RX ファミリ
DSP 機能命令を活用した多倍長乗算プログラム

R01AN0226JJ0100
Rev.1.00
2011.03.14

要旨

この文書は、RX ファミリの DSP 機能命令の使用方法を説明します。また、DSP 機能命令を使用した多倍長乗算プログラム例を紹介します。

動作確認デバイス

RX ファミリ

目次

1. はじめに

多倍長演算 (multiple precision arithmetic) とは、コンピュータのハードウェア命令で直接扱うことができる数の範囲を越える精度の数値計算のことです。例えば、RXのような 32 ビットマイクロコンピュータで直接扱うことができる数の範囲は 0 から $2^{32} - 1 = 4294967295$ に制限されます (符号無し整数として見た場合)。このようなハードウェアの制限を越える数値の計算をしたい場合には多倍長演算を実行するプログラムが必要になります。一般に、多倍長演算はハードウェアの固定精度の計算では精度が不足する場合や、計算結果のオーバフローが問題になるような場合に使用されます。多倍長演算の典型的な応用は公開鍵暗号です。公開鍵暗号のアルゴリズムでは非常に大きな桁数の整数演算が必要になります。

本アプリケーションノートでは、RX ファミリ CPU コア (以下 RX と略します) の積和演算命令の応用事例として多倍長数の乗算プログラムについて説明します。乗算に加えて、加算、減算、除算の多倍長演算プログラムについても示しますので、これらを合せて多倍長数の四則演算プログラムとして使用できます。

なお、RX の積和演算命令の詳細は「RX ファミリ ユーザーズマニュアル ソフトウェア編 (RJJ09B0465)」を参照ください。また、多倍長演算のアルゴリズムの詳細は次に示す文献を参照ください。

【注】 [1] D. E. Knuth, Seminumerical Algorithms, The Art of Computer Programming, Vol. 2, pp.265-284, 3rd edition, Addison Wesley, 1997.

2. 多倍長数のデータ表現

コンピュータのハードウェアが直接扱うことができる範囲を越える数値を多倍長数 (multiple precision number) と呼びます。本章では多倍長演算の対象となる多倍長数のデータ表現方法について説明します。

多倍長数は符号無し整数とします。RX には 16 ビット ×16 ビットの積和演算命令があります。この積和演算命令を活用して多倍長数の乗算プログラムを実現したいので、ここでは多倍長数を符号無し 16 ビット整数の配列で表現することにします。

符号無し 16 ビット整数は 2^{16}個の数を表現できます。そこで配列の各要素を 2^{16}進記法の「数字」と見なすと、長さNの符号無し 16 ビット整数配列によって 2^{16}進記法でN桁の整数を表現できます。なお、N桁の「数字」は配列の中で添字 0 の要素を最下位桁とし、最上位桁に向って昇順に並んでいるものと約束します。

以上を図 1 に示します。

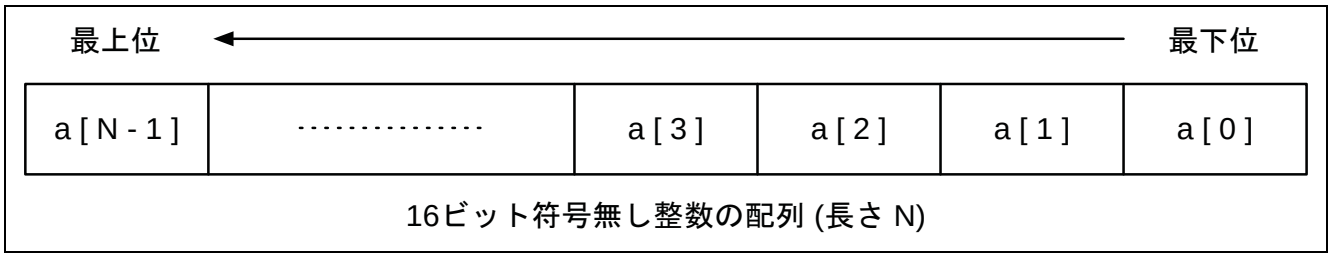

図 1　16 ビット符号無し整数配列による多倍長数のデータ表現

多倍長数 a[N]に格納されている数値は次の式で表わすことができます。

$$a[N-1] \times 2^{16 \times (N-1)} + \ldots + a[2] \times 2^{32} + a[1] \times 2^{16} + a[0]$$

