typedef struct
{
  int _mp_alloc;
  int _mp_size;
  mp_limb_t *_mp_d;
} __mpz_struct;

typedef __mpz_struct mpz_t[1];
```

図 1　多倍長整数を扱う構造体

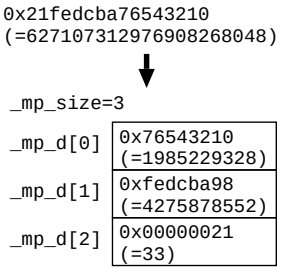

図 2　mpz_t を用いた多倍長整数の表現

多倍長整数の演算は，各演算に対応する関数を呼び出すことにより行う．例えば，加算は mpz_add(mpz_t r, const mpz_t a, const mpz_t b) を呼び出すことにより実行でき，a, b を加算した結果が r に格納される．

2.3　バイナリ合成システム ACAP

ACAP [5] は，MIPS R3000 の機械語プログラムを入力として，レジスタ転送レベルのハードウェアの記述を合成する処理系である．C 等の高水準言語を入力としてハードウェアを合成する高位合成 [7] に対して，このような技術はバイナリ合成と呼ばれ，ポインタやライブラリ呼出し等を含む広範なプログラムを処理の対象とすることができる．

ACAP の合成処理の流れを図 3 に示す．入力は MIPS アセンブリプログラムであり，これは コンパイルが C プログラムから生成したものでも手書きのものでもよい．このアセンブリを CDFG (Control Data Flow Graph) に変換し，データフロー解析，スケジューリング，バインディングの各処理を行って RTL の中間表現を合成し，最終的に Verilog HDL を出力する．

ACAP には，分割コンパイルモード，全体合成モード，アクセラレータ合成モードの 3 つの合成モードが実装されているが，本稿ではこのうち，全体合成モードを用いる．全体合成モードは，リンク済みの実行可能バイナリコードから，それを実行する MIPS と機能等価なハードウェアを

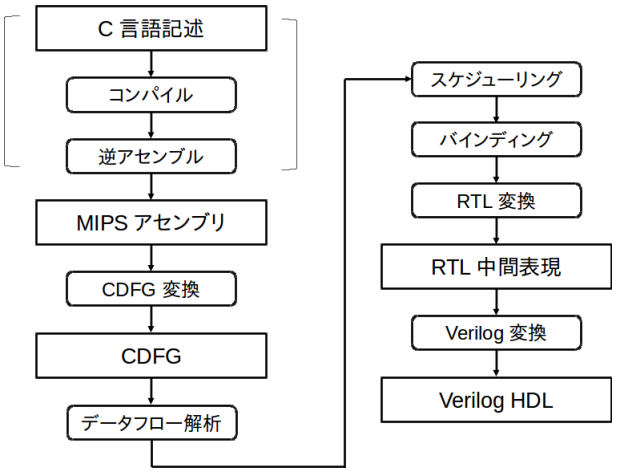

図 3　ACAP の合成処理の流れ

合成するものである．合成されたハードウェアは，MIPS と命令メモリを置き換えることができる．

3　GMP をリンクしたコードからの RSA 暗号回路のバイナリ合成

3.1　概要

本稿では，2048 ビットの RSA 暗号化/復号回路をバイナリ合成を用いて設計する．RSA 暗号化/復号プログラムを，高位合成向けに修正した mini-gmp とリンクして作成し，ACAP によりこのプログラムと動作等価な回路を合成する．

合成される回路の規模を抑制するため，mini-gmp から RSA の処理で使用しない機能は削除する．また，標準入出力や動的割り当て等，OS のサポートを要する処理は合成することができないため，必要なものは代替の処理に書き換える．また，処理速度向上のため，必要な関数は積極的にインライン展開されるようにする．作成したプログラムを PC および MIPS R3000 互換プロセッサ上で動作の検証を行った後，合成した回路を RTL シミュレーションおよび FPGA 上で実行し，メモリダンプが MIPS で実行したものと一致することを確認する．

3.2　RSA 暗号化/復号プログラム

RSA 暗号化/復号のメインプログラムを図 4 に示す．メインルーチン (70 〜 83 行目) では，制御用変数 _RUN_sub を監視し，その値が 1 になれば暗号化関数 encrypt を，2 になれば復号関数 decrypt を実行する．関数 encrypt (25 〜 35 行目) および関数 decrypt (37 〜 68 行目) では，2 章で示したアルゴリズムに従って暗号化および復号を行う．構造体 rsa_t (6 〜 19 行目) は，鍵および文データを格納するために用いる．グローバル変数として宣言した rsa_t 型変数 rsa1 (23 行目) に，必要な鍵および文データを格納し，関数 encrypt または関数 decrypt を実行することにより，その結果をそれぞれ配列 enc_out, dec_out に格納する．

3.3　mini-gmp の修正

本稿の RSA 暗号化/復号では，加算，減算，乗算，剰余算，べき乗算を用いる．これらの演算に必要な GMP のインタフェース関数を表 2 に示す．加算，減算，乗算には，表

```
01:  #include "rsa.h"
02:
03:  #define KEYBIT 2048
04:  #define KEYLEN KEYBIT / (sizeof(unsigned long) * 8)
05:
06:  typedef struct rsa {
07:    unsigned long n[KEYLEN];
08:    unsigned long e[1];
09:    unsigned long p[KEYLEN / 2];
10:    unsigned long q[KEYLEN / 2];
11:    unsigned long dmp1[KEYLEN / 2];
