修 士 論 文

楕円曲線暗号の効率的な実装に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

永田　智芳

2010 年 9 月

修 士 論 文

楕円曲線暗号の効率的な実装に関する研究

指導教官　宮地　充子 教授

審査委員主査　宮地　充子 教授
審査委員　金子　峰雄 教授
審査委員　上原　隆平 准教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

0710902　永田　智芳

提出年月: 2010 年 8 月

概 要

楕円曲線暗号は小さな鍵サイズでも安全性が高く，処理速度が速く，消費電力が低く，メモリ使用量も少ないため，従来の暗号に比べて注目されている．楕円曲線暗号を効率的に実装するためには，楕円曲線上のスカラー倍算を高速に処理する必要がある．スカラーの大きさは多倍長で160ビット以上に及ぶため，多項式時間で計算することが不可欠であり，高速化の研究が進められてきた．

多項式時間で計算する方法は，Binary Methodに始まり，Sliding Window法，そしてオープンソースの暗号ライブラリであるOpenSSLの楕円曲線ライブラリには，Interleaving Exponentiation[7]と呼ばれる方法が採用されている．

どのようなスカラー倍算アルゴリズムであっても，内部で実行される演算は楕円加算と楕円２倍算である．楕円加算と楕円２倍算の計算量を少なくするための研究も進められており，最新の成果[8]を利用することによって，OpenSSLのスカラー倍算は高速化することが可能である．

楕円加算と楕円２倍算の内部で実行される演算は，多倍長mod演算である．楕円曲線暗号で使用する定義体は，素数pを法とする有限体であるから，高速な法演算を実行するためのアルゴリズムも研究されてきた．Incomplete Reduction[9]という方法は，高速にMod演算を行なうために，ビット単位ではなく，ワード単位で計算を行なっている．Incomplete Reductionを導入することによって，楕円加算と楕円２倍算は高速化することが可能である．

多倍長mod演算の内部で実行される演算は，多倍長演算である．多倍長演算は何度も繰り返し実行されるため，その高速化がスカラー倍算の実行時間に与える影響は大きい．多倍長演算を高速化する方法として，条件分岐の削減や演算のマージが提案されている[10][11]．既存研究ではx86プロセッサ上で実験を行なっているが，同様の方法を他のプロセッサにも応用することが可能である．特にARMプロセッサは，携帯電話等の小型機器に実装されており，低消費電力，省メモリといった特徴を持つ楕円曲線暗号が力を発揮すべきプラットフォームである．

ARM9を利用して多倍長演算の実験を行なったところ，多倍長２乗算の速度が遅いことが分かった．原因はメモリアクセスの回数が多いことであった．その問題を解決するために，ARMが得意とするシフト演算と，複数ワードのデータをまとめて転送する命令を利用し，演算のマージを提案する．提案方法はメモリアクセスの回数を８割近く削減し，実行速度を20%以上高速化するという結果を得た．また，同じくARMアーキテクチャを採用しているiPhoneにおいて，提案方法を実験した．従来の実装に対して提案方法は，約80%の高速化を達成した．さらに，最新の楕円加算と楕円２倍算の公式を使って，スカラー倍算を実行したところ，従来の方法に対して63.09%の実行時間に短縮するという結果を得た．

目 次

第1章　序論

1.1　研究の背景

インターネット上で安全な通信を行なう技術として，Secure Socket Layer(SSL) があげられる．SSL は公開鍵暗号を利用した暗号化通信のプロトコルであり，通信する情報の盗聴と改竄を防ぎ，通信する相手のなりすましを防ぐ．従来から SSL に利用されている RSA 暗号と比較して，楕円曲線暗号は小さな鍵サイズ，高速な処理速度，低消費電力，省メモリといった特徴があり，小型機器における利用が注目されている．

現在，世界で広く普及している組込機器用プロセッサの 1 つとして，ARM プロセッサがある．ARM は携帯電話，携帯ゲーム機などに実装されており，楕円曲線暗号の低消費電力，省メモリという特徴が有効である．また，インターネット上で SSL 通信を行なう際に，広く利用されているライブラリは OpenSSL である．本研究では，ARM と OpenSSL を利用して，楕円曲線暗号を効率的に実装する方法を提案する．

1.2　研究の目的と成果

楕円曲線暗号において，最も計算量の大きな部分はスカラー倍算である．よって，楕円曲線暗号の効率的な実装を実現するためには，スカラー倍算を高速化することが重要である．既存研究の中には，x86 プロセッサをターゲットとしたものがあり，その成果は携帯電話等の小型機器に対しても，応用することが可能である．そこでまず初めに，既存研究の成果を ARM プロセッサ上で実験し，問題点を明らかにする．次に問題を解決する方法として，ARM プロセッサが得意とするシフト演算を利用し，従来行なわれていない演算を新たに提案する．最後に，提案方法の実行速度を従来の方法と比較し，スカラー倍算が高速化されることを示す．

1.3　本論文の構成

本論文の構成は以下のとおりである．2 章では，楕円曲線暗号とスカラー倍算アルゴリズムについて述べる．3 章では楕円曲線暗号の効率的な実装に関する既存研究について記述する．4 章では，既存研究の成果を ARM プロセッサと OpenSSL を利用して実験する．

実験の結果として明らかになった問題点に対して，新たな演算の方法を提案する．5 章で結論を述べる．

第2章　準備

2.1　楕円曲線暗号

楕円曲線暗号は，楕円離散対数問題 (ECDLP) に基づいている．ここで，有限体 F_p 上の楕円曲線を E/F_p とする．ただし $p > 3$，p は素数である．E/F_p の Weierstrass 標準形は，Affine 座標系において次のように表すことができる．

$$E/F_p : y^2 = x^3 + ax + b$$
$$(a, b \in F_p,\ 4a^3 = 27b^2 \neq 0)$$

楕円曲線上の点を $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ とし，加法 $P_3 = P_1 + P_2 = (x_3, y_3)$ の計算を，以下のように定義する [1].

$\langle\ P_1 \neq P_2$ の場合 $\rangle$

$$\begin{aligned} \lambda &= \frac{y_2 - y_1}{x_2 - x_1} \\ x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{aligned}$$

$\langle\ P_1 = P_2$ の場合 $\rangle$

$$\begin{aligned} \lambda &= \frac{3x_1^2 + a}{2y_1} \\ x_3 &= \lambda^2 - 2x_1 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{aligned}$$

乗算を M，2 乗算を S，逆元の計算を I とすると，加法の計算量は $2M + S + I$ であり，2 倍算の計算量は $2M + 2S + I$ である．

楕円曲線上の加法から，スカラー倍点 $kP = P + P + ... + P$ を定義することができる．スカラー値 k を，kP と P から求める問題を楕円離散対数問題 (ECDLP) という．現在，ECDLP を効率的に解くアルゴリズムは知られておらず，楕円曲線暗号の安全性は ECDLP に基づいている．

楕円曲線暗号では k を秘密鍵とし，点 P に関してスカラー倍を求める計算を行う．よって楕円曲線暗号を効率的に実装するためには，スカラー倍算を多項式時間で行なうことが重要である．

2.2 Left-to-Right Binary Method

スカラー倍算 nP は Binary Method を使って多項式時間で計算することができる．ここで n は l ビットの整数，P は楕円曲線上の点とする．また，楕円曲線は標準的な Weierstrass 型とする．

表 2.1: Left-to-Right Binary Method On Elliptic Curve

INPUT: $P \in E/F_p$, $n = (n_{l-1} \dots n_0)_2$
OUTPUT: $nP \in E/F_p$
01. $Y \leftarrow P$
02. for $i \leftarrow l-2$ to $i \geq 0$ do
03. $Y \leftarrow \mathrm{ECDBL}(Y)$
04. if $n_i = 1$ then
05. $Y \leftarrow \mathrm{ECADD}(Y, P)$
06. return Y

Left-to-Right Binary Method の計算量は，$l-1$ 回の 2 倍算と，平均 $\frac{1}{2}(l-1)$ 回の加算である．よって 2 倍算を DBL，加算を ADD とすると，計算量は以下のように表すことができる．

$$(l-1)DBL + \frac{(l-1)}{2}ADD \tag{2.1}$$

2.3 Right-to-Left Binary Method

一方，Right-to-Left Binary Method のアルゴリズムは，以下のように表すことができる．Right-to-Left Binary Method の計算量は，l 回の 2 倍算と，平均 $\frac{1}{2}l$ 回の加算である．

$$lDBL + \frac{l}{2}ADD \tag{2.2}$$

表 2.2: Right-to-Left Binary Method

INPUT: $P \in E/F_p$, $n = (n_{l-1} \dots n_0)_2$ OUTPUT: $nP \in E/F_p$
01. $Y \leftarrow O$ 02. $Z \leftarrow P$ 03. for $i \leftarrow 0$ to $i \leq l-1$ do 04. if $n_i = 1$ then 05. $Y \leftarrow \mathrm{ECADD}(Y, Z)$ 06. $Z \leftarrow \mathrm{ECDBL}(Z)$ 07. return Y

2.4 Montgomery's Ladder

Montgomery's Ladder は，楕円曲線上のスカラー倍算を高速化するために提案された方法である [2][3]．また，サイドチャネル攻撃に耐性があるという性質がある．ここで，$L_j = \sum_{i=j}^{l-1} k_i 2^{i-j}$，$H_j = L_j + 1$ とおくと，以下のような観察が得られる．

$$\begin{aligned} L_j &= 2L_{j+1} + k_j \\ &= L_{j+1} + H_{j+1} + k_j - 1 \\ &= 2H_{j+1} + k_j - 2 \end{aligned}$$

この観察に基づいて，Montgomery's Ladder のアルゴリズムは以下のように表すことができる．

$$(L_j, H_j) = \begin{cases} (2L_{j+1}, L_{j+1} + H_{j+1}) \; if \; k_j = 0 \\ (L_{j+1} + H_{j+1}, 2H_{j+1}) \; if \; k_j = 1 \end{cases}$$

単純に計算量だけを考えると，Montgomery's Ladder は l 回の 2 倍算と l 回の加算が必要である．すなわち，$lDBL + lADD$ であるから，Binary Method と比較して計算量が多い．

2.5 2^k-ary algorithm

2^k-ary algorithm [4] はビットをまとめて処理して，乗算の回数を減らす．但し，事前計算として値 $3P, 5P, \dots, (2^k - 1)P$ が必要である．

表 2.3: Montgomery's Ladder

INPUT: $P \in E/F_p$, $n = (n_{l-1} \dots n_0)_2$ OUTPUT: $nP \in E/F_p$
01. $P_1 \leftarrow P$ 02. $P_2 \leftarrow \mathrm{ECDBL}(P)$ 03. for $i \leftarrow l-2$ to $i \geq 0$ do 04. if $n_i = 0$ then 05. $P_2 \leftarrow \mathrm{ECADD}(P_1, P_2)$ 06. $P_1 \leftarrow \mathrm{ECDBL}(P_1)$ 07. else 08. $P_1 \leftarrow \mathrm{ECADD}(P_1, P_2)$ 09. $P_2 \leftarrow \mathrm{ECDBL}(P_2)$ 10. return P_1

表 2.4: 2^k-ary algorithm

INPUT: $P \in E/F_p$, $k \geq 1$, $n = (n_{l-1} \dots n_0)_{2^k}$ 事前計算した値 $3P, 5P, \dots, (2^k-1)P$ u は奇数. OUTPUT: $nP \in E/F_p$
01. $P_1 \leftarrow P$ 02. for $i \leftarrow l-1$ to $i \geq 0$ do 03. for $j \leftarrow 1$ to $k-s$ do 04. $P_1 \leftarrow \mathrm{ECDBL}(P_1)$ 05. $P_1 \leftarrow \mathrm{ECADD}(P_1, uP)$ 06. for $j \leftarrow 1$ to s do 07. $P_1 \leftarrow \mathrm{ECDBL}(P_1)$ 08. return P_1

Example. $k = 4$, $n = (1195770894172030396825 1)_{10}$
$= (1010001000001110101000110000011111110101100101111011100000000011111111111011)_2$
$= (\ 2\ 8\ 8\ 3\ 10\ 8\ 12\ 1\ 15\ 13\ 6\ 5\ 14\ 14\ 0\ 1\ 15\ 15\ 11)_{2^4}$

２進数表記のビット長は $l = 74$ である．Binary method では 112 回の演算が必要であった．すなわち，$39M + 73S$ である．一方，2^k-ary algorithm では 97 回の演算が必要である．すなわち，事前計算として $7M + S$，メインループで $17M + 72S$ である．

2.6 Sliding window method

2^k-ary algorithm を改良した方法が，Sliding window method である．２進数表記で連続する 0 をスキップするため，2^k-ary algorithm と比較して，乗算の回数が少ない．

表 2.5: Sliding window method

INPUT: $P \in E/F_p$，$k \geq 1$，$n = (n_{l-1} \dots n_0)_{2^k}$ 事前計算した値 $3P, 5P, \dots, (2^k - 1)P$ OUTPUT: $nP \in E/F_p$
01. $P_1 \leftarrow P$ 02. for $i \leftarrow l-1$ to $i \geq 0$ do 03. if $n_i = 0$ then 04. $P_1 \leftarrow$ECDBL(P_1) 05. $i \leftarrow l-1$ 06. else 07. $s \leftarrow max\{i-k+1, 0\}$ 08. while $n_s = 0$ do 09. $s \leftarrow s+1$ 10. for $h = 1$ to $i-s+1$ do 11. $P_1 \leftarrow$ECDBL(P_1) 12. $u \leftarrow (n_i \dots n_s)_2$ 13. $P_1 \leftarrow$ECADD(P_1, uP) 14. $i \leftarrow s-1$ 15. return P_1

Example. $k = 4$，$n = (11957708941720303968251)_{10}$
$= (\underline{101}\ 000\ \underline{1}\ 00000\ \underline{111}\ 0\ \underline{101}\ 000\ \underline{11}\ 00000\ \underline{1111}\ \underline{111}$
$0\ \underline{1011}\ 00\ \underline{1011}\ \underline{1101}\ \underline{11}\ 00000000\ \underline{1111}\ \underline{1111}\ \underline{1101}\ \underline{1})_2$
上記の例では，事前計算を含めて 93 回の演算が必要である．すなわち，$21M + 72S$ である．

2.7 NAF representation

2進表記に負の符号を使うことによって，乗算の回数を減らすことができる [5]．2進展開された値に関して，i回連続するビット1を，1に続いて$i-1$回連続する0と末尾の-1に置き換えると，ハミングウェイトを小さくすることができる．ハミングウェイトを最小にする方法として，non-adjacent form(NAF) があげられる [6]．以下，$\overline{1}=-1$である．
Example. $n=(478)_{10}=(1000\overline{1}000\overline{1}0)_{NAF}$

表 2.6: NAF representation

INPUT: $n=(n_l n_{l-1}\dots n_0)_{2^k}$，$n_l=n_{l-1}=0$ OUTPUT: $(n'_{l-1}\dots n'_0)_{NAF}$
1. $c_0 \leftarrow 0$ 2. for $i=0$ to $l-1$ do 3. $\quad c_{i+1} \leftarrow \lfloor (c_i+n_i+n_{i+1})/2 \rfloor$ 4. $\quad n' \leftarrow c_i+n_i-2c_{i+1}$ 5. return $(n'_{l-1}\dots n'_0)_{NAF}$

Example. $n=(1195770894172030396825 1)_{10}$
$=(10100010000100\overline{1}0101001 0\overline{1}0000100000000\overline{1}0\overline{1}0\overline{1}010\overline{1}0000\overline{1}00\overline{1}000000010000000000\overline{1}0\overline{1})_{NAF}$

ここで$k=4$として，Sliding window methodを適用すると，
$n=(\underline{101}\ 000\ \underline{1}\ 0000\ \underline{100\overline{1}}\ 0\ \underline{101}\ 00\ \underline{10\overline{1}}\ 0000\ \underline{1}\ 0000000\ \underline{\overline{1}0\overline{1}}\ 0$
$\underline{\overline{1}01}\ 0\ \underline{\overline{1}}\ 0000\ \underline{\overline{1}00\overline{1}}\ 0000000\ \underline{1}\ 0000000000\ \underline{\overline{1}0\overline{1}})_{NAF}$

となり，事前計算を含めて演算の回数は90回，すなわち$18M+72S$である．

第3章　既存研究

本章では楕円曲線暗号のスカラー倍算の高速化に関する既存研究を示す.