次に図 1 に対応する多倍長数のC言語プログラム例を示します。このプログラムでは桁数Nを 32 として、最大 $2^{512} - 1$ までの大きさの符号無し整数を扱う場合を示しています。

```
#include <stdint.h>
#define N      32      /* 多倍長数の長さ (符号無し 16 ビット整数配列)  */
uint16_t a[N];
```

例えば、十進記法の 12345678901234567890 は次の C 言語の初期値付き配列として表現できます。初期値が指定されていない配列の要素 (上位の桁) はすべて 0 になることに注意してください。

```
uint16_t num[N] = { 0x0ad2, 0xeb1f, 0xa98c, 0xab54 };
```

3.　多倍長数の乗算

本章では RX の積和演算命令を使った多倍長数の乗算について説明します。

3.1　　16 ビット符号無し整数の乗算

多倍長数を 16 ビット符号無し整数の配列で表現するので、多倍長乗算を実現するための基本演算として 16 ビット符号無し整数の乗算が必要になります。RX には 32 ビット ×32 ビットの乗算命令と 16 ビット ×16 ビットの積和演算命令がありますが、いずれの命令も符号付きの乗算命令です。これに対して、多倍長数は符号無し整数なので、16 ビット ×16 ビットの符号無し整数の乗算が必要になります。そこで、16 ビット ×16 ビットの符号無し整数の乗算を RX の積和演算命令で作成します。

基本的なアイディアを図 2 に示します。ポイントは、16 ビット符号無し整数の被乗数と乗数を最上位ビット (b15 のみ) と最上位ビット以外の部分 (b14 から b0 まで) に分解して考えることです。つまり、16 ビット符号無し整数の被乗数 a と乗数 b の乗算を次の 4 つの部分の和として考えます：

1. a の下位 15 ビットと b の下位 15 ビットの積
2. a の最上位ビットが 1 のとき b の下位 15 ビットを左に 15 ビットシフトした値を加える
   (a の最上位ビットと b の下位 15 ビットの積に等しい)
3. b の最上位ビットが 1 のとき a の下位 15 ビットを左に 15 ビットシフトした値を加える
   (b の最上位ビットと a の下位 15 ビットの積に等しい)
4. a と b の最上位ビットがいずれも 1 のとき 0x40000000 を加える
   (a の最上位ビットと b の最上位ビットの積に等しい)

上の 1.の a の下位 15 ビットと b の下位 15 ビットの積を RX の積和演算命令 MULLO で実行します。MULLO 命令は 16 ビットの符号付き乗算を行いますが、両方のオペランドも 15 ビットの符号無し整数ですので問題ありません。

16 ビット符号無し整数の乗算関数 mul16 のプログラムを次に示します。この関数は、符号無し 16 ビット整数 a と b の乗算結果を 32 ビット符号無し整数で返します。なおこの関数はアセンブリ言語で記述されています。そのため#pragma inline_asm 宣言を使用します。

```
/*
  符号無し 16 ビット整数の乗算
  結果は符号無し 32 ビット整数で返す
 */
#pragma inline_asm mul16
static uint32_t mul16 (uint16_t a, uint16_t b)
{
       push.l  r6
       mov.l   r1,r3
       and     #7fffh,r3
       mov.l   r2,r4
       and     #7fffh,r4
       mov.l   #0,r5
       tst     #8000h,r1
       bz      ?+
       mov.l   r4,r6
       shll    #15,r6
       add     r6,r5
  ?:
       tst     #8000h,r2
       bz      ?+
       mov.l   r3,r6
       shll    #15,r6
       add     r6,r5
       tst     #8000h,r1
       bz      ?+
       add     #40000000h,r5
  ?:
       mullo   r3,r4
       mvfacmi r1
       add     r5,r1
       pop     r6
}
```