12:    unsigned long dmq1[KEYLEN / 2];
13:    unsigned long iqmp[KEYLEN / 2];
14:
15:    unsigned long plain_in[KEYLEN];
16:    unsigned long enc_in[KEYLEN];
17:    unsigned long enc_out[KEYLEN];
18:    unsigned long dec_out[KEYLEN];
19:  } rsa_t;
20:
21:  void* volatile _RUN_sub;
22:
23:  rsa_t rsa1;
24:
25:  static void encrypt(void)
26:  {
27:    mpz_t mpz_plain, mpz_enc, mpz_n, mpz_e;
28:
29:    mpz_init(mpz_plain, rsa1.plain_in, KEYLEN);
30:    mpz_init(mpz_enc, rsa1.enc_out, 0);
31:    mpz_init(mpz_n, rsa1.n, KEYLEN);
32:    mpz_init(mpz_e, rsa1.e, 1);
33:
34:    mpz_powm(mpz_enc, mpz_plain, mpz_e, mpz_n);
35:  }
36:
37:  static void decrypt(void)
38:  {
39:    mpz_t mpz_enc, mpz_dec, mpz_n, mpz_p, mpz_q;
40:    mpz_t mpz_dmp1, mpz_dmq1, mpz_iqmp;
41:    mpz_t mpz_decp, mpz_decq, mpz_tmp;
42:    mpz_t mpz_one;
43:    unsigned long one_data[1] = {1};
44:    unsigned long decp_data[ARRAY_SIZE];
45:    unsigned long decq_data[ARRAY_SIZE];
46:    unsigned long tmp_data[ARRAY_SIZE];
47:
48:    mpz_init(mpz_one, one_data, 1);
49:    mpz_init(mpz_enc, rsa1.enc_in, KEYLEN);
50:    mpz_init(mpz_dec, rsa1.dec_out, 0);
51:    mpz_init(mpz_n, rsa1.n, KEYLEN);
52:    mpz_init(mpz_p, rsa1.p, KEYLEN / 2);
53:    mpz_init(mpz_q, rsa1.q, KEYLEN / 2);
54:    mpz_init(mpz_dmp1, rsa1.dmp1, KEYLEN / 2);
55:    mpz_init(mpz_dmq1, rsa1.dmq1, KEYLEN / 2);
56:    mpz_init(mpz_iqmp, rsa1.iqmp, KEYLEN / 2);
57:    mpz_init(mpz_decp, decp_data, 0);
58:    mpz_init(mpz_decq, decq_data, 0);
59:    mpz_init(mpz_tmp, tmp_data, 0);
60:
61:    mpz_powm(mpz_decp, mpz_enc, mpz_dmp1, mpz_p);
62:    mpz_powm(mpz_decq, mpz_enc, mpz_dmq1, mpz_q);
63:    mpz_sub(mpz_tmp, mpz_decp, mpz_decq);
64:    mpz_mul(mpz_tmp, mpz_tmp, mpz_iqmp);
65:    mpz_mul(mpz_tmp, mpz_tmp, mpz_q);
66:    mpz_powm(mpz_tmp, mpz_tmp, mpz_one, mpz_n);
67:    mpz_add(mpz_dec, mpz_tmp, mpz_decq);
68:  }
69:
70:  int main(void)
71:  {
72:    START:
73:      while (!_RUN_sub) {;}
74:
75:      if (_RUN_sub == (int*)1) {encrypt();}
76:      else if (_RUN_sub == (int*)2) {decrypt();}
77:      else {}
78:      _RUN_sub = (int*)0;
79:
80:      goto START;
81:
82:    return 0;
83:  }
```

図 4　RSA 暗号化/復号プログラム

2 のそれぞれの演算に対応する関数を用い, 剰余算, べき乗算には, mpz_powm を用いる.