3.1　Algorithms for Multi-exponentiation, 2001 [7]

3.1.1　Abstract

Multi-exponentiation を実行する従来の方法と比較して，提案する方法である Interleaving Exponentiation は効率的である．楕円曲線のように逆元計算の容易な群においては，符号付き 2 進数表記を用いることによって，Interleaving Exponentiation は従来の方法よりも優れている.

3.1.2　OpenSSL におけるスカラー倍算の実装

OpenSSL に実装されているスカラー倍算は，wNAF を利用した Interleaving Exponentiation と呼ばれる方法が実装されている [7]．wNAF(width-w non-adjacent form) とは，サイズ w のウィンドウを持ち，正の整数 n を以下の式のように，符号付き 2 進数に展開したものである.

$$n = \sum_{i=0}^{l-1} n_i 2^i$$

但し，$w > 1$，n_i は 0 または奇数，$|n_i| < 2^{w-1}$ である．全ての正の整数は，唯一の wNAF 表現を持つ.

wNAF representation
入力：正の整数 n，$w > 1$
出力：n の wNAF 表現 $(n_{l-1} \cdots n_0)_{wNAF}$
01: $i \leftarrow 0$
02: while $n > 0$ do
03: if n is odd then
04: $n_i \leftarrow n \mod 2^w$
05: $n \leftarrow n - n_i$
06: else
07: $n_i \leftarrow 0$
08: $n \leftarrow n/2$
09: $i \leftarrow i + 1$
10: return $(n_{l-1} \cdots n_0)_{wNAF}$

Example. $n = 11957708941720303968251$，$w = 4$
$n = (5000100000007000500003000010000000\bar{1}000\bar{5}0003000\bar{1}0007000000010000000000\bar{5})_{wNAF}$

wNAF-Based Interleaving Exponentiation Method
入力：G の元 $g_1, \cdots, g_d$，指数 $e_1, \cdots e_d$ window のサイズ w
出力：$\prod_{i=1}^{d} g_i^{e_i}$
Precomputation:
1: for i from 1 to k do g_i^E
但し E は $1 \leq E \leq 2^{w_i} - 1$ を満たす奇数
Evaluation:
01: $A \leftarrow 1_G$
02: for i from 1 to k do
03: $N_i[b], \cdots, N_i[b_i + 1] \leftarrow 0, \cdots, 0$
04: $N_i[b_i], \cdots, N_i[0] \leftarrow$width$(w_i + 1)$ NAF of e_i
05: for j from b down to 0 do
06: $A \leftarrow A^2$
07: for i from 1 to k do
08: if $N_i[j] \neq 0$ then
09: $A \leftarrow A \cdot g_i^{N_i[j]}$
10: return(A)

Precomputation のコストは，2 乗算が $\#\{i \in \{1, \cdots, k\} | w_i > 1\}$ である．乗算は $(\sum_{1 \leq i \leq k} 2^{w_i - 1}) - k$ 回である．Evaluation のコストは，2 乗算が $b - max_i w_i$ から b 回．乗算は $\sum_{1 \leq i \leq k} b_i \cdot \frac{1}{w_i + 2}$ 回である．

3.2 Faster addition and doubling on elliptic curves, 2008 [8]

3.2.1 Abstract

Edwards は楕円曲線の新しい標準形を発表した．本稿は Edwards 曲線における群の演算の高速な公式を提示する．広範囲にわたって異なる形の楕円曲線と座標系の比較を行なう．上位の演算であるマルチスカラー倍算だけでなく，基本的な演算である 2 倍算や mixed addition，non-mixed addition，unified addition も含む．

3.2.2 OpenSSL における楕円加算公式の実装

OpenSSL に実装されている楕円加算と楕円 2 倍算は，ヤコビアン座標系の加算公式を利用している．乗算を M，2 乗算を S とすると，最新の加算公式 [8] では，楕円加算の計算量は $11M + 5S$，楕円 2 倍算の計算量は $2M + 8S$ である．一方で，OpenSSL において，楕円加算の計算量は $12M + 4S$，楕円 2 倍算の計算量は $4M + 6S$ である．よって，OpenSSL の楕円加算と 2 倍算は高速化することが可能である．

OpenSSL の楕円加算 $(12M + 4S)$
入力：楕円曲線上の点 $(X1, Y1, Z1)$，$(X2, Y2, Z2)$ 出力：楕円曲線上の点 $(X3, Y3, Z3)$ $= (X1, Y1, Z1) + (X2, Y2, Z2)$
01: $U1 = X1 \times Z2^2$
02: $U2 = X2 \times Z1^2$
03: $S1 = Y1 \times Z2^3$
04: $S2 = Y2 \times Z1^3$
05: if $(U1 = U2)$ then
06: 　　if $(S1 = S2)$ then
07: 　　　　return 2 倍算 $(X1, Y1, Z1)$
08: 　　else
09: 　　　　return 無限遠点
10: $H = U1 - U2$
11: $R = S1 - S2$
12: $H' = U1 + U2$
13: $R' = S1 + S2$
14: $X3 = R^2 - H^2 \times H'^2$
15: $Y3 = R \times (H^2 \times H' - 2 \times X3) - R' \times H^3$
16: $Z3 = H \times Z1 \times Z2$
17: return $(X3, Y3, Z3)$

OpenSSL の楕円２倍算 $(4M + 6S)$

入力：楕円曲線上の点 (X, Y, Z)
出力：楕円曲線上の点 (X', Y', Z')
$= (X, Y, Z) + (X, Y, Z)$

01: if $(Y = 0)$ then
02: 　　return 無限遠点
03: $n1 = 3 \times X^2 + a \times Z^4$
03: $n2 = 4 \times X \times Y^2$
05: $X' = n1^2 - 2 \times n2$
06: $Y' = n1 \times (n2 - X') - 8 \times Y^4$
07: $Z' = 2 \times Y \times Z$
08: return (X', Y', Z')

最新の楕円加算 $(11M + 5S)$

入力：楕円曲線上の点 $(X1, Y1, Z1)$，$(X2, Y2, Z2)$
出力：楕円曲線上の点 $(X3, Y3, Z3)$
$= (X1, Y1, Z1) + (X2, Y2, Z2)$

01: $U1 = X1 \times Z2^2$
02: $U2 = X2 \times Z1^2$
03: $S1 = Y1 \times Z2^3$
04: $S2 = Y2 \times Z1^3$
05: if $(U1 = U2)$ then
06: 　　if $(S1 \neq S2)$ then
07: 　　　　return 無限遠点
08: 　　else
09: 　　　　return ２倍算 $(X1, Y1, Z1)$
10: $H = U2 - U1$
11: $R = 2(S2 - S1)$
12: $I = (2H)^2$
13: $J1 = IH$
14: $J2 = IU2$
15: $X3 = R^2 - J1 - 2J2$
16: $Y3 = R(J2 - X3) - 2S1J1$
17: $Z3 = ((Z1 + Z2)^2 - Z1^2 - Z2^2)H$
18: return $(X3, Y3, Z3)$

最新の楕円２倍算 $(2M + 8S)$
入力：楕円曲線上の点 (X, Y, Z) 出力：楕円曲線上の点 (X', Y', Z') $= (X, Y, Z) + (X, Y, Z)$
01: if $(Y = 0)$ then 02: return 無限遠点 03: $S = 4XY^2 = 2((X + Y^2)^2 - X^2 - Y^4)$ 04: $M = 3X^2 + aZ^4$ 05: $X' = M^2 - 2S$ 06: $Y' = M(S - X') - 8Y^4$ 07: $Z' = 2YZ = (Y + Z)^2 - Y^2 - Z^2$ 08: return (X', Y', Z')

表 3.1: 計算量の比較

	楕円加算	楕円２倍算
OpenSSL	$12M + 4S$	$4M + 6S$
最新版 [8]	$11M + 5S$	$2M + 8S$

3.3 Incomplete Reduction in Modular Arithmetic, 2002 [9]

3.3.1 Abstract

任意の長さの任意の素数 p を法とする有限体 $GF(p)$ において，高速な算術演算をソフトウェアによって実装する新しい方法を示す．提案方法の最も重要な特徴は，処理の低速なビットレベルの演算を回避し，処理の高速なワードレベルの演算を実行するようにしたことである．提案方法は，有限体 $GF(p)$ 上に定義された公開鍵暗号アルゴリズムに対して応用することが可能であり，特に楕円曲線デジタル署名アルゴリズムに対する応用が注目される．

3.3.2 Incomplete Reduction(IR)

論文 [9] では，Modular Reduction を効率よく行なう方法として，Incomplete Reduction(IR) が提案されている．IR は高速に Mod 演算を行なうために，ビット単位ではなく，ワード単位で計算を行なっている．ここで，以下のように定義する．

- w: コンピューターのワード数，すなわち $w \in [8, 16, 32, 64]$
- p: 素数
- k: p のビット数，すなわち $k = \lceil log_2 p \rceil$
- s: p のワード数，すなわち $s = \lceil \frac{k}{w} \rceil$
- m: s ワードのビット数，すなわち $m = sw$
- C: Completely Reduced Numbers,
  $C = \{0, 1, \ldots, (p-1)\}$
- I: Incompletely Reduced Numbers,
  $I = \{0, 1, \ldots, p-1, p, p+1, \ldots, (2^m - 1)\}$

Modular Addition を通常の Reduction で行なう場合と，IR を使う場合のアルゴリズムを表に示し比較する．IR の方がステップ数が少ないことが分かる．

表 3.2: Modular Addition with Complete Reduction

INPUT: integers $a, b \in [0, p-1]$, $c < 2^w$ $p = 2^m - c$, $m = sw$, where $n, s, w \in Z^+$ OUTPUT: $r = a + b(mod\ p)$
01. $carry = 0$ 02. for i from 0 to $s-1$ do 03. $(carry, r[i]) \leftarrow a[i] + b[i] + carry$ 04. if $carry = 1$ then 05. $carry = 0$ 06. $(carry, r[0]) \leftarrow r[0] + c$ 07. for i from 1 to $s-1$ do 08. $(carry, r[i]) \leftarrow a[i] + b[i] + carry$ 09. else 10. $borrow = 0$ 11. for i from 0 to $s-1$ do 12. $(borrow, R[i]) \leftarrow r[i] - p[i] - borrow$ 13. if $borrow = 0$ 14. $r \leftarrow R$ 15. return r

Modular Multiplication は，Montgomery Modular Multiplication に対して IR を適用しても，ほとんど効果が得られないことが分かっている [9]．その理由は，Montgomery

表 3.3: Modular Addition with Imcomplete Reduction

INPUT: integers $a, b \in [0, p-1]$, $c < 2^w$ $p = 2^m - c$, $m = sw$, where $n, s, w \in Z^+$ OUTPUT: $r = a + b(mod\ 2^m)$
01. $carry = 0$ 02. for i from 0 to $s-1$ do 03. $(carry, r[i]) \leftarrow a[i] + b[i] + carry$ 04. if $carry = 1$ then 05. $carry = 0$ 06. $(carry, r[0]) \leftarrow r[0] + c$ 07. for i from 1 to $s-1$ do 08. $(carry, r[i]) \leftarrow a[i] + b[i] + carry$ 09. return r

表 3.4: Modular Subtraction with Complete Reduction

INPUT: integers $a, b \in [0, p-1]$, $c < 2^w$ $p = 2^m - c$, $m = sw$, where $n, s, w \in Z^+$ OUTPUT: $r = a + b(mod\ p)$
01. $borrow = 0$ 02. for i from 0 to $s-1$ do 03. $(borrow, r[i]) \leftarrow a[i] - b[i] - borrow$ 04. if $borrow = 1$ then 05. $carry = 0$ 06. for i from 0 to $s-1$ do 07. $(carry, r[i]) \leftarrow r[i] + p[i] + carry$ 08. return r

Modular Multiplication の Complete Reduction と Incomplete Reduction の違いがほとんど無いからである.

表 3.5: Modular Division by 2 with Complete Reduction

INPUT: integers $a, b \in [0, 2^m - 1]$, $c < 2^w$ $p = 2^m - c$, $m = sw$, where $n, s, w \in Z^+$ OUTPUT: $r = a/2 (mod\ p)$
01. $carry = 0$ 02. if a is odd then 03. for i from 0 to $s - 1$ do 04. $(carry, r[i]) \leftarrow a[i] + p[i] + carry$ 05. $(carry, r[s-1]) \leftarrow (carry, r[s-1])/2$ 06. for i from $s - 2$ to 0 do 07. $(carry, r[i]) \leftarrow (carry, r[i])/2$ 08. $borrow = 0$ 09. for i from 0 to $s - 1$ do 10. $(borrow, R[i]) \leftarrow r[i] - p[i] - borrow$ 11. if $borrow = 0$ 12. $r \leftarrow R$ 13. return r

3.4 Efficient Techniques for High-Speed Elliptic Curve Cryptography, 2010 [10], Analysis of Efficient Techniques for Fast Elliptic Curve Cryptography on x86-64 based Processors, 2010 [11]

3.4.1 Abstract

本稿では，近年新たに普及してきた x86-64 アーキテクチャにおける，楕円曲線の点の乗算の計算を高速化するための技術に関して，その効率性を計測する実験データを分析し，提示する．とりわけ，F_p 上の高速な体の演算を実現するため，条件分岐の削減と Incomplete Reduction を組み合わせた場合の効率性を研究する．さらに，x86-64 上でデータ依存性がある場合の影響を調べ，パイプラインのストールを減らしたり，メモリの読み書きの回数を減らしたり，関数呼び出しの回数を減らしたりするための，一般的な方法を提案する．結果として，x86-64 アーキテクチャにおける既存研究の最善の結果に比べて 31%高速な，点の乗算の実装を得ることができた．

表 3.6: Modular Division by 2 with Imcomplete Reduction

INPUT: integers $a, b \in [0, 2^m - 1]$, $c < 2^w$ $p = 2^m - c$, $m = sw$, where $n, s, w \in Z^+$ OUTPUT: $r = a/2 (mod\ 2^m)$
01. $carry = 0$ 02. if a is odd then 03. for i from 0 to $s - 1$ do 04. $(carry, r[i]) \leftarrow a[i] + p[i] + carry$ 05. $(carry, r[s-1]) \leftarrow (carry, r[s-1])/2$ 06. for i from $s - 2$ to 0 do 07. $(carry, r[i]) \leftarrow (carry, r[i])/2$ 08. return r

3.4.2 Elimination of Conditional Branch

条件分岐は実行速度の遅れにつながる．CPU は分岐予測が外れると，パイプラインをフラッシュするために，数クロックを無駄に費やすことになるからである．条件分岐を取り除く方法として，look-up table を利用する方法がある [11][10]．例えば，C 言語で条件分岐の一つである For ループを，i= 1 down to 100 として１００回繰り返す場合を考える．ループの内部で配列の要素を参照するときに，array[i] というアドレス計算を行なうが，この計算は，x86 アーキテクチャのアセンブリ言語を利用して省くことが可能である．すなわち，アセンブリ言語を利用することによって，ループカウンタを増加させながら条件分岐を行なう必要はなくなる．

3.4.3 Minimizing Data Dependencies

プログラム中の連続するデータに依存関係があると，CPU のパイプラインにストールが発生するため，処理速度が遅くなる．

Corollary 1. I_i と I_j を各々 *write* 命令，*read* 命令とする．データ依存関係 $W(I_i) \cap R(I_j) \neq \phi$ において，$i < j$ である．また，非スーパースカラーパイプラインアーキテクチャにおいて，I_i と I_j は各々，i サイクル目，j サイクル目に実行される．このとき，$\rho = j - i < \delta_{write}$ ならば，パイプラインは少なくとも $(\delta_{write} - \rho)$ サイクルだけストールする．(δ_{write}) は，*write* 命令 I_i のフェッチから，パイプラインレイテンシを完了させるために必要なサイクル数を表す．