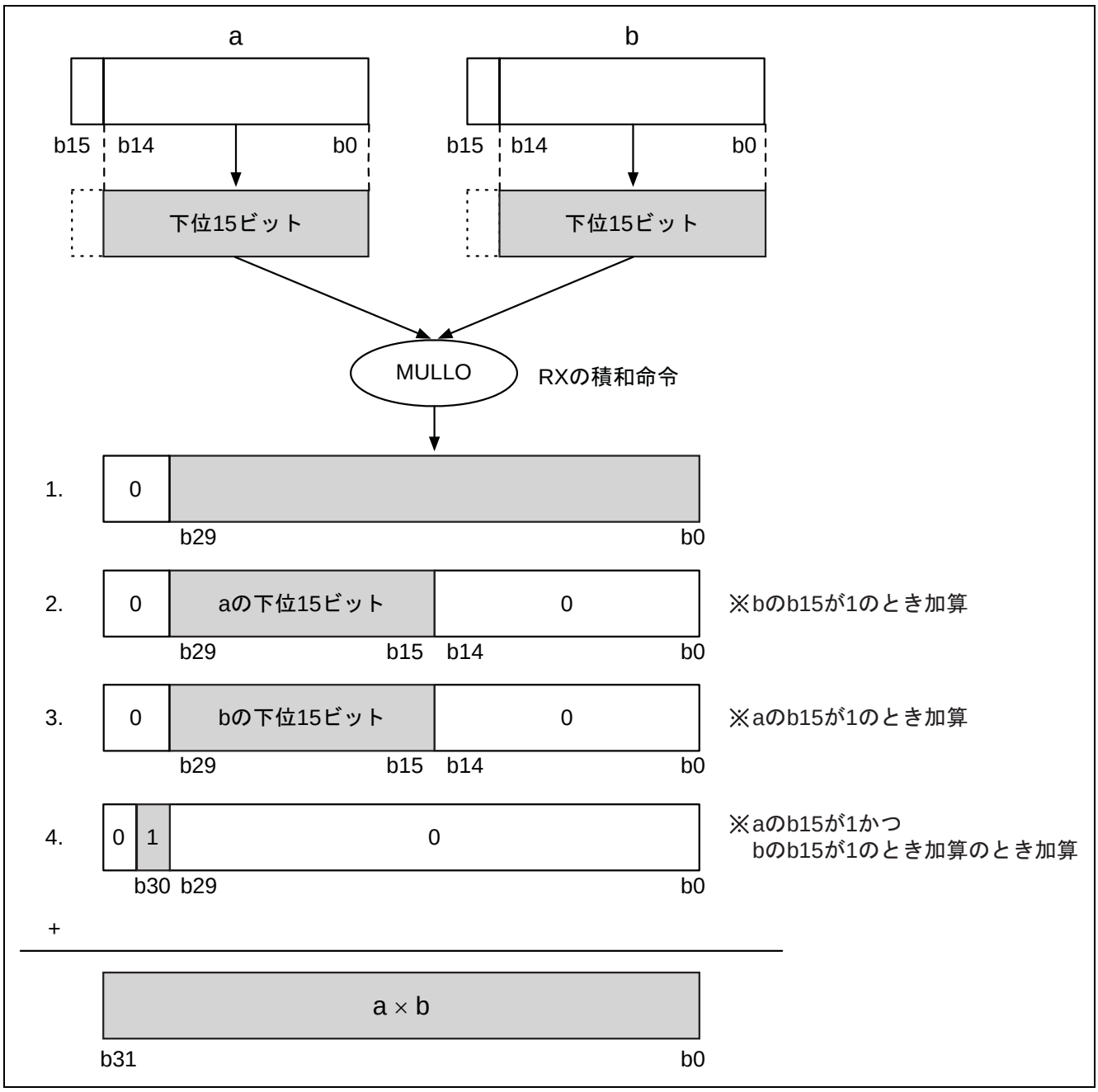

図 2　16 ビット × 16 ビットの符号無し整数の乗算

3.2　筆算による多倍長数の乗算

多倍長数の乗算は筆算によって計算します。多倍長数を N 桁の 16 ビット符号無し整数と考えて、乗数のおのおのの桁を被乗数の各桁に掛けた結果 (32 ビット符号無し整数) を所定の位取りの位置に順次加算します。筆算による 4 桁の多倍長数の乗算の例を図 3 に示します。