表 2　RSA 暗号化/復号に必要な関数

関数	機能
mpz_init(r)	多倍長整数を扱う構造体 r の初期化
mpz_add(r, a, b)	$r = a + b$
mpz_sub(r, a, b)	$r = a - b$
mpz_mul(r, a, b)	$r = ab$
mpz_powm(r, b, e, m)	$r = b^e \bmod m$

表 2 に示した関数およびそれらから呼び出される関数以外は, RSA 暗号化/復号では利用しないため, すべて削除できる. また, mpz_powm は, 引数 e が負数かそれ以外で処理が分岐するが, RSA 暗号化/復号では e は常に正なので, e が負数の場合の処理およびこの処理のみで用いられる関数は削除できる.

mini-gmp は, 標準ライブラリとして assert.h, ctype.h, stdio.h, string.h, stddef.h, limits.h, stdlib.h を利用している.

assert.h, ctype.h, stdio.h, string.h は, 実行中診断, 標準入出力, 文字列による多倍長整数の操作等を行うために利用している. これらの機能は, RSA 暗号化/復号回路では使用しないためそのまま削除する.

stddef.h, limits.h は, sizeof 演算子の結果の型 size_t, 空ポインタ定数 NULL, char 型のビット数 CHAR_BIT を参照するために利用している. これらの型および値は, mini-gmp 内に再定義し, プログラム内でそのまま参照できるようにする.

stdlib.h は, プログラムの異常終了, 動的配列の割り当てを行うために利用している. プログラムの異常終了処理は, RSA 暗号化/復号回路では使用しないためそのまま削除する. 動的配列の割り当てに関する処理は, mini-gmp 内の関数ごとに, それぞれ以下に示すように修正する.

- gmp_xalloc_limb(size)
  関数 malloc を用いて, 要素数 size 分の unsigned long int 型の動的配列の割り当てを行う関数マクロである. 本稿では, 各関数でこの関数を用いる代わりに, 十分な要素数を持った配列をローカル変数として定義し用いる.
- mpz_init (mpz_t r)
  mpz_t 型変数 r の初期化を行う関数である.
  本稿では, 引数として新たに mp_ptr (unsigned long int*) 型変数 d と, int 型変数 size を取る. d は, 呼出し元で定義された十分な大きさ (ARRAY_SIZE) を持った配列であり, これを _mp_d に代入する. _mp_alloc には, d の要素数 ARRAY_SIZE を代入する. size は, d で使用されている要素数を表す. 初期化の際, d に既に値が定義されているならそのサイズを渡し, 定義されていなければ 0 を渡すものとする.
- MPZ_REALLOC(z, n)
  関数 realloc を用いて, mpz_t 型変数 z の _mp_d が指す動的配列の再割り当てを行う関数マクロである. 本稿では, 再割り当てを行わず, z の _mp_d をそのまま返すようにする.
- mpz_swap(mpz_t u, mpz_t v)
  mpz_t 型変数 u, v の _mp_alloc, _mp_size, _mp_d を入れ替える関数である.
  この関数は, mpz_mul や mpz_powm の中で, 演算に使用される作業用の mpz_t 型変数と, 引数として受け取った出力用の mpz_t 型変数を入れ替えるために用いる. 作業用の mpz_t 型変数の _mp_d は, 関数内で定義されたローカル変数配列を指すように mpz_init を用いて初期化される.
  しかし, mpz_swap が行う _mp_d の入れ替えは, そのポインタを入れ替えるだけである. ローカル変数は, 関数の終了と共に破棄されるため, この処理では演算

表 3　RSA 暗号化/復号回路の合成結果

	ALU	MLDV	Slices	FFs	LUTs	Delay [ns]	Cycles (Enc)	Cycles (Dec)
MIPS	-	-	2,718 (1.00)	3,587 (1.00)	6,475 (1.00)	21.22 (1.00)	26,028,247 (1.00)	1,335,280,390 (1.00)
HW1	3	1	4,302 (1.58)	1,884 (0.53)	13,497 (2.08)	21.89 (1.03)	3,690,096 (0.14)	187,596,163 (0.14)
HW2	2	2	4,342 (1.60)	1,828 (0.51)	13,018 (2.01)	21.89 (1.03)	3,467,874 (0.13)	176,482,547 (0.13)

結果を正しく受け取ることができない.
本稿では, u と v それぞれの _mp_d が指す配列の中身を先頭から入れ替える処理に変更する. これにより, 演算を行う関数の中のローカル変数が破棄されても, 演算結果を正しく受け取ることができる.

- gmp_free(d)
  関数 free を用いて, 動的配列 d の解放を行う. 本稿における mini-gmp では, 動的配列は用いないため, この関数はそのまま削除する.

また, 合成対象とする関数には, すべて static 修飾子を付加する. これにより, 使用頻度の少ない関数はコンパイル時にインライン展開が行われる.