Definition 1. *2* つの体の演算 $OP_i(op_m, op_n, res_p)$ と $OP_j(op_r, op_s, res_t)$ は，$i < j$ かつ $res_p = op_r$ または $res_p = op_s$ の場合，体の算術レベルにおいてデータ依存であるとい

う．ここで，OP_i と OP_j はプログラムの実行中に i 番目と j 番目の位置で体の演算が行なわれることを表す．また，op と res はそれぞれ入力と結果を保持するレジスターである．$j-i=1$ ならば，体の算術レベルにおいて連続的データ依存であるという．すなわち，OP_i と OP_j は実行順において連続している．

依存関係を取り除くために，以下 3 つの方法がある．

- Field Arithmetic Scheduling
  体の演算をスケジューリングすることによって，データ依存関係にある演算を連続的ではなくする方法である．どの演算も，実行レイテンシー δ_{ins} は write 命令のレイテンシーよりも長いと仮定する．すなわち，$\delta_{ins} > \delta_{write}$ である．したがって，データ依存関係のない命令を 2 つの連続する命令間に挿入することによって，新たな相対位置 $\rho_{new,x}$ を確保して，$\rho_{new,x} = \rho_x + \delta_{ins} > \delta write$ を達成することができる．ここで，ρ_x は元の相対位置である．

- Merging Point Operations
  この方法は，体の演算をスケジューリングすることによって得られる効果を補うものである．点の演算の内部で，データ依存性の高い体の演算を行なう場合，次に実行する体の演算を挿入することによって，データ依存性を低くすることができる．w-NAF 法など高速なスカラー倍算を実行する際には，楕円曲線上の 2 倍算命令が連続して実行されるので，その連続回数分の 2 倍算を一つにまとめた関数を作成することによって，データ依存性を低くすることが可能である．その副作用的効果として，関数の呼び出し回数が減り，さらに実行速度の向上が見込める．

- Merging Field Operations
  複数ある体の演算を一つにまとめることによって，さらなる速度の向上が可能である．この方法が有効になるのは，以下 2 つの場合である．まず初めに，ある体の演算の出力が，続く演算の入力として必要な場合である．この場合は，データ依存性を解消することが可能である．次に，入力が複数の命令によって必要とされる場合である．この場合は，メモリの読み書き回数を削減することが可能である．論文 [11] において提案されているものは，$a-2b\ (mod\ p)$，$a+a+a\ (mod\ p)$，$a-b\ (mod\ p)$，$(a-b)-2c\ (mod\ p)$ である．

第4章　実験

4.1　ARMプロセッサ

ARMは主に組込機器に用いられる32bit RISCプロセッサである[12]．低消費電力であることから，携帯電話やPDA，携帯ゲーム機に採用され，世界で100億個以上出荷されているプロセッサである[13]．ARM命令セットの特徴として，条件実行，シフタオペランド，Sビット指定，複数レジスタ-メモリ間転送があげられる．

まず初めに条件実行であるが，ARMの全ての命令は状態レジスタのフラグに合わせて条件付きで実行することが可能である．この機能によって，分岐命令を省き，パイプラインの乱れを無くし，ストールを削減することができる．一方で，ARMは他のRISCアーキテクチャにみられる分岐遅延スロットが無い．分岐遅延スロットは，分岐命令の直後の命令を実行することにより，パイプラインの乱れを防ぐ方法であるが，ARMにおいては，パイプラインの乱れを防ぐ方法として，条件実行を採用しているのである．

次にシフタオペランドであるが，ARMのデータ処理命令では，データの演算や転送を行なったついでに，レジスタの値を算術シフト，左右論理シフト，ローテートさせることが可能である．シフトのビット数を指定する機能をシフタオペランドという．シフタオペランドにより，命令フェッチやメモリアクセスの回数を削減することが可能である．一方で，ARMは除算命令を持たない．２の乗数で除算をする場合は，シフタオペランドを利用することによって，効率的な演算が可能である．しかし，それ以外の値で除算する場合は，C言語のライブラリ等ソフトウェアによる除算ルーチンを実行するか，ハードウェアに除算器を追加する必要がある．

さらにSビット指定であるが，ARMのデータ処理命令では，データの演算や転送を行なった結果を，状態レジスタに反映するかどうか選択することができる．ニモニックにSを付けることによって，状態レジスタが更新される．Sが付かない場合は，状態レジスタは何も影響を受けない．この機能によって，加算や減算を効率よく実行することが可能である．例えば，加算命令はデータの加算処理以外にも，アドレスを更新する際に用いられる．加算命令の結果が毎回状態レジスタに反映される場合，データの加算処理だけでなく，アドレス更新処理を行なったときにも，キャリーフラグやゼロフラグが影響を受けてしまう．アドレス更新処理の結果を状態レジスタに反映しないために，状態レジスタをメモリに退避して，必要な時に復帰する手間が発生する．ARMのSビット指定機能によって，このような手間を省くことが可能である．

最後に複数レジスタ-メモリ間転送であるが，ARMのLDM/STM命令を実行すると，一

度のメモリアクセスで複数ワードのデータを転送することが可能である．一般的に ARM プロセッサが実装されている機器はメモリの転送速度が遅く，メモリにアクセスする回数が多くなれば，それだけ実行速度が遅くなってしまう．しかし，複数レジスタ-メモリ間転送の仕組みを利用することによって，メモリアクセスの回数を減らすことが可能である．

以上の特徴を利用し，x86 プロセッサに関する既存研究 [11][10] で提案されている，条件分岐の削減と演算のマージという手法を，ARM プロセッサにも応用することが可能である．ここでは，ARM プロセッサにおいて条件分岐の削減と演算のマージを実験し，成果を明らかにする．

4.2 OpenSSL

インターネット上で SSL 通信を行なう際に，広く利用されているライブラリが OpenSSL である．OpenSSL において，スカラー倍算は EC_POINT_mul という関数に実装されており，そのアルゴリズムは wNAF を利用した Interleaving Exponentiation である [7].

EC_POINT_mul の内部で呼び出される楕円曲線上の演算は，EC_POINT_dbl と EC_POINT_add であり，それぞれ Jacobian 座標系の加算公式が実装されている．次に，EC_POINT_dbl と EC_POINT_add の内部では，多倍長 mod 演算の ec_GFp_mont_field_mul と ec_GFp_mont_field_sqr が呼び出されている．さらに，ec_GFp_mont_field_mul と ec_GFp_mont_field_sqr の内部で呼び出される多倍長演算は，BN_mul，BN_sqr，BN_from_montgomery_word である．最後に，BN_mul，BN_sqr，BN_from_montgomery_word の内部では，多倍長演算をアセンブリ言語で最適化した，小文字の bn で始まる関数が呼び出されている．

表 4.1: OpenSSL で実装されている関数の階層構造

関数名	機能
EC_POINT_mul	楕円スカラー倍算
EC_POINT_add	楕円加算
EC_POINT_dbl	楕円２倍算
ec_GFp_mont_field_mul	多倍長 mod 乗算
ec_GFp_mont_field_sqr	多倍長 mod ２乗算
BN_mul	多倍長乗算
BN_sqr	多倍長２乗算
BN_from_montgomery_word	多倍長 mod 還元
bn_mul_normal	多倍長演算
bn_mul_words	
bn_mul_add_words	
bn_sqr_normal	
bn_sqr_words	
bn_add_words	
bn_sub_words	

4.3 多倍長演算の高速化

4.3.1 OpenSSL に実装されている乗算と２乗算

OpenSSL において，楕円曲線上の演算が行なわれるときに利用される多倍長演算は，筆算の要領で行なわれる教科書的なアルゴリズムである．160bit から 224bit の演算は，Karatsuba 乗算や高速フーリエ変換アルゴリズムを利用するには bit 長が短すぎて，かえって低速になってしまうためであると考えられる [1].

アルゴリズム 17　Schoolbook multiplication

入力：m word の整数 $u = (u_{m-1} \cdots u_0)_b$
　　n word の整数 $v = (v_{n-1} \cdots v_0)_b$
出力：$(m+n)$ word の整数 $uv = w$
　　$= (w_{m+n-1} \cdots w_0)_b$

```
for i = 0 to n − 1 do w_i ← 0
for i = 0 to n − 1 do
    k ← 0
    if v_i = 0 then w_{m+i} ← 0
    else
        for j = 0 to m − 1 do
            t ← v_i u_j + w_{i+j} + k
            w_{i+j} ← t  mod b
            k ← ⌊t/b⌋
        w_{m+i} ← k
return (w_{m+n−1} ··· w_0)_b
```

アルゴリズム 18　Schoolbook Squaring

入力：n word の整数 $u = (u_{m-1} \cdots u_0)_b$
出力：$(2n)$ word の整数 $u^2 = w = (w_{2n-1} \cdots w_0)_b$

```
for i = 0 to 2n − 1 do w_i ← 0
for i = 0 to n − 1 do
    t ← u_i^2 + w_{2i}
    w_{2i} ← t  mod b
    k ← ⌊t/b⌋
    for j = i + 1 to n − 1 do
        t ← 2u_i u_j + w_{i+j} + k
        w_{i+j} ← t  mod b
        k ← ⌊t/b⌋
    w_{i+n} ← k
return (w_{2n−1} ··· w_0)_b
```

多倍長乗算の計算量は，整数のワード数を n とすると，n^2 回の 1 ワード乗算を行なう．一方，多倍長 2 乗算の計算量は，$(n^2+n)/2$ である．160bit は 1 ワード 32bit とすると，5 ワードである．よって乗算の計算量の理論値は $5^2 = 25$ である．また 160bit の 2 乗算は，$(5^2+5)/2 = 15$ であり，計算量は乗算の 0.6 倍である．理論的には 2 乗算のほうが高速であることが分かる．以下の例をみると，乗算が重複しているので省略することが可能である．

ex. $a[5] \times a[5]$

				a4	a3	a2	a1	a0
			$\times$)	a4	a3	a2	a1	a0
				$a0a4$	$a0a3$	$a0a2$	$a0a1$	$a0a0$
			$a1a4$	$a1a3$	$a1a2$	$a1a1$	$a1a0$	
		$a2a4$	$a2a3$	$a2a2$	$a2a1$	$a2a0$		
	$a3a4$	$a3a3$	$a3a2$	$a3a1$	$a3a0$			
$a4a4$	$a4a3$	$a4a2$	$a4a1$	$a4a0$				
				$2 \cdot a0a4$	$2 \cdot a0a3$	$2 \cdot a0a2$	$2 \cdot a0a1$	
			$2 \cdot a1a4$	$2 \cdot a1a3$	$2 \cdot a1a2$			
		$2 \cdot a2a4$	$2 \cdot a2a3$					
	$2 \cdot a3a4$							
$a4a4$		$a3a3$		$a2a2$		$a1a1$		$a0a0$

4.3.2 C言語による実装

以下は，C言語によるOpenSSLのソースコードの内容である．ソースコードの詳細は付録に記す．多倍長乗算は，関数bn_mul_normal()である．その内部では，積算bn_mul_words()を実行した後に，積和bn_mul_add_words()を繰り返すことによって実装されている．

■ bn_mul_normal(): $r = a \times b$ を求める
・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数aのアドレス
int na: 多倍長変数aのワード数
BN_ULONG *b: 多倍長変数bのアドレス
int nb: 多倍長変数bのワード数
・output:
戻り値なし．

多倍長2乗算は，関数bn_sqr_normal()である．その内部では，多倍長乗算に比べて少ない回数の積和bn_mul_add_words()を繰り返した後に，和算bn_add_words()と2乗算bn_sqr_words()を実行することによって実装されている．

■ bn_sqr_normal(): $r = a \times a$ を求める
・input:
BN_ULONG *r: 計算結果を格納する開始アドレス

BN_ULONG *a: 多倍長変数 a のアドレス
int n: 多倍長変数 a のワード数
BN_ULONG *tmp: 計算用領域のアドレス

・output:
戻り値なし.

C 言語による実装をした場合の実行速度を計測する．実行環境は，Mac OS X 10.5.3, Intel Core 2 Duo 2.1 GHz，SDRAM 2GHz 667 MHz DDR2，GCC 4.0.1 である．計測の方法は以下のとおりである．

- 160bit の乱数 x, y を 100 組用意する.
- 実行時間の計測は，rdtsc 命令を使用し，クロック数を取得する.
- 乗算の場合は各組に対して 10000000 回 $x \times y$ の演算を行ない，実行時間を計測する.
- 2 乗算の場合は各組に対して 10000000 回 $x \times x$ の演算を行ない，実行時間を計測する.
- 計測したクロック数の合計を処理回数（10000000）で割り，1 回当たりのクロック数を計算する.
- 100 組に対して計測を行ない，各組の平均クロック数を計算する.

計測に用いるソースコードの例を示す.

ソースコード 4.1: Rdtsc

```
1  #define LOOP_AVE          100
2  #define LOOP_SUM          10000000
3
4  long long exec_rdtsc(void);
5  asm("_exec_rdtsc:");
6  asm("                    rdtsc");
7  asm("                    ret");
8
9  int i,j;
10 unsigned long long r0, r1, llsum, llave;
11
12 llave = 0;
13 for(j = 0; j < LOOP_AVE; j++)
14 {
15     llsum = 0;
16     for(i = 0; i < LOOP_SUM; i++)
17     {
```

```
18          r0 = exec_rdtsc();
19          func();
20          r1 = exec_rdtsc();
21          llsum += r1-r0;
22
23      }
24      llave += llsum/LOOP_SUM;
25  }
26
27  printf("func: %d\n", llave/LOOP_AVE);
28  printf("\n");
```

以下の表は，実験の結果である．bn_mul_normal の速度は 557 クロック，bn_mul_normal の中で呼び出された関数は bn_mul_words が 1 回，bn_mul_add_words が 4 回であり，それぞれ実行速度は 106 クロック，131 クロックである．

bn_sqr_normal の速度に関しても同様に，bn_sqr_normal は 497 クロック，その内部で呼び出された関数は，bn_mul_words，bn_mul_add_words，bn_add_words，bn_sqr_words であり，それぞれの速度を記してある．

表 4.2: bn_mul_normal の速度 (C 言語)

関数	クロック数
bn_mul_normal	557
bn_mul_words	106
bn_mul_add_words	131
bn_mul_add_words	131
bn_mul_add_words	131
bn_mul_add_words	131

表 4.3: bn_sqr_normal の速度 (C 言語)

関数	クロック数
bn_sqr_normal	497
bn_mul_words	98
bn_mul_add_words	86
bn_mul_add_words	79
bn_mul_add_words	62
bn_add_words	123
bn_sqr_words	57
bn_add_words	124

4.3.3 x86 アセンブラによる実装

OpenSSL では，x86 系のコンピュータ用にアセンブラのソースコードが付属している．これは関数 bn_mul_words()，bn_mul_add_words()，bn_sqr_words()，bn_add_words() 等の演算を高速化したものである．

■ bn_mul_words(): $rp = ap \times w$ を求める
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_mul_add_words(): 積和 $rp = rp + ap \times w$ を求める
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_sqr_words(): 各ワードの 2 乗を求める
・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
int n: 多倍長変数 a のワード数
・output:
戻り値 なし

■ bn_add_words(): $r = a + b$ を求める
・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
BN_ULONG *b: 多倍長変数 b のアドレス
int n: 多倍長変数 a または b の大きな方のワード数
・output:

戻り値 BN_ULONG: 繰り上がり値

4.3.4 問題点

以下はC言語版と同じ要領で計測したアセンブラの実行速度である．

表 4.4: bn_mul_normal の速度 (アセンブラ)

関数	クロック数
bn_mul_normal	293
bn_mul_words	68
bn_mul_add_words	74
bn_mul_add_words	74
bn_mul_add_words	74
bn_mul_add_words	74

表 4.5: bn_sqr_normal の速度 (アセンブラ)