3.4 検証の方針

作成した RSA 暗号化/復号プログラムと生成した回路の検証は, 以下に示す手順に従って行う. 検証に使用する鍵データは OpenSSL [8] を用いて生成, シミュレーションは Xilinx ISim 14.1 で行う.

1. PC 上での検証
   プログラムを PC 上で実行し, 暗号文を復号した結果が元の平文と一致することを確認する.
2. MIPS R3000 互換プロセッサ上での検証
   プログラムを MIPS R3000 互換プロセッサ上で RTL シミュレーションにより実行し, 暗号文を復号した結果が元の平文と一致することを確認する. また, 平文を暗号化した結果が, 同じ平文を 1 で暗号化した結果と一致することを確認する.
3. 合成した回路の検証
   プログラムからバイナリ合成を用いて生成した回路を RTL シミュレーションにより実行し, 暗号文を復号した結果が元の平文と一致することを確認する. また, 平文を暗号化した結果が, 同じ平文を 1, 2 で暗号化した結果と一致することを確認する.
4. FPGA 上での検証
   生成した回路を, 論理合成および配置配線によって FPGA 上に実装し, その上で実行して, 暗号文を復号した結果が元の平文と一致することを確認する. また, 平文を暗号化した結果が, 同じ平文を 1, 2, 3 で暗号化した結果と一致することを確認する.

4 合成および性能評価

ACAP を用いて生成した RSA 暗号化/復号回路を, Xilinx ISE 14.1 を用いて FPGA Xilinx Spartan-6 LX150 をターゲットとして論理合成, 配置配線を行った. C プログラムのコンパイルは GCC 4.8.2 (mips-elf ターゲット) で, 最適化オプション -Os を指定して行った. なお, 合成処理において, チェイニング (依存関係のある複数の演算を 1 クロックサイクルで行うこと) は行っていない.

合成結果を表 3 に示す. MIPS は MIPS R3000 互換プロセッサ, HW1, HW2 は本稿の RSA 暗号化/復号回路である. ALU と MLDV は, それぞれ合成時に指定した ALU と乗除算器の個数である. Slices, FFs, LUTs, Delay は, それぞれのスライス数, フリップフロップ数, LUT 数, 遅延時間である. ただし, HW1, HW2 の回路規模は, MIPS と RSA 暗号化/復号回路をまとめて合成したものから, MIPS の回路規模を引いたものである. Cycles (Enc) と Cycles (Dec) は, それぞれ暗号化, 復号の実行に要した実行サイクル数である.

今回は, ALU と MLDV の数を 2 通りの組み合わせで実験したが, MLDV を 2 つ用いる構成の方が若干高速であり, MIPS R3000 互換プロセッサに比べ約 1.6 倍の回路規模で暗号化/復号とも約 7.6 倍の高速化が達成できた. このように, 種々のデータパス構成に対して回路規模と処理速度の探索が用意に行えることが, バイナリ合成を用いる一つの利点である.

RSA 暗号化/復号プログラムの作成, 回路の合成, FPGA への実装, 動作検証まで約 9 ヶ月 (エフォート 30% × 1 人) で完了した.

5 むすび

本稿では, RSA 暗号化/復号回路を ACAP によるバイナリ合成を用いることによって自動的に設計し, その動作結果が正しいことを確認した.

今後の課題としては, RSA 暗号化/復号回路の回路規模や遅延時間削減のための, ACAP のスケジューリング, バインディングアルゴリズムの改善等が挙げられる.

謝辞

本稿の研究は, 京都高度技術研究所を通じて独立行政法人情報処理機構 (IPA) の協力で実施したものである. 本稿の研究に関して有益な御助言を頂いた元立命館大学の中谷嵩之氏, 元京都大学の矢野正治氏, 元関西学院大学の田村真平氏に感謝致します.

参考文献

[1] 瀬戸 洋一 著: 情報セキュリティ概論, 日本工業出版 (2007).
[2] 岡本 栄司 著: 暗号理論入門 第 2 版, 共立出版 (2002).
[3] G. Stitt and F. Vahid: "Binary synthesis," *ACM Trans. on Design Automation of Electronic Systems*, vol. 12, no. 3, article 34 (Aug. 2007).
[4] http://gmplib.org/ (accessed 2015-07-01).
[5] N. Ishiura, H. Kanbara, and H. Tomiyama: "ACAP: Binary Synthesizer Based on MIPS Object Codes," in *Proc. ITC-CSCC 2014*, pp. 725–728 (July 2014).
[6] 神原弘之, 金城良太, 矢野正治, 戸田勇希, 小柳滋: "パイプラインプロセッサを理解するための教材: RUE-CHIP1 プロセッサ," 情処関西支部大会, A-09 (Sept. 2009).
[7] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin:*High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).
[8] http://www.openssl.org/ (accessed 2015-07-01).