関数	クロック数
bn_sqr_normal	352
bn_mul_words	63
bn_mul_add_words	60
bn_mul_add_words	54
bn_mul_add_words	49
bn_add_words	100
bn_sqr_words	58
bn_add_words	99

２乗算は乗算よりも遅いことが分かる．関数bn_add_words() が遅いからである．bn_add_words() のソースコードをみると，命令数が多く，またジャンプ命令も多いが，これはどのようなワード数の変数に対しても演算を行なうことができるようになっているからである．つまり，計算するワード数によって場合分けが生じ，分岐を増やさなければならない．また，長いビット長の演算をする際には，加算と減算を使用してアドレスの更新をするため，計算途中で発生したキャリーが捨てられてしまう．そこで，キャリーを空いてい

るレジスタに保持し，必要に応じて取り出す手間が増えてしまい，結果として命令数が増えるのである．

4.3.5 改善案

条件分岐を削減し，キャリーのロードとストアを削減する方法として，楕円曲線上の演算でよく使用される 160bit 専用の関数を提案する．160bit と決まっていればワード数が固定されるので，条件分岐もキャリーのロードとストアも必要がなくなる．ソースコードは付録に示す．

■ bn_add_10_words(): 10 words の値 $r = a + b$ を求める
・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
BN_ULONG *b: 多倍長変数 b のアドレス
・output:
戻り値 なし

以下の表は，改善案の実行速度である．条件分岐を減らすことによって，実行時間は約 59%まで削減された．

表 4.6: bn_sqr_normal の速度 (アセンブラ改)

関数	クロック数	改善比率
bn_mul_words	63	-
bn_mul_add_words	60	-
bn_mul_add_words	54	-
bn_mul_add_words	49	-
bn_add_10_words	59	59.00%
bn_sqr_words	58	-
bn_add_10_words	58	58.58%

さらに，160bit 専用の bn_mul_words，bn_add_mul_words，bn_sqr_words を作成することによって，bn_sqr_normal を高速化することが可能である．ソースコードは付録に示す．

■ bn_mul_4_words(): $rp = ap \times w$ を求める
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_mul_add_3_words(): 積和 $rp = rp + ap \times w$ を求める
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_mul_add_2_words(): 積和 $rp = rp + ap \times w$ を求める
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_mul_add_1_word(): 積和 $rp = rp + ap \times w$ を求める
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_sqr_5_words(): 各ワードの２乗を求める
・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
・output:
戻り値 なし

以下の表は，改善した bn_sqr_normal の実行速度である．

表 4.7: bn_sqr_normal の速度 (アセンブラ改２)

関数	クロック数	改善比率
bn_mul_4_words	55	87.30%
bn_mul_add_3_words	55	91.66%
bn_mul_add_2_words	49	90.74%
bn_mul_add_1_word	46	93.87%
bn_add_10_words	59	59.00%
bn_sqr_5_words	54	93.10%
bn_add_10_words	58	58.58%

同様の改良を bn_mul_normal に対しても行なうことができる．160bit 専用の関数 bn_mul_160 と bn_sqr_160 を作成し，それぞれ bn_mul_normal，bn_sqr_normal と置き換えて速度を計測した．

表 4.8: bn_mul_160 の速度 (アセンブラ改３)

関数	クロック数	改善比率
bn_mul_160	238	67.61%
bn_mul_5_words	59	86.76%
bn_mul_add_5_words	69	93.24%
bn_mul_add_5_words	69	93.24%
bn_mul_add_5_words	69	93.24%
bn_mul_add_5_words	69	93.24%

表 4.9: bn_sqr_160 の速度 (アセンブラ改３)

関数	クロック数	改善比率
bn_sqr_160	215	61.07%
bn_mul_4_words	55	87.30%
bn_mul_add_3_words	55	91.66%
bn_mul_add_2_words	49	90.74%
bn_mul_add_1_word	46	93.87%
bn_add_10_words	59	59.00%
bn_sqr_5_words	54	93.10%
bn_add_10_words	58	58.58%

4.3.6 ARM における多倍長２乗算の実装

x86 と同様に，ARM アセンブラを使って多倍長２乗算を実装する．

■ bn_mul_words(): $rp = ap \times w$ を求める
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_mul_add_words(): 積和 $rp = rp + ap \times w$ を求める
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_sqr_words(): 各ワードの２乗を求める
・input:

BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
int n: 多倍長変数 a のワード数
・output:
戻り値 なし

■ bn_add_words(): $r = a + b$ を求める
・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
BN_ULONG *b: 多倍長変数 b のアドレス
int n: 多倍長変数 a または b の大きな方のワード数
・output:
戻り値 BN_ULONG: 繰り上がり値

ARM による実装をした場合の実行速度を計測する．使用した CPU は ARM9 である．計測の方法は以下のとおりである．

- 160bit の乱数 x, y を 100 組用意する．
- 実行時間の計測は，tick 命令を使用し，チック数（クロックの 64 分周）を取得する．
- 乗算の場合は各組に対して 10000000 回 $x \times y$ の演算を行ない，実行時間を計測する．
- 2 乗算の場合は各組に対して 10000000 回 $x \times x$ の演算を行ない，実行時間を計測する．
- 計測したクロック数の合計を処理回数（10000000）で割り，1 回当たりのクロック数を計算する．
- 100 組に対して計測を行ない，各組の平均クロック数を計算する．

計測に用いるソースコードの例を示す．

ソースコード 4.2: ArmRdtsc

```
1 #define LOOP_AVE        100
2 #define LOOP_SUM        10000000
3
4 static Tick t0, t1;
5 static u64 sum, ave;
```

```
6
7  ave = 0;
8  for(int k = 0; k < LOOP_AVE; k++)
9  {
10     sum = 0;
11     for(int m = 0; m < LOOP_SUM; m++)
12     {
13         t0 = GetTick();
14         func();
15         t1 = GetTick();
16         sum += (t1-t0);
17     }
18     ave += sum;
19 }
```

理論値では，多倍長２乗算の方が多倍長乗算よりも高速である．しかし，OpenSSL の多倍長２乗算を ARM で実行すると，多倍長乗算よりも遅い．以下は ARM 版 C 言語とアセンブラの実行速度である．x86 と異なり，C 言語で作成した場合も，bn_sqr_normal の方が遅い．また，単純に x86 のアセンブリコードを移植した場合も，bn_sqr_normal の方が遅い．

表 4.10: bn_mul_normal の速度 (ARM 版 C 言語)

関数	チック数
bn_mul_normal	9.28274
bn_mul_words	2.14097
bn_mul_add_words	2.29695
bn_mul_add_words	2.29695
bn_mul_add_words	2.29695
bn_mul_add_words	2.29695

表 4.11: bn_sqr_normal の速度 (ARM 版 C 言語)

関数	チック数
bn_sqr_normal	9.39272
(mul との比較)	(101.18%))
bn_mul_words	1.87529
bn_mul_add_words	1.73540
bn_mul_add_words	1.46898
bn_mul_add_words	1.21894
bn_add_words	2.29917
bn_sqr_words	1.73655
bn_add_words	2.29917

4.3.7 問題点

まず初めに，x86 で行なった高速化と同じように，160 ビット専用の関数を用意した．これによって分岐の回数を減らし，キャリーの代入処理を減らした．しかし，それでもなお２乗算は乗算よりも遅かった．以下は速度計測の結果である．

表 4.12: bn_mul_normal の速度 (ARM アセンブラ)

関数	チック数
bn_mul_normal	5.43837
bn_mul_words	1.42210
bn_mul_add_words	1.51721
bn_mul_add_words	1.51721
bn_mul_add_words	1.51721
bn_mul_add_words	1.51721

表 4.13: bn_sqr_normal の速度 (ARM アセンブラ)

関数	チック数
bn_sqr_normal	5.89165
(mul との比較)	(108.33%)
bn_mul_words	1.32802
bn_mul_add_words	1.29697
bn_mul_add_words	1.18769
bn_mul_add_words	1.07817
bn_add_words	1.46899
bn_sqr_words	1.29578
bn_add_words	1.46898

表 4.14: bn_mul_160 の速度 (ARM アセンブラ)

関数	チック数
bn_mul_160	3.76684
bn_mul_5_words	1.12517
bn_mul_add_4_words	1.25021
bn_mul_add_4_words	1.25021
bn_mul_add_4_words	1.25021
bn_mul_add_4_words	1.25021

表 4.15: bn_sqr_160 の速度 (ARM アセンブラ)

関数	チック数
bn_sqr_160	4.04783
(mul との比較)	(107.46%)
bn_mul_4_words	1.06266
bn_mul_add_3_words	1.07844
bn_mul_add_2_words	1.00016
bn_mul_add_1_words	0.9636
bn_add_10_words	1.26595
bn_sqr_5_words	1.09392
bn_add_10_words	1.26563

関数 bn_sqr_160 のアルゴリズムは，乗算回数を減らすことによって高速化を図っている．乗算をしている部分を抜き出して比較してみると，高速化されていることが分かる．よって問題点は，乗算以外の部分，特に関数 bn_add_10_words である．

表 4.16: bn_mul_160 の乗算部分の速度 (ARM アセンブラ)

関数	チック数
bn_mul_5_words	1.12517
bn_mul_add_5_words	1.25021
bn_mul_add_5_words	1.25021
bn_mul_add_5_words	1.25021
bn_mul_add_5_words	1.25021

表 4.17: bn_sqr_160 の乗算部分の速度 (ARM アセンブラ)

関数	チック数
bn_mul_4_words	1.06266
bn_mul_add_3_words	1.07844
bn_mul_add_2_words	1.00016
bn_mul_add_1_words	0.9636
bn_sqr_5_words	1.09392
(mul との比較)	(84.86%)

4.3.8 改善案

乗算部分は高速化されているので，残りの加算部分について考察する．OpenSSL のソースコードでは，２乗算を計算する処理全体が２つの部分に分かれている．１つは各ワードを単純に２乗する処理，もう１つは各ワードを積和して２倍する処理である．加算の関数 bn_add_10_words は，まず初めに各ワードの積和 r を２倍する処理に利用され，$r + r = 2r$ を計算する．次に各ワードの２乗をした値と $2r$ を足し合わせるために，再度関数 bn_add_10_words が呼び出される．

関数 bn_add_10_words を呼び出すことによって，演算の回数が増えることはない．演算の回数が変わらないのに処理時間がかかるのは，メモリアクセスの回数が増えるからである．以下の表は，メモリアクセス命令の ldr(ロード命令) と str(ストア命令) が使用された回数をまとめたものである．bn_sqr_160 の方が，bn_mul_160 よりもメモリアクセスの回数が多いことが分かる．

表 4.18: bn_mul_160 のメモリアクセス命令の回数 (ARM アセンブラ)

関数	ldr	str
bn_mul_5_words	5	5
bn_mul_add_5_words	10	5
bn_mul_add_5_words	10	5
bn_mul_add_5_words	10	5
bn_mul_add_5_words	10	5
合計	45	25

表 4.19: bn_sqr_160 のメモリアクセス命令の回数 (ARM アセンブラ)

関数	ldr	str
bn_mul_4_words	4	4
bn_mul_add_3_words	6	3
bn_mul_add_2_words	4	2
bn_mul_add_1_words	2	1
bn_add_10_words	20	10
bn_sqr_5_words	5	10
bn_add_10_words	20	10
合計	61	40

メモリアクセスの回数を減らす方法を考える．まず初めに，メモリアクセス命令の ldr と str は複数ワードをまとめてアクセスする命令 ldm と stm に置き換えることができる．次に，値を２倍にする処理は加算ではなくて，シフト演算で代替できる．ARM の場合，シフト演算を高速に処理する仕組みがあるので，高速化が期待できる．さらに，２回目に呼び出される加算処理は，積和を行なえば不要である．よって，関数 bn_add_10_words は２つとも削除することができる．メモリアクセスの回数を削減した結果，２乗算は乗算の約 83 % まで高速化することができた．以下は改善されたメモリアクセスの回数，実行速度の計測結果およびソースコードである．ソースコードの詳細は付録に示す．

表 4.20: bn_mul_160 のメモリアクセス命令の回数 (ARM アセンブラ改)

関数	ldm	ldr	stm	str
bn_mul_5_words	2	1	2	1
bn_mul_add_5_words	4	2	1	1
bn_mul_add_5_words	4	2	1	1
bn_mul_add_5_words	4	2	1	1
bn_mul_add_5_words	4	2	1	1
合計	18	9	6	5

表 4.21: bn_sqr_160 のメモリアクセス命令の回数 (ARM アセンブラ改)

関数	ldm	ldr	stm	str
bn_sqr_5_words	2	1	2	2
bn_mul_shift_add_4_words	2	2	1	1
bn_mul_shift_add_3_words	2	0	1	0
bn_mul_shift_add_2_words	2	0	1	0
bn_mul_shift_add_1_word	0	2	0	1
合計	8	5	5	4

表 4.22: bn_mul_160 の速度 (ARM アセンブラ改)

関数	チック数
bn_mul_160	3.62558
bn_mul_5_words	1.11006
bn_mul_add_4_words	1.21895
bn_mul_add_4_words	1.21895
bn_mul_add_4_words	1.21895
bn_mul_add_4_words	1.21895

表 4.23: bn_sqr_160 の速度 (ARM アセンブラ改)

関数	チック数
bn_sqr_160	3.14129
(mul との比較)	(86.64%)
bn_sqr_5_words	1.09392
bn_mul_shift_add_4_words	1.26576
bn_mul_shift_add_3_words	1.15642
bn_mul_shift_add_2_words	1.04762
bn_mul_shift_add_1_words	0.9638

■ bn_sqr_5_words(): 各ワードの２乗を求める

・input:

BN_ULONG *r: 計算結果を格納する開始アドレス

BN_ULONG *a: 多倍長変数 a のアドレス
int n: 多倍長変数 a のワード数
・output:
戻り値 なし

■ bn_mul_shift_add_4_words(): $rp = rp + 2 \times ap \times w$ を求める (4 ワード)
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_mul_shift_add_3_words(): $rp = rp + 2 \times ap \times w$ を求める (3 ワード)
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_mul_shift_add_2_words(): $rp = rp + 2 \times ap \times w$ を求める (2 ワード)
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

■ bn_mul_shift_add_1_word(): $rp = rp + 2 \times ap \times w$ を求める (1 ワード)
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス

int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

4.4 iPhone における実験

ARM9 における実験では，条件分岐の削減と，演算のマージを行なうことによって，多倍長乗算と 2 乗算の速度を高速化することができた．そこで，同じく ARM プロセッサを実装している iPhone においても，同様の成果がでるかどうか実験を行なった．結果として，同様に多倍長乗算と 2 乗算の速度を高速化することが可能である．

表 4.24: bn_mul_normal の速度

言語	実行速度 (単位:μs)	比率 (%)
C 言語	534.0	100%
ARM アセンブラ	127.5	23.87%

表 4.25: bn_sqr_normal の速度

言語	実行速度 (単位:μs)	比率 (%)
C 言語	694.0	100%
ARM アセンブラ	101.3	14.59%

表 4.26: bn_mul_160 の速度 (ARM アセンブラ)

関数	実行速度 (単位:μs)
bn_mul_160	82.9
bn_mul_5_words	18.1
bn_mul_add_5_words	20.0
bn_mul_add_5_words	20.0
bn_mul_add_5_words	20.0
bn_mul_add_5_words	20.0

表 4.27: bn_sqr_160 の速度 (ARM アセンブラ)

関数	実行速度 (単位:μs)
bn_sqr_160	66.5
(mul との比較)	(80.21%)
bn_sqr_5_words	16.7
bn_mul_shift_add_4_words	22.8
bn_mul_shift_add_3_words	19.7
bn_mul_shift_add_2_words	17.1
bn_mul_shift_add_1_words	15.3

4.4.1 多倍長加算と減算

OpenSSL に実装されている多倍長加算と減算は，それぞれbn_add_words，bn_sub_words である．処理速度を比較すると，減算のほうが速いことが分かる．

表 4.28: 多倍長加算と減算の処理速度 (ARM C 言語)

関数	実行時間 (単位:μs)
bn_add_words	0.850
bn_sub_words	0.835

同様に，多倍長MOD加算とMOD減算は，それぞれBN_mod_add_quick，BN_mod_sub_quick である．処理速度を比較すると，減算のほうが速いことが分かる．

多倍長MOD加算は，Incomplete Reduction(IR) を導入することによって高速化するこ

表 4.29: 多倍長 MOD 加算と MOD 減算の処理速度 (ARM C 言語)

関数	実行時間 (単位:μs)
BN_mod_add_quick	1.53
BN_mod_sub_quick	1.01

表 4.30: Incomplete Reduction(IR) の導入 (ARM C 言語)

関数	実行時間 (単位:μs)
BN_mod_add (IR)	1.45
BN_mod_add_quick	1.53

とができる．しかし依然として，減算のほうが速いことが分かる．

4.4.2 加算公式の高速化

関数EC_POINT_mulの内部で実行される楕円加算と楕円２倍算は，ヤコビアン座標系の加算公式が実装されている．乗算を M，２乗算を S とすると，加算の計算量は $12M+4S$，２倍算の計算量は $4M+6S$ である．これに対して最新の加算公式 [8] では，加算の計算量は $11M+5S$，２倍算の計算量は $2M+8S$ であるから，OpenSSL の楕円加算と２倍算は高速化することが可能である．さらに，楕円曲線のパラメータ a の値を -3 とすることによって，楕円２倍算の計算量を減らすことができる．

表 4.31: 楕円２倍算の計算量の比較

	$a \neq -3$	$a = -3$
OpenSSL	$4M+6S$	$4M+4S$
最新版 [8]	$2M+8S$	$3M+5S$

楕円 2 倍算 ($a = -3$, $3M + 5S$)
入力：楕円曲線上の点 (X, Y, Z) 出力：楕円曲線上の点 (X', Y', Z') $= (X, Y, Z) + (X, Y, Z)$
01: if $(Y = 0)$ then 02: return 無限遠点 03: $S = 4XY^2$ 04: $M = 3X^2 - 3Z^4 = 3(X + Z^2)(X - Z^2)$ 05: $X' = M^2 - 2S$ 06: $Y' = M(S - X') - 8Y^4$ 07: $Z' = (Y + Z)^2 - Y^2 - Z^2$ 08: return (X', Y', Z')

表 4.32: 楕円 2 倍算 ($a = -3$) の実行速度 (ARM C 言語)

	計算量	実行時間 (単位:μs)
OpenSSL	$4M + 4S$	82.0
最新版 [8]	$3M + 5S$	65.6

4.4.3 スカラー倍算の高速化

以上の高速化手法を利用して，スカラー倍算の高速化を行なう．結果として，従来の方法に対して 63.09%の時間でスカラー倍算が実行された．

表 4.33: スカラー倍算の処理速度 (ARM C 言語)

	実行時間 (単位:μs)	比率 (%)
従来	31700	100.00
提案方法	20000	63.09

第5章　結論

本研究では，ARM プロセッサにおける楕円スカラー倍算の実装を行なった．スカラー倍算の内部で呼び出す多倍長演算は，条件分岐を削減し，演算をマージすることによって効率を改善することが可能である．特に ARM プロセッサを実装する小型機器は，メモリのデータ転送速度が遅い場合が多く，演算のマージによってメモリアクセスの回数を削減することが，高速化を実現するために重要である．演算のマージを行なうにあたって，ARM プロセッサの得意とするシフト演算と，複数ワードのデータをまとめて転送する命令を利用し，従来の多倍長演算の実行時間を約 80%削減することができた．結果として，高速化した多倍長演算と最新の加法公式を利用し，楕円スカラー倍算の実行時間を従来の約 63%まで削減するという結果を得た．

謝辞

本研究を遂行するにあたり，宮地充子教授からは多大なるご指導とご助言を賜わりました．幾度と無く励ましのお言葉をいただき，社会人として働きながら研究を進めるという，決して平坦ではない道のりをここまで歩むことができました．研究に対しては厳しく，人としては優しく接してくださったおかげで，私は成長することができたのだと思います．心より感謝申し上げます．また宮地研究室の諸氏には，石川と東京の距離をこえて，交友を結んでくださいました．研究に関して質問があれば即座に返答していただき，参考となる文献があれば送っていただきました．本研究の成果は，そのような協力なしにはありえなかったことを，ここに記します．

参考文献

[1] Henri Cohen, Gerhard Fray, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren 'HANDBOOK OF ELLIPTIC AND HYPERELLIPTIC CURVE CRYPTOGRAPHY', 2006

[2] Marc Joye and Sung-Ming Yen 'The Montgomery Powering Ladder, 2002

[3] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation, January 1987.

[4] Alfred Brauer, 'On Addition Chains', 1939

[5] Andrew D. Booth, 'A SIGNED BINARY MULTIPLICATION TECHNIQUE', 1950

[6] F. Morain & J. Olivios, 'Speeding up the computations on an elliptic curve using addition-subtraction chains', 1990

[7] Bodo Moeller, 'Algorithms for Multi-exponentiation', 2001

[8] Daniel J. Bernstein and Tanja Lange, 'Faster addition and doubling on elliptic curves', 2007

[9] T. Yanik, E. Savas, and C. K. Koc 'Incomplete Reduction in Modular Arithmetic', Computers and Digital Technique, 149(2):46-52, March 2002

[10] Patrick Longa, and Catherine Gebotys 'Efficient Techniques for High-Speed Elliptic Curve Cryptography', 2010

[11] Patrick Longa, and Catherine Gebotys 'Analysis of Efficient Techniques for Fast Elliptic Curve Cryptography on x86-64 based Processors', 2010

[12] David Seal 'ARM Architecture Reference Manual Second Edition', 2000

[13] ARM プレスルーム, http://www.jp.arm.com/pressroom/08/080125.html, 2008

[14] OpenSSL Project, http://www.openssl.org/, 2010.

付 録A　実験で使用したソースコード

OpenSSL ライブラリ [14] より，C 及び x86 アセンブリ言語で記述されたプログラムを引用した．引用したプログラムのキャプションには，引用番号 [14] がふられている．一方で我々が提案する方法は，x86 アセンブリ言語で記述されたプログラムの中で，条件分岐を削減したものと，ARM アセンブリ言語で記述されたプログラムである．

■ bn_mul_normal(): $r = a \times b$ を求める
・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
int na: 多倍長変数 a のワード数
BN_ULONG *b: 多倍長変数 b のアドレス
int nb: 多倍長変数 b のワード数
・output:
戻り値なし．

ソースコード A.1: bn_mul_normal[14]

```
1  void bn_mul_normal(BN_ULONG *r, BN_ULONG *a, int na,
2                     BN_ULONG *b, int nb)
3  {
4      BN_ULONG *rr;
5
6  #ifdef BN_COUNT
7      fprintf(stderr,"bn_mul_normal %d * %d\n",na,nb);
8  #endif
9
10     if (na < nb)
11     {
12         int itmp;
13         BN_ULONG *ltmp;
14
15         itmp=na; na=nb; nb=itmp;
16         ltmp=a;  a=b;   b=ltmp;
17
18     }
19     rr= &(r[na]);
20     if (nb <= 0)
21     {
```

```
22          (void)bn_mul_words(r,a,na,0);
23          return;
24      }
25      else
26          rr[0]=bn_mul_words(r,a,na,b[0]);
27
28      for (;;)
29      {
30          if (--nb <= 0) return;
31          rr[1]=bn_mul_add_words(&(r[1]),a,na,b[1]);
32          if (--nb <= 0) return;
33          rr[2]=bn_mul_add_words(&(r[2]),a,na,b[2]);
34          if (--nb <= 0) return;
35          rr[3]=bn_mul_add_words(&(r[3]),a,na,b[3]);
36          if (--nb <= 0) return;
37          rr[4]=bn_mul_add_words(&(r[4]),a,na,b[4]);
38          rr+=4;
39          r+=4;
40          b+=4;
41      }
42  }
```

■ bn_sqr_normal(): $r = a \times a$ を求める

・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
int n: 多倍長変数 a のワード数
BN_ULONG *tmp: 計算用領域のアドレス

・output:
戻り値なし.

ソースコード A.2: bn_sqr_normal[14]

```
1   void bn_sqr_normal(BN_ULONG *r, const BN_ULONG *a,
2                      int n, BN_ULONG *tmp)
3   {
4       int i,j,max;
5       const BN_ULONG *ap;
6       BN_ULONG *rp;
7
8       max=n*2;
9       ap=a;
10      rp=r;
11      rp[0]=rp[max-1]=0;
12      rp++;
13      j=n;
14
```

```
15      if (--j > 0)
16      {
17          ap++;
18          rp[j]=bn_mul_words(rp,ap,j,ap[-1]);
19          rp+=2;
20      }
21
22      for (i=n-2; i>0; i--)
23      {
24          j--;
25          ap++;
26          rp[j]=bn_mul_add_words(rp,ap,j,ap[-1]);
27          rp+=2;
28      }
29
30      bn_add_words(r,r,r,max);
31
32      /* There will not be a carry */
33
34      bn_sqr_words(tmp,a,n);
35
36      bn_add_words(r,r,tmp,max);
37  }
```

■ bn_mul_words(): $rp = ap \times w$ を求める

・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数

・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.3: bn_mul_words[14]

```
1   .text
2   .globl     _bn_mul_words
3   _bn_mul_words:
4       pushl     %ebp
5       pushl     %ebx
6       pushl     %esi
7       pushl     %edi
8
9
10      xorl      %esi,          %esi
11      movl      20(%esp),      %edi
12      movl      24(%esp),      %ebx
```

```
13      movl    28(%esp),       %ebp
14      movl    32(%esp),       %ecx
15      andl    $4294967288,    %ebp
16      jz      .L004mw_finish
17  .L005mw_loop:
18
19      movl    (%ebx),         %eax
20      mull    %ecx
21      addl    %esi,           %eax
22      adcl    $0,             %edx
23      movl    %eax,           (%edi)
24      movl    %edx,           %esi
25
26      movl    4(%ebx),        %eax
27      mull    %ecx
28      addl    %esi,           %eax
29      adcl    $0,             %edx
30      movl    %eax,           4(%edi)
31      movl    %edx,           %esi
32
33      movl    8(%ebx),        %eax
34      mull    %ecx
35      addl    %esi,           %eax
36      adcl    $0,             %edx
37      movl    %eax,           8(%edi)
38      movl    %edx,           %esi
39
40      movl    12(%ebx),       %eax
41      mull    %ecx
42      addl    %esi,           %eax
43      adcl    $0,             %edx
44      movl    %eax,           12(%edi)
45      movl    %edx,           %esi
46
47      movl    16(%ebx),       %eax
48      mull    %ecx
49      addl    %esi,           %eax
50      adcl    $0,             %edx
51      movl    %eax,           16(%edi)
52      movl    %edx,           %esi
53
54      movl    20(%ebx),       %eax
55      mull    %ecx
56      addl    %esi,           %eax
57      adcl    $0,             %edx
58      movl    %eax,           20(%edi)
```

```
59      movl    %edx,           %esi
60
61      movl    24(%ebx),       %eax
62      mull    %ecx
63      addl    %esi,           %eax
64      adcl    $0,             %edx
65      movl    %eax,           24(%edi)
66      movl    %edx,           %esi
67
68      movl    28(%ebx),       %eax
69      mull    %ecx
70      addl    %esi,           %eax
71      adcl    $0,             %edx
72      movl    %eax,           28(%edi)
73      movl    %edx,           %esi
74
75      addl    $32,            %ebx
76      addl    $32,            %edi
77      subl    $8,             %ebp
78      jz      .L004mw_finish
79      jmp     .L005mw_loop
80  .L004mw_finish:
81      movl    28(%esp),       %ebp
82      andl    $7,             %ebp
83      jnz     .L006mw_finish2
84      jmp     .L007mw_end
85  .L006mw_finish2:
86
87      movl    (%ebx),         %eax
88      mull    %ecx
89      addl    %esi,           %eax
90      adcl    $0,             %edx
91      movl    %eax,           (%edi)
92      movl    %edx,           %esi
93      decl    %ebp
94      jz      .L007mw_end
95
96      movl    4(%ebx),        %eax
97      mull    %ecx
98      addl    %esi,           %eax
99      adcl    $0,             %edx
100     movl    %eax,           4(%edi)
101     movl    %edx,           %esi
102     decl    %ebp
103     jz      .L007mw_end
104
```

```
105         movl    8(%ebx),    %eax
106         mull    %ecx
107         addl    %esi,       %eax
108         adcl    $0,         %edx
109         movl    %eax,       8(%edi)
110         movl    %edx,       %esi
111         decl    %ebp
112         jz      .L007mw_end
113
114         movl    12(%ebx),   %eax
115         mull    %ecx
116         addl    %esi,       %eax
117         adcl    $0,         %edx
118         movl    %eax,       12(%edi)
119         movl    %edx,       %esi
120         decl    %ebp
121         jz      .L007mw_end
122
123         movl    16(%ebx),   %eax
124         mull    %ecx
125         addl    %esi,       %eax
126         adcl    $0,         %edx
127         movl    %eax,       16(%edi)
128         movl    %edx,       %esi
129         decl    %ebp
130         jz      .L007mw_end
131
132         movl    20(%ebx),   %eax
133         mull    %ecx
134         addl    %esi,       %eax
135         adcl    $0,         %edx
136         movl    %eax,       20(%edi)
137         movl    %edx,       %esi
138         decl    %ebp
139         jz      .L007mw_end
140
141         movl    24(%ebx),   %eax
142         mull    %ecx
143         addl    %esi,       %eax
144         adcl    $0,         %edx
145         movl    %eax,       24(%edi)
146         movl    %edx,       %esi
147 .L007mw_end:
148         movl    %esi,       %eax
149         popl    %edi
150         popl    %esi
```

```
151     popl    %ebx
152     popl    %ebp
153     ret
154 .L_bn_mul_words_end:
```

■ bn_mul_add_words(): 積和 $rp = rp + ap \times w$ を求める

・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数

・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.4: bn_mul_add_words[14]

```
1  .text
2  .globl    _bn_mul_add_words
3  _bn_mul_add_words:
4       pushl    %ebp
5       pushl    %ebx
6       pushl    %esi
7       pushl    %edi
8
9
10      xorl     %esi,          %esi
11      movl     20(%esp),      %edi
12      movl     28(%esp),      %ecx
13      movl     24(%esp),      %ebx
14      andl     $4294967288,     %ecx
15      movl     32(%esp),      %ebp
16      pushl    %ecx
17      jz       .L000maw_finish
18 .L001maw_loop:
19      movl     %ecx,          (%esp)
20
21      movl     (%ebx),          %eax
22      mull     %ebp
23      addl     %esi,          %eax
24      movl     (%edi),          %esi
25      adcl     $0,          %edx
26      addl     %esi,          %eax
27      adcl     $0,          %edx
28      movl     %eax,          (%edi)
29      movl     %edx,          %esi
30
```

```
31      movl    4(%ebx),    %eax
32      mull    %ebp
33      addl    %esi,       %eax
34      movl    4(%edi),    %esi
35      adcl    $0,         %edx
36      addl    %esi,       %eax
37      adcl    $0,         %edx
38      movl    %eax,       4(%edi)
39      movl    %edx,       %esi
40
41      movl    8(%ebx),    %eax
42      mull    %ebp
43      addl    %esi,       %eax
44      movl    8(%edi),    %esi
45      adcl    $0,         %edx
46      addl    %esi,       %eax
47      adcl    $0,         %edx
48      movl    %eax,       8(%edi)
49      movl    %edx,       %esi
50
51      movl    12(%ebx),   %eax
52      mull    %ebp
53      addl    %esi,       %eax
54      movl    12(%edi),   %esi
55      adcl    $0,         %edx
56      addl    %esi,       %eax
57      adcl    $0,         %edx
58      movl    %eax,       12(%edi)
59      movl    %edx,       %esi
60
61      movl    16(%ebx),   %eax
62      mull    %ebp
63      addl    %esi,       %eax
64      movl    16(%edi),   %esi
65      adcl    $0,         %edx
66      addl    %esi,       %eax
67      adcl    $0,         %edx
68      movl    %eax,       16(%edi)
69      movl    %edx,       %esi
70
71      movl    20(%ebx),   %eax
72      mull    %ebp
73      addl    %esi,       %eax
74      movl    20(%edi),   %esi
75      adcl    $0,         %edx
76      addl    %esi,       %eax
```

```
77      adcl    $0,         %edx
78      movl    %eax,         20(%edi)
79      movl    %edx,         %esi
80
81      movl    24(%ebx),     %eax
82      mull    %ebp
83      addl    %esi,         %eax
84      movl    24(%edi),     %esi
85      adcl    $0,         %edx
86      addl    %esi,         %eax
87      adcl    $0,         %edx
88      movl    %eax,         24(%edi)
89      movl    %edx,         %esi
90
91      movl    28(%ebx),     %eax
92      mull    %ebp
93      addl    %esi,         %eax
94      movl    28(%edi),     %esi
95      adcl    $0,         %edx
96      addl    %esi,         %eax
97      adcl    $0,         %edx
98      movl    %eax,         28(%edi)
99      movl    %edx,         %esi
100
101     movl    (%esp),         %ecx
102     addl    $32,          %ebx
103     addl    $32,          %edi
104     subl    $8,         %ecx
105     jnz     .L001maw_loop
106 .L000maw_finish:
107     movl    32(%esp),     %ecx
108     andl    $7,         %ecx
109     jnz     .L002maw_finish2
110     jmp     .L003maw_end
111 .L002maw_finish2:
112
113     movl    (%ebx),         %eax
114     mull    %ebp
115     addl    %esi,         %eax
116     movl    (%edi),         %esi
117     adcl    $0,         %edx
118     addl    %esi,         %eax
119     adcl    $0,         %edx
120     decl    %ecx
121     movl    %eax,         (%edi)
122     movl    %edx,         %esi
```

```
123     jz      .L003maw_end
124
125     movl    4(%ebx),    %eax
126     mull    %ebp
127     addl    %esi,       %eax
128     movl    4(%edi),    %esi
129     adcl    $0,         %edx
130     addl    %esi,       %eax
131     adcl    $0,         %edx
132     decl    %ecx
133     movl    %eax,       4(%edi)
134     movl    %edx,       %esi
135     jz      .L003maw_end
136
137     movl    8(%ebx),    %eax
138     mull    %ebp
139     addl    %esi,       %eax
140     movl    8(%edi),    %esi
141     adcl    $0,         %edx
142     addl    %esi,       %eax
143     adcl    $0,         %edx
144     decl    %ecx
145     movl    %eax,       8(%edi)
146     movl    %edx,       %esi
147     jz      .L003maw_end
148
149     movl    12(%ebx),   %eax
150     mull    %ebp
151     addl    %esi,       %eax
152     movl    12(%edi),   %esi
153     adcl    $0,         %edx
154     addl    %esi,       %eax
155     adcl    $0,         %edx
156     decl    %ecx
157     movl    %eax,       12(%edi)
158     movl    %edx,       %esi
159     jz      .L003maw_end
160
161     movl    16(%ebx),   %eax
162     mull    %ebp
163     addl    %esi,       %eax
164     movl    16(%edi),   %esi
165     adcl    $0,         %edx
166     addl    %esi,       %eax
167     adcl    $0,         %edx
168     decl    %ecx
```

```
169     movl    %eax,           16(%edi)
170     movl    %edx,           %esi
171     jz      .L003maw_end
172
173     movl    20(%ebx),       %eax
174     mull    %ebp
175     addl    %esi,           %eax
176     movl    20(%edi),       %esi
177     adcl    $0,             %edx
178     addl    %esi,           %eax
179     adcl    $0,             %edx
180     decl    %ecx
181     movl    %eax,           20(%edi)
182     movl    %edx,           %esi
183     jz      .L003maw_end
184
185     movl    24(%ebx),       %eax
186     mull    %ebp
187     addl    %esi,           %eax
188     movl    24(%edi),       %esi
189     adcl    $0,             %edx
190     addl    %esi,           %eax
191     adcl    $0,             %edx
192     movl    %eax,           24(%edi)
193     movl    %edx,           %esi
194 .L003maw_end:
195     movl    %esi,           %eax
196     popl    %ecx
197     popl    %edi
198     popl    %esi
199     popl    %ebx
200     popl    %ebp
201     ret
202 .L_bn_mul_add_words_end:
```

■ bn_sqr_words(): 各ワードの２乗を求める

・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
int n: 多倍長変数 a のワード数

・output:
戻り値 なし

ソースコード A.5: bn_sqr_words[14]

```
1 .text
```

```
.globl  _bn_sqr_words
_bn_sqr_words:
    pushl   %ebp
    pushl   %ebx
    pushl   %esi
    pushl   %edi


    movl    20(%esp),   %esi
    movl    24(%esp),   %edi
    movl    28(%esp),   %ebx
    andl    $4294967288,    %ebx
    jz      .L008sw_finish
.L009sw_loop:

    movl    (%edi),     %eax
    mull    %eax
    movl    %eax,       (%esi)
    movl    %edx,       4(%esi)

    movl    4(%edi),    %eax
    mull    %eax
    movl    %eax,       8(%esi)
    movl    %edx,       12(%esi)

    movl    8(%edi),    %eax
    mull    %eax
    movl    %eax,       16(%esi)
    movl    %edx,       20(%esi)

    movl    12(%edi),   %eax
    mull    %eax
    movl    %eax,       24(%esi)
    movl    %edx,       28(%esi)

    movl    16(%edi),   %eax
    mull    %eax
    movl    %eax,       32(%esi)
    movl    %edx,       36(%esi)

    movl    20(%edi),   %eax
    mull    %eax
    movl    %eax,       40(%esi)
    movl    %edx,       44(%esi)

    movl    24(%edi),   %eax
```

```
48      mull    %eax
49      movl    %eax,           48(%esi)
50      movl    %edx,           52(%esi)
51
52      movl    28(%edi),       %eax
53      mull    %eax
54      movl    %eax,           56(%esi)
55      movl    %edx,           60(%esi)
56
57      addl    $32,            %edi
58      addl    $64,            %esi
59      subl    $8,             %ebx
60      jnz     .L009sw_loop
61  .L008sw_finish:
62      movl    28(%esp),       %ebx
63      andl    $7,             %ebx
64      jz      .L010sw_end
65
66      movl    (%edi),         %eax
67      mull    %eax
68      movl    %eax,           (%esi)
69      decl    %ebx
70      movl    %edx,           4(%esi)
71      jz      .L010sw_end
72
73      movl    4(%edi),        %eax
74      mull    %eax
75      movl    %eax,           8(%esi)
76      decl    %ebx
77      movl    %edx,           12(%esi)
78      jz      .L010sw_end
79
80      movl    8(%edi),        %eax
81      mull    %eax
82      movl    %eax,           16(%esi)
83      decl    %ebx
84      movl    %edx,           20(%esi)
85      jz      .L010sw_end
86
87      movl    12(%edi),       %eax
88      mull    %eax
89      movl    %eax,           24(%esi)
90      decl    %ebx
91      movl    %edx,           28(%esi)
92      jz      .L010sw_end
93
```

```
94      movl    16(%edi),       %eax
95      mull    %eax
96      movl    %eax,           32(%esi)
97      decl    %ebx
98      movl    %edx,           36(%esi)
99      jz      .L010sw_end
100
101     movl    20(%edi),       %eax
102     mull    %eax
103     movl    %eax,           40(%esi)
104     decl    %ebx
105     movl    %edx,           44(%esi)
106     jz      .L010sw_end
107
108     movl    24(%edi),       %eax
109     mull    %eax
110     movl    %eax,           48(%esi)
111     movl    %edx,           52(%esi)
112 .L010sw_end:
113     popl    %edi
114     popl    %esi
115     popl    %ebx
116     popl    %ebp
117     ret
118 .L_bn_sqr_words_end:
```

■ bn_add_words(): $r = a + b$ を求める

・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
BN_ULONG *b: 多倍長変数 b のアドレス
int n: 多倍長変数 a または b の大きな方のワード数

・output:
戻り値 BN_ULONG: 繰り上がり値

ソースコード A.6: bn_add_words[14]

```
1  .text
2  .globl    _bn_add_words
3  _bn_add_words:
4      pushl    %ebp
5      pushl    %ebx
6      pushl    %esi
7      pushl    %edi
8
9
```

```
10      movl    20(%esp),       %ebx
11      movl    24(%esp),       %esi
12      movl    28(%esp),       %edi
13      movl    32(%esp),       %ebp
14      xorl    %eax,           %eax
15      andl    $4294967288,    %ebp
16      jz      .L011aw_finish
17  .L012aw_loop:
18
19      movl    (%esi),         %ecx
20      movl    (%edi),         %edx
21      addl    %eax,           %ecx
22      movl    $0,             %eax
23      adcl    %eax,           %eax
24      addl    %edx,           %ecx
25      adcl    $0,             %eax
26      movl    %ecx,           (%ebx)
27
28      movl    4(%esi),        %ecx
29      movl    4(%edi),        %edx
30      addl    %eax,           %ecx
31      movl    $0,             %eax
32      adcl    %eax,           %eax
33      addl    %edx,           %ecx
34      adcl    $0,             %eax
35      movl    %ecx,           4(%ebx)
36
37      movl    8(%esi),        %ecx
38      movl    8(%edi),        %edx
39      addl    %eax,           %ecx
40      movl    $0,             %eax
41      adcl    %eax,           %eax
42      addl    %edx,           %ecx
43      adcl    $0,             %eax
44      movl    %ecx,           8(%ebx)
45
46      movl    12(%esi),       %ecx
47      movl    12(%edi),       %edx
48      addl    %eax,           %ecx
49      movl    $0,             %eax
50      adcl    %eax,           %eax
51      addl    %edx,           %ecx
52      adcl    $0,             %eax
53      movl    %ecx,           12(%ebx)
54
55      movl    16(%esi),       %ecx
```

```
56      movl    16(%edi),       %edx
57      addl    %eax,           %ecx
58      movl    $0,             %eax
59      adcl    %eax,           %eax
60      addl    %edx,           %ecx
61      adcl    $0,             %eax
62      movl    %ecx,           16(%ebx)
63
64      movl    20(%esi),       %ecx
65      movl    20(%edi),       %edx
66      addl    %eax,           %ecx
67      movl    $0,             %eax
68      adcl    %eax,           %eax
69      addl    %edx,           %ecx
70      adcl    $0,             %eax
71      movl    %ecx,           20(%ebx)
72
73      movl    24(%esi),       %ecx
74      movl    24(%edi),       %edx
75      addl    %eax,           %ecx
76      movl    $0,             %eax
77      adcl    %eax,           %eax
78      addl    %edx,           %ecx
79      adcl    $0,             %eax
80      movl    %ecx,           24(%ebx)
81
82      movl    28(%esi),       %ecx
83      movl    28(%edi),       %edx
84      addl    %eax,           %ecx
85      movl    $0,             %eax
86      adcl    %eax,           %eax
87      addl    %edx,           %ecx
88      adcl    $0,             %eax
89      movl    %ecx,           28(%ebx)
90
91      addl    $32,            %esi
92      addl    $32,            %edi
93      addl    $32,            %ebx
94      subl    $8,             %ebp
95      jnz     .L012aw_loop
96  .L011aw_finish:
97      movl    32(%esp),       %ebp
98      andl    $7,             %ebp
99      jz      .L013aw_end
100
101     movl    (%esi),         %ecx
```

```
102     movl    (%edi),         %edx
103     addl    %eax,           %ecx
104     movl    $0,             %eax
105     adcl    %eax,           %eax
106     addl    %edx,           %ecx
107     adcl    $0,             %eax
108     decl    %ebp
109     movl    %ecx,           (%ebx)
110     jz      .L013aw_end
111
112     movl    4(%esi),        %ecx
113     movl    4(%edi),        %edx
114     addl    %eax,           %ecx
115     movl    $0,             %eax
116     adcl    %eax,           %eax
117     addl    %edx,           %ecx
118     adcl    $0,             %eax
119     decl    %ebp
120     movl    %ecx,           4(%ebx)
121     jz      .L013aw_end
122
123     movl    8(%esi),        %ecx
124     movl    8(%edi),        %edx
125     addl    %eax,           %ecx
126     movl    $0,             %eax
127     adcl    %eax,           %eax
128     addl    %edx,           %ecx
129     adcl    $0,             %eax
130     decl    %ebp
131     movl    %ecx,           8(%ebx)
132     jz      .L013aw_end
133
134     movl    12(%esi),       %ecx
135     movl    12(%edi),       %edx
136     addl    %eax,           %ecx
137     movl    $0,             %eax
138     adcl    %eax,           %eax
139     addl    %edx,           %ecx
140     adcl    $0,             %eax
141     decl    %ebp
142     movl    %ecx,           12(%ebx)
143     jz      .L013aw_end
144
145     movl    16(%esi),       %ecx
146     movl    16(%edi),       %edx
147     addl    %eax,           %ecx
```

```
148     movl    $0,         %eax
149     adcl    %eax,         %eax
150     addl    %edx,         %ecx
151     adcl    $0,         %eax
152     decl    %ebp
153     movl    %ecx,         16(%ebx)
154     jz      .L013aw_end
155
156     movl    20(%esi),     %ecx
157     movl    20(%edi),     %edx
158     addl    %eax,         %ecx
159     movl    $0,         %eax
160     adcl    %eax,         %eax
161     addl    %edx,         %ecx
162     adcl    $0,         %eax
163     decl    %ebp
164     movl    %ecx,         20(%ebx)
165     jz      .L013aw_end
166
167     movl    24(%esi),     %ecx
168     movl    24(%edi),     %edx
169     addl    %eax,         %ecx
170     movl    $0,         %eax
171     adcl    %eax,         %eax
172     addl    %edx,         %ecx
173     adcl    $0,         %eax
174     movl    %ecx,         24(%ebx)
175 .L013aw_end:
176     popl    %edi
177     popl    %esi
178     popl    %ebx
179     popl    %ebp
180     ret
181 .L_bn_add_words_end:
```

■ bn_add_10_words(): 10 words の値 $r = a + b$ を求める

・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
BN_ULONG *b: 多倍長変数 b のアドレス

・output:
戻り値 なし

ソースコード A.7: bn_add_10_words

```
1 .text
2 .globl    _bn_add_10_words
```

```
3  _bn_add_10_words:
4      pushl    %ebp
5      pushl    %ebx
6      pushl    %esi
7      pushl    %edi
8
9      movl     20(%esp),    %ebx
10     movl     24(%esp),    %esi
11     movl     28(%esp),    %edi
12     clc
13
14     movl     (%esi),        %ecx
15     movl     (%edi),        %edx
16     addl     %edx,          %ecx
17     movl     %ecx,          (%ebx)
18
19     movl     4(%esi),     %ecx
20     movl     4(%edi),     %edx
21     adcl     %edx,          %ecx
22     movl     %ecx,          4(%ebx)
23
24     movl     8(%esi),     %ecx
25     movl     8(%edi),     %edx
26     adcl     %edx,          %ecx
27     movl     %ecx,          8(%ebx)
28
29     movl     12(%esi),    %ecx
30     movl     12(%edi),    %edx
31     adcl     %edx,          %ecx
32     movl     %ecx,          12(%ebx)
33
34     movl     16(%esi),    %ecx
35     movl     16(%edi),    %edx
36     adcl     %edx,          %ecx
37     movl     %ecx,          16(%ebx)
38
39     movl     20(%esi),    %ecx
40     movl     20(%edi),    %edx
41     adcl     %edx,          %ecx
42     movl     %ecx,          20(%ebx)
43
44     movl     24(%esi),    %ecx
45     movl     24(%edi),    %edx
46     adcl     %edx,          %ecx
47     movl     %ecx,          24(%ebx)
48
```

```
49      movl    28(%esi),       %ecx
50      movl    28(%edi),       %edx
51      adcl    %edx,           %ecx
52      movl    %ecx,           28(%ebx)
53
54      movl    32(%esi),       %ecx
55      movl    32(%edi),       %edx
56      adcl    %edx,           %ecx
57      movl    %ecx,           32(%ebx)
58
59      movl    36(%esi),       %ecx
60      movl    36(%edi),       %edx
61      adcl    %edx,           %ecx
62      movl    %ecx,           36(%ebx)
63
64      popl    %edi
65      popl    %esi
66      popl    %ebx
67      popl    %ebp
68      ret
69  .L_bn_add_10_words_end:
```

■ bn_mul_4_words(): $rp = ap \times w$ を求める

・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
BN_ULONG w: ワード長変数

・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.8: bn_mul_4_words

```
1   .text
2   .globl      _bn_mul_4_words
3   _bn_mul_4_words:
4       pushl     %ebp
5       pushl     %ebx
6       pushl     %esi
7       pushl     %edi
8
9       xorl    %esi,           %esi
10      movl    20(%esp),       %edi
11      movl    24(%esp),       %ebx
12      movl    28(%esp),       %ecx
13
14      movl    (%ebx),           %eax
```

```
15      mull    %ecx
16      movl    %eax,           (%edi)
17      movl    %edx,           %esi
18
19      movl    4(%ebx),        %eax
20      mull    %ecx
21      addl    %esi,           %eax
22      adcl    $0,             %edx
23      movl    %eax,           4(%edi)
24      movl    %edx,           %esi
25
26      movl    8(%ebx),        %eax
27      mull    %ecx
28      addl    %esi,           %eax
29      adcl    $0,             %edx
30      movl    %eax,           8(%edi)
31      movl    %edx,           %esi
32
33      movl    12(%ebx),       %eax
34      mull    %ecx
35      addl    %esi,           %eax
36      adcl    $0,             %edx
37      movl    %eax,           12(%edi)
38      movl    %edx,           %esi
39
40      movl    %esi,           %eax
41
42      popl    %edi
43      popl    %esi
44      popl    %ebx
45      popl    %ebp
46      ret
47  .L_bn_mul_4_words_end:
```

■ bn_mul_add_3_words(): 積和 $rp = rp + ap \times w$ を求める

・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
BN_ULONG w: ワード長変数

・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.9: bn_mul_add_3_words

```
1   .text
2   .globl      _bn_mul_add_3_words
3   _bn_mul_add_3_words:
```

```
4       pushl   %ebp
5       pushl   %ebx
6       pushl   %esi
7       pushl   %edi
8
9       xorl    %esi, %esi
10      movl    20(%esp), %edi
11      movl    24(%esp), %ebx
12      movl    28(%esp), %ebp
13
14      movl    (%ebx), %eax
15      mull    %ebp
16      addl    %esi, %eax
17      movl    (%edi), %esi
18      adcl    $0, %edx
19      addl    %esi, %eax
20      adcl    $0, %edx
21      movl    %eax, (%edi)
22      movl    %edx, %esi
23
24      movl    4(%ebx), %eax
25      mull    %ebp
26      addl    %esi, %eax
27      movl    4(%edi), %esi
28      adcl    $0, %edx
29      addl    %esi, %eax
30      adcl    $0, %edx
31      movl    %eax, 4(%edi)
32      movl    %edx, %esi
33
34      movl    8(%ebx), %eax
35      mull    %ebp
36      addl    %esi, %eax
37      movl    8(%edi), %esi
38      adcl    $0, %edx
39      addl    %esi, %eax
40      adcl    $0, %edx
41      movl    %eax, 8(%edi)
42      movl    %edx, %esi
43
44      movl    %esi, %eax
45
46      popl    %edi
47      popl    %esi
48      popl    %ebx
49      popl    %ebp
```

```
50      ret
51  .L_bn_mul_add_3_words_end:
```

■ bn_mul_add_2_words(): 積和 $rp = rp + ap \times w$ を求める

・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
BN_ULONG w: ワード長変数

・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.10: bn_mul_add_2_words

```
1   .text
2   .globl     _bn_mul_add_2_words
3   _bn_mul_add_2_words:
4       pushl     %ebp
5       pushl     %ebx
6       pushl     %esi
7       pushl     %edi
8
9       xorl      %esi,          %esi
10      movl      20(%esp),      %edi
11      movl      24(%esp),      %ebx
12      movl      28(%esp),      %ebp
13
14      movl      (%ebx),          %eax
15      mull      %ebp
16      addl      %esi,          %eax
17      movl      (%edi),          %esi
18      adcl      $0,          %edx
19      addl      %esi,          %eax
20      adcl      $0,          %edx
21      movl      %eax,          (%edi)
22      movl      %edx,          %esi
23
24      movl      4(%ebx),      %eax
25      mull      %ebp
26      addl      %esi,          %eax
27      movl      4(%edi),      %esi
28      adcl      $0,          %edx
29      addl      %esi,          %eax
30      adcl      $0,          %edx
31      movl      %eax,          4(%edi)
32      movl      %edx,          %esi
33
```

```
34      movl    %esi,           %eax
35
36      popl    %edi
37      popl    %esi
38      popl    %ebx
39      popl    %ebp
40      ret
41  .L_bn_mul_add_2_words_end:
```

■ bn_mul_add_1_word(): 積和 $rp = rp + ap \times w$ を求める

・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
BN_ULONG w: ワード長変数

・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.11: bn_mul_add_1_word

```
1   .text
2   .globl      _bn_mul_add_1_word
3   _bn_mul_add_1_word:
4       pushl    %ebp
5       pushl    %ebx
6       pushl    %esi
7       pushl    %edi
8
9       xorl    %esi,           %esi
10      movl    20(%esp),       %edi
11      movl    24(%esp),       %ebx
12      movl    28(%esp),       %ebp
13
14      movl    (%ebx),           %eax
15      mull    %ebp
16      movl    (%edi),           %esi
17      addl    %esi,           %eax
18      adcl    $0,           %edx
19      movl    %eax,           (%edi)
20      movl    %edx,           %esi
21
22      movl    %esi,           %eax
23
24      popl    %edi
25      popl    %esi
26      popl    %ebx
27      popl    %ebp
```

```
28      ret
29  .L_bn_mul_add_1_word_end:
```

■ bn_sqr_5_words(): 各ワードの２乗を求める

・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス

・output:
戻り値 なし

ソースコード A.12: bn_sqr_5_words

```
1   .text
2   .globl      _bn_sqr_5_words
3   _bn_sqr_5_words:
4       pushl     %ebp
5       pushl     %ebx
6       pushl     %esi
7       pushl     %edi
8
9       movl      20(%esp),     %esi
10      movl      24(%esp),     %edi
11
12      movl      (%edi),         %eax
13      mull      %eax
14      movl      %eax,         (%esi)
15      movl      %edx,         4(%esi)
16
17      movl      4(%edi),      %eax
18      mull      %eax
19      movl      %eax,         8(%esi)
20      movl      %edx,         12(%esi)
21
22      movl      8(%edi),      %eax
23      mull      %eax
24      movl      %eax,         16(%esi)
25      movl      %edx,         20(%esi)
26
27      movl      12(%edi),     %eax
28      mull      %eax
29      movl      %eax,         24(%esi)
30      movl      %edx,         28(%esi)
31
32      movl      16(%edi),     %eax
33      mull      %eax
34      movl      %eax,         32(%esi)
```

```
35      movl    %edx,           36(%esi)
36
37      popl    %edi
38      popl    %esi
39      popl    %ebx
40      popl    %ebp
41      ret
42  .L_bn_sqr_5_words_end:
```

■ bn_mul_words(): $rp = ap \times w$ を求める

・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数

・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.13: bn_mul_words.arm

```
1   asm BN_ULONG bn_mul_words(
2       register BN_ULONG *rp,
3       register const BN_ULONG *ap,
4       register int num,
5       register BN_ULONG w)
6   {
7       mov     r7, #0
8       adds    r8, r7, #0
9
10      ldr     r9, =0xffffffff8
11      ands    r10, r2, r9
12      beq     L004mw_finish
13
14  L005mw_loop:
15
16      ldr     r4, [r1]
17      umull   r5, r6, r4, r3
18      adds    r5, r5, r7
19      adcs    r6, r6, r8
20      str     r5, [r0]
21      mov     r7, r6
22
23      ldr     r4, [r1, #4]
24      umull   r5, r6, r4, r3
25      adds    r5, r5, r7
26      adcs    r6, r6, r8
27      str     r5, [r0, #4]
```

```
28      mov     r7, r6
29
30      ldr     r4, [r1, #8]
31      umull   r5, r6, r4, r3
32      adds    r5, r5, r7
33      adcs    r6, r6, r8
34      str     r5, [r0, #8]
35      mov     r7, r6
36
37      ldr     r4, [r1, #12]
38      umull   r5, r6, r4, r3
39      adds    r5, r5, r7
40      adcs    r6, r6, r8
41      str     r5, [r0, #12]
42      mov     r7, r6
43
44      ldr     r4, [r1, #20]
45      umull   r5, r6, r4, r3
46      adds    r5, r5, r7
47      adcs    r6, r6, r8
48      str     r5, [r0, #20]
49      mov     r7, r6
50
51      ldr     r4, [r1, #24]
52      umull   r5, r6, r4, r3
53      adds    r5, r5, r7
54      adcs    r6, r6, r8
55      str     r5, [r0, #24]
56      mov     r7, r6
57
58      ldr     r4, [r1, #28]
59      umull   r5, r6, r4, r3
60      adds    r5, r5, r7
61      adcs    r6, r6, r8
62      str     r5, [r0, #28]
63      mov     r7, r6
64
65      add     r0, r0, #32
66      add     r1, r1, #32
67      sub     r2, r2, #8
68
69      beq     L004mw_finish
70      b       L005mw_loop
71
72  L004mw_finish:
73
```

```
74      mov     r10, #7
75      ands    r2, r2, r10
76      bne     L006mw_finish2
77      b       L007mw_end
78
79  L006mw_finish2:
80
81      ldr     r4, [r1]
82      umull   r5, r6, r4, r3
83      adds    r5, r5, r7
84      adcs    r6, r6, r8
85      str     r5, [r0]
86      mov     r7, r6
87      sub     r2, r2, #1
88      beq     L007mw_end
89
90      ldr     r4, [r1, #4]
91      umull   r5, r6, r4, r3
92      adds    r5, r5, r7
93      adcs    r6, r6, r8
94      str     r5, [r0, #4]
95      mov     r7, r6
96      sub     r2, r2, #1
97      beq     L007mw_end
98
99      ldr     r4, [r1, #8]
100     umull   r5, r6, r4, r3
101     adds    r5, r5, r7
102     adcs    r6, r6, r8
103     str     r5, [r0, #8]
104     mov     r7, r6
105     sub     r2, r2, #1
106     beq     L007mw_end
107
108     ldr     r4, [r1, #12]
109     umull   r5, r6, r4, r3
110     adds    r5, r5, r7
111     adcs    r6, r6, r8
112     str     r5, [r0, #12]
113     mov     r7, r6
114     sub     r2, r2, #1
115     beq     L007mw_end
116
117     ldr     r4, [r1, #16]
118     umull   r5, r6, r4, r3
119     adds    r5, r5, r7
```

```
120     adcs    r6, r6, r8
121     str     r5, [r0, #16]
122     mov     r7, r6
123     sub     r2, r2, #1
124     beq     L007mw_end
125
126     ldr     r4, [r1, #20]
127     umull   r5, r6, r4, r3
128     adds    r5, r5, r7
129     adcs    r6, r6, r8
130     str     r5, [r0, #20]
131     mov     r7, r6
132     sub     r2, r2, #1
133     beq     L007mw_end
134
135     ldr     r4, [r1, #24]
136     umull   r5, r6, r4, r3
137     adds    r5, r5, r7
138     adcs    r6, r6, r8
139     str     r5, [r0, #24]
140     mov     r7, r6
141     sub     r2, r2, #1
142     beq     L007mw_end
143
144 L007mw_end:
145
146     mov     r0, r7
147     mov     pc, lr
148 }
```

■ bn_mul_add_words(): 積和 $rp = rp + ap \times w$ を求める

・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数

・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.14: bn_mul_add_words.arm

```
1 asm BN_ULONG bn_mul_add_words(
2     register BN_ULONG *rp,
3     register const BN_ULONG *ap,
4     register int num,
5     register BN_ULONG w)
```

```
{
    mov     r7, #0
    adds    r8, r7, #0

    ldr     r9, =0xfffffff8
    ands    r10, r2, r9
    beq     L000maw_finish

L001maw_loop:

    ldr     r4, [r1]
    umull   r5, r6, r4, r3
    adds    r5, r5, r7
    ldr     r7, [r0]
    adcs    r6, r6, r8
    adcs    r7, r6, r8
    str     r5, [r0]
    mov     r7, r6

    ldr     r4, [r1, #4]
    umull   r5, r6, r4, r3
    adds    r5, r5, r7
    ldr     r7, [r0, #4]
    adcs    r6, r6, r8
    adcs    r7, r6, r8
    str     r5, [r0, #4]
    mov     r7, r6

    ldr     r4, [r1, #8]
    umull   r5, r6, r4, r3
    adds    r5, r5, r7
    ldr     r7, [r0, #8]
    adcs    r6, r6, r8
    adcs    r7, r6, r8
    str     r5, [r0, #8]
    mov     r7, r6

    ldr     r4, [r1, #12]
    umull   r5, r6, r4, r3
    adds    r5, r5, r7
    ldr     r7, [r0, #12]
    adcs    r6, r6, r8
    adcs    r7, r6, r8
    str     r5, [r0, #12]
    mov     r7, r6

```

```
52      ldr     r4, [r1, #16]
53      umull   r5, r6, r4, r3
54      adds    r5, r5, r7
55      ldr     r7, [r0, #16]
56      adcs    r6, r6, r8
57      adcs    r7, r6, r8
58      str     r5, [r0, #16]
59      mov     r7, r6
60
61      ldr     r4, [r1, #20]
62      umull   r5, r6, r4, r3
63      adds    r5, r5, r7
64      ldr     r7, [r0, #20]
65      adcs    r6, r6, r8
66      adcs    r7, r6, r8
67      str     r5, [r0, #20]
68      mov     r7, r6
69
70      ldr     r4, [r1, #24]
71      umull   r5, r6, r4, r3
72      adds    r5, r5, r7
73      ldr     r7, [r0, #24]
74      adcs    r6, r6, r8
75      adcs    r7, r6, r8
76      str     r5, [r0, #24]
77      mov     r7, r6
78
79      ldr     r4, [r1, #28]
80      umull   r5, r6, r4, r3
81      adds    r5, r5, r7
82      ldr     r7, [r0, #28]
83      adcs    r6, r6, r8
84      adcs    r7, r6, r8
85      str     r5, [r0, #28]
86      mov     r7, r6
87
88      add     r1, r1, #32
89      add     r0, r0, #32
90      sub     r2, r2, #8
91      bne     L001maw_loop
92
93  L000maw_finish:
94
95      ands    r2, r2, #7
96      bne     L002maw_finish2
97      b       L003maw_end
```

```
98
99  L002maw_finish2:
100
101     ldr     r4, [r1]
102     umull   r5, r6, r4, r3
103     adds    r5, r5, r7
104     ldr     r7, [r0]
105     adcs    r6, r6, r8
106     adcs    r7, r6, r8
107     subs    r7, r7, #1
108     str     r5, [r0]
109     mov     r7, r6
110     beq     L003maw_end
111
112     ldr     r4, [r1, #4]
113     umull   r5, r6, r4, r3
114     adds    r5, r5, r7
115     ldr     r7, [r0, #4]
116     adcs    r6, r6, r8
117     adcs    r7, r6, r8
118     subs    r7, r7, #1
119     str     r5, [r0, #4]
120     mov     r7, r6
121     beq     L003maw_end
122
123     ldr     r4, [r1, #8]
124     umull   r5, r6, r4, r3
125     adds    r5, r5, r7
126     ldr     r7, [r0, #8]
127     adcs    r6, r6, r8
128     adcs    r7, r6, r8
129     subs    r7, r7, #1
130     str     r5, [r0, #8]
131     mov     r7, r6
132     beq     L003maw_end
133
134     ldr     r4, [r1, #12]
135     umull   r5, r6, r4, r3
136     adds    r5, r5, r7
137     ldr     r7, [r0, #12]
138     adcs    r6, r6, r8
139     adcs    r7, r6, r8
140     subs    r7, r7, #1
141     str     r5, [r0, #12]
142     mov     r7, r6
143     beq     L003maw_end
```

```
144
145     ldr     r4, [r1, #16]
146     umull   r5, r6, r4, r3
147     adds    r5, r5, r7
148     ldr     r7, [r0, #16]
149     adcs    r6, r6, r8
150     adcs    r7, r6, r8
151     subs    r7, r7, #1
152     str     r5, [r0, #16]
153     mov     r7, r6
154     beq     L003maw_end
155
156     ldr     r4, [r1, #20]
157     umull   r5, r6, r4, r3
158     adds    r5, r5, r7
159     ldr     r7, [r0, #20]
160     adcs    r6, r6, r8
161     adcs    r7, r6, r8
162     subs    r7, r7, #1
163     str     r5, [r0, #20]
164     mov     r7, r6
165     beq     L003maw_end
166
167     ldr     r4, [r1, #24]
168     umull   r5, r6, r4, r3
169     adds    r5, r5, r7
170     ldr     r7, [r0, #24]
171     adcs    r6, r6, r8
172     adcs    r7, r6, r8
173     subs    r7, r7, #1
174     str     r5, [r0, #24]
175     mov     r7, r6
176     beq     L003maw_end
177
178 L003maw_end:
179
180     mov     r0, r7
181     mov     pc, lr
182
183 }
```

■ bn_sqr_words(): 各ワードの２乗を求める

・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
int n: 多倍長変数 a のワード数

・output:
戻り値 なし

ソースコード A.15: bn_sqr_words.arm

```
asm void bn_sqr_words(
    register BN_ULONG *r,
    register const BN_ULONG *a,
    register int n)
{
    mov     r7, #0
    adds    r8, r7, #0

    ldr     r9, =0xfffffff8
    ands    r10, r2, r9
    beq     L008sw_finish

L009sw_loop:

    ldr     r4, [r1]
    umull   r5, r6, r4, r4
    str     r5, [r0]
    str     r6, [r0, #4]

    ldr     r4, [r1, #4]
    umull   r5, r6, r4, r4
    str     r5, [r0, #8]
    str     r6, [r0, #12]

    ldr     r4, [r1, #8]
    umull   r5, r6, r4, r4
    str     r5, [r0, #16]
    str     r6, [r0, #20]

    ldr     r4, [r1, #12]
    umull   r5, r6, r4, r4
    str     r5, [r0, #24]
    str     r6, [r0, #28]

    ldr     r4, [r1, #16]
    umull   r5, r6, r4, r4
    str     r5, [r0, #32]
    str     r6, [r0, #36]

    ldr     r4, [r1, #20]
    umull   r5, r6, r4, r4
```

```
42      str     r5, [r0, #40]
43      str     r6, [r0, #44]
44
45      ldr     r4, [r1, #24]
46      umull   r5, r6, r4, r4
47      str     r5, [r0, #48]
48      str     r6, [r0, #52]
49
50      ldr     r4, [r1, #28]
51      umull   r5, r6, r4, r4
52      str     r5, [r0, #56]
53      str     r6, [r0, #60]
54
55      add     r1, r1, #32
56      add     r0, r0, #64
57      subs    r2, r2, #8
58      bne     L009sw_loop
59
60  L008sw_finish:
61
62      ands    r2, r2, #7
63      beq     L010sw_end
64
65      ldr     r4, [r1]
66      umull   r5, r6, r4, r4
67      str     r5, [r0]
68      subs    r2, r2, #1
69      str     r6, [r0, #4]
70      beq     L010sw_end
71
72      ldr     r4, [r1, #4]
73      umull   r5, r6, r4, r4
74      str     r5, [r0, #8]
75      subs    r2, r2, #1
76      str     r6, [r0, #12]
77      beq     L010sw_end
78
79      ldr     r4, [r1, #8]
80      umull   r5, r6, r4, r4
81      str     r5, [r0, #16]
82      subs    r2, r2, #1
83      str     r6, [r0, #20]
84      beq     L010sw_end
85
86      ldr     r4, [r1, #12]
87      umull   r5, r6, r4, r4
```

```
88      str     r5, [r0, #24]
89      subs    r2, r2, #1
90      str     r6, [r0, #28]
91      beq     L010sw_end
92
93      ldr     r4, [r1, #16]
94      umull   r5, r6, r4, r4
95      str     r5, [r0, #32]
96      subs    r2, r2, #1
97      str     r6, [r0, #36]
98      beq     L010sw_end
99
100     ldr     r4, [r1, #20]
101     umull   r5, r6, r4, r4
102     str     r5, [r0, #40]
103     subs    r2, r2, #1
104     str     r6, [r0, #44]
105     beq     L010sw_end
106
107     ldr     r4, [r1, #24]
108     umull   r5, r6, r4, r4
109     str     r5, [r0, #48]
110     subs    r2, r2, #1
111     str     r6, [r0, #52]
112
113 L010sw_end:
114
115     mov     pc, lr
116
117 }
```

■ bn_add_words(): $r = a + b$ を求める

・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
BN_ULONG *b: 多倍長変数 b のアドレス
int n: 多倍長変数 a または b の大きな方のワード数

・output:
戻り値 BN_ULONG: 繰り上がり値

ソースコード A.16: bn_add_words.arm

```
1 asm BN_ULONG bn_add_words(
2     register BN_ULONG *r,
3     register const BN_ULONG *a,
4     register const BN_ULONG *b,
```

```
5       register int n)
6   {
7       mov     r7, #0
8       adds    r8, r7, #0
9
10      ldr     r9, =0xfffffff8
11      ands    r10, r3, r9
12      beq     L011aw_finish
13
14  L012aw_loop:
15
16      ldr     r4, [r1]
17      ldr     r5, [r2]
18      adds    r4, r4, r7
19      adcs    r6, r5, r4
20      str     r6, [r0]
21
22      ldr     r4, [r1, #4]
23      ldr     r5, [r2, #4]
24      adcs    r6, r5, r4
25      str     r6, [r0, #4]
26
27      ldr     r4, [r1, #8]
28      ldr     r5, [r2, #8]
29      adcs    r6, r5, r4
30      str     r6, [r0, #8]
31
32      ldr     r4, [r1, #12]
33      ldr     r5, [r2, #12]
34      adcs    r6, r5, r4
35      str     r6, [r0, #12]
36
37      ldr     r4, [r1, #16]
38      ldr     r5, [r2, #16]
39      adcs    r6, r5, r4
40      str     r6, [r0, #16]
41
42      ldr     r4, [r1, #20]
43      ldr     r5, [r2, #20]
44      adcs    r6, r5, r4
45      str     r6, [r0, #20]
46
47      ldr     r4, [r1, #24]
48      ldr     r5, [r2, #24]
49      adcs    r6, r5, r4
50      str     r6, [r0, #24]
```

```
51
52      ldr     r4, [r1, #28]
53      ldr     r5, [r2, #28]
54      adcs    r6, r5, r4
55      str     r6, [r0, #28]
56
57      adc     r7, r8, r8
58
59      add     r1, r1, #32
60      add     r2, r2, #32
61      add     r0, r0, #32
62      subs    r3, r3, #8
63      bne     L012aw_loop
64
65  L011aw_finish:
66
67      ands    r3, r3, #7
68      beq     L013aw_end
69
70      ldr     r4, [r1]
71      ldr     r5, [r2]
72      adds    r4, r4, r7
73      adcs    r6, r5, r4
74      adc     r7, r8, r8
75      subs    r3, r3, #1
76      str     r6, [r0]
77      beq     L013aw_end
78
79      ldr     r4, [r1, #4]
80      ldr     r5, [r2, #4]
81      adds    r4, r4, r7
82      adcs    r6, r5, r4
83      adc     r7, r8, r8
84      subs    r3, r3, #1
85      str     r6, [r0, #4]
86      beq     L013aw_end
87
88      ldr     r4, [r1, #8]
89      ldr     r5, [r2, #8]
90      adds    r4, r4, r7
91      adcs    r6, r5, r4
92      adc     r7, r8, r8
93      subs    r3, r3, #1
94      str     r6, [r0, #8]
95      beq     L013aw_end
96
```

```
97      ldr     r4, [r1, #12]
98      ldr     r5, [r2, #12]
99      adds    r4, r4, r7
100     adcs    r6, r5, r4
101     adc     r7, r8, r8
102     subs    r3, r3, #1
103     str     r6, [r0, #12]
104     beq     L013aw_end
105
106     ldr     r4, [r1, #16]
107     ldr     r5, [r2, #16]
108     adds    r4, r4, r7
109     adcs    r6, r5, r4
110     adc     r7, r8, r8
111     subs    r3, r3, #1
112     str     r6, [r0, #16]
113     beq     L013aw_end
114
115     ldr     r4, [r1, #20]
116     ldr     r5, [r2, #20]
117     adds    r4, r4, r7
118     adcs    r6, r5, r4
119     adc     r7, r8, r8
120     subs    r3, r3, #1
121     str     r6, [r0, #20]
122     beq     L013aw_end
123
124     ldr     r4, [r1, #24]
125     ldr     r5, [r2, #24]
126     adds    r4, r4, r7
127     adcs    r6, r5, r4
128     adc     r7, r8, r8
129     subs    r3, r3, #1
130     str     r6, [r0, #24]
131
132 L013aw_end:
133
134     mov     r0, r7
135     mov     pc, lr
136 }
```

■ bn_sqr_5_words(): 各ワードの２乗を求める

・input:
BN_ULONG *r: 計算結果を格納する開始アドレス
BN_ULONG *a: 多倍長変数 a のアドレス
int n: 多倍長変数 a のワード数

・output:
戻り値 なし

ソースコード A.17: bn_sqr_5_words_fast.arm

```
asm
void bn_sqr_5_words(
    register BN_ULONG *rp,
    register const BN_ULONG *ap)
{
    stmfd    sp!,{r2-r7, lr}

    ldmia    r1!,{r2,r3}
    umull    r4, r5, r2, r2
    umull    r6, r7, r3, r3
    stmia    r0!,{r4-r7}

    ldmia    r1!,{r2,r3}
    umull    r4, r5, r2, r2
    umull    r6, r7, r3, r3
    stmia    r0!,{r4-r7}

    ldr      r4, [r1]
    umull    r5, r6, r4, r4
    str      r5, [r0]
    str      r6, [r0, #4]

    ldmfd    sp!,{r2-r7, pc}
}
```

■ bn_mul_shift_add_4_words(): $rp = rp + 2 \times ap \times w$ を求める (4 ワード)
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.18: bn_mul_shift_add_4_words.arm

```
asm
BN_ULONG bn_mul_shift_add_4_words(
    register BN_ULONG *rp,
    register const BN_ULONG *ap,
```

```
5       register BN_ULONG w)
6   {
7       stmfd     sp!,{r3-r12, lr}
8
9       eor       r6, r6, r6
10
11      ldmia     r1!,{r3-r5}
12      umull     r7, r8, r3, r2
13      umull     r9, r10, r4, r2
14      umull     r11, r12, r5, r2
15
16      movs      r7, r7, LSL #1
17      adc       r8, r6, r8, LSL #1
18      movs      r9, r9, LSL #1
19      adc       r10, r6, r10, LSL #1
20      adds      r9, r9, r8
21      adc       r10, r6, r10
22      movs      r11, r11, LSL #1
23      adc       r12, r6, r12, LSL #1
24      adds      r11, r11, r10
25      adc       r12, r6, r12
26
27      ldmia     r0, {r3,r4,r5}
28      adds      r3, r3, r7
29      adcs      r4, r4, r9
30      adcs      r5, r5, r11
31      stmia     r0!,{r3,r4,r5}
32
33      ldr       r3, [r1]
34      umull     r4, r5, r3, r2
35      movs      r4, r4, LSL #1
36      add       r4, r4, r12
37      adc       r5, r6, r5, LSL #1
38
39      ldr       r6, [r0]
40      adds      r6, r6, r4
41      str       r6, [r0]
42
43      adc       r0, r5, #0
44
45      ldmfd     sp!,{r3-r12, pc}
46  }
```

■ bn_mul_shift_add_3_words(): $rp = rp + 2 \times ap \times w$ を求める (3 ワード)

・input:

BN_ULONG *rp: 計算結果を格納する開始アドレス

BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数

・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.19: bn_mul_shift_add_3_words.arm

```
asm
BN_ULONG bn_mul_shift_add_3_words(
    register BN_ULONG *rp,
    register const BN_ULONG *ap,
    register BN_ULONG w)
{
    stmfd    sp!,{r3-r12, lr}

    eor      r6, r6, r6

    ldmia    r1!,{r3-r5}
    umull    r7, r8, r3, r2
    umull    r9, r10, r4, r2
    umull    r11, r12, r5, r2

    movs     r7, r7, LSL #1
    adc      r8, r6, r8, LSL #1
    movs     r9, r9, LSL #1
    adc      r10, r6, r10, LSL #1
    adds     r9, r9, r8
    adc      r10, r6, r10
    movs     r11, r11, LSL #1
    adc      r12, r6, r12, LSL #1
    adds     r11, r11, r10
    adc      r12, r6, r12

    ldmia    r0, {r3,r4,r5}
    adds     r3, r3, r7
    adcs     r4, r4, r9
    adcs     r5, r5, r11
    stmia    r0, {r3,r4,r5}

    adc      r0, r12, #0

    ldmfd    sp!,{r3-r12, pc}
}
```

■ bn_mul_shift_add_2_words(): $rp = rp + 2 \times ap \times w$ を求める (2 ワード)

・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数
BN_ULONG w: ワード長変数
・output:
戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.20: bn_mul_shift_add_2_words.arm

```
asm
BN_ULONG bn_mul_shift_add_2_words(
    register BN_ULONG *rp,
    register const BN_ULONG *ap,
    register BN_ULONG w)
{
    stmfd    sp!,{r3-r10, lr}

    eor      r6, r6, r6

    ldmia    r1, {r3,r4}
    umull    r7, r8, r3, r2
    umull    r9, r10, r4, r2

    movs     r7, r7, LSL #1
    adc      r8, r6, r8, LSL #1
    movs     r9, r9, LSL #1
    adc      r10, r6, r10, LSL #1
    adds     r9, r9, r8
    adc      r10, r6, r10

    ldmia    r0, {r3,r4}
    adds     r3, r3, r7
    adcs     r4, r4, r9
    stmia    r0, {r3,r4}

    adc      r0, r10, #0

    ldmfd    sp!,{r3-r10, pc}
}
```

■ bn_mul_shift_add_1_word(): $rp = rp + 2 \times ap \times w$ を求める (1 ワード)
・input:
BN_ULONG *rp: 計算結果を格納する開始アドレス
BN_ULONG *ap: 多倍長変数 ap のアドレス
int num: 多倍長変数 ap のワード数

BN_ULONG w: ワード長変数

・output:

戻り値 BN_ULONG: 乗算の繰り上がり値

ソースコード A.21: bn_mul_shift_add_1_word.arm

```
asm
BN_ULONG bn_mul_shift_add_1_word(
    register BN_ULONG *rp,
    register const BN_ULONG *ap,
    register BN_ULONG w)
{
    stmfd    sp!,{r3-r6, lr}

    eor      r6, r6, r6

    ldr      r3, [r1]
    umull    r4, r5, r3, r2
    movs     r4, r4, LSL #1
    adc      r5, r6, r5, LSL #1

    ldr      r6, [r0]
    adds     r6, r6, r4
    str      r6, [r0]

    adc      r0, r5, #0

    ldmfd    sp!,{r3-r6, pc}
}
```