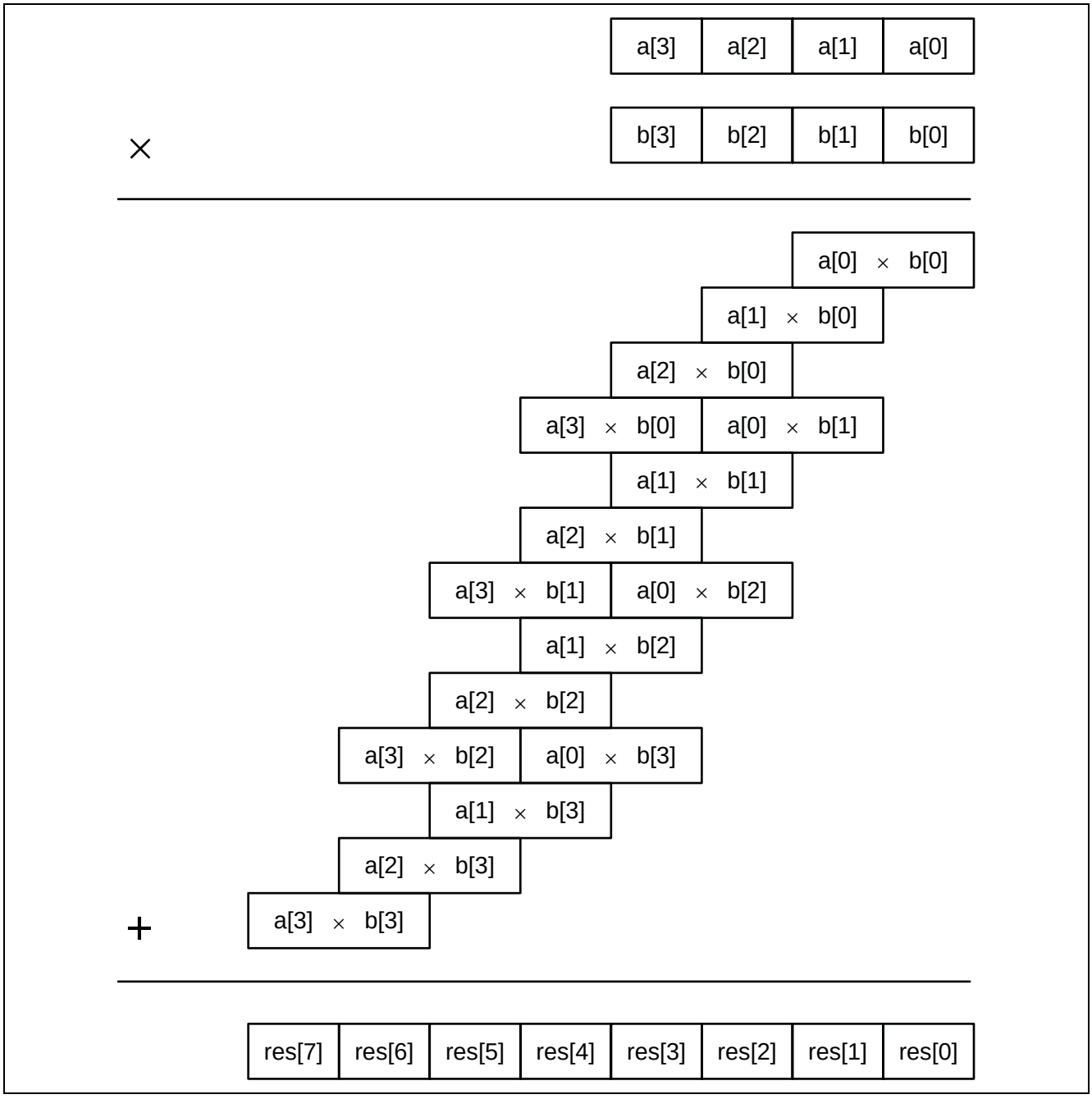

図 3　筆算による多倍長乗算の例 (4 桁 × 4 桁)

3.3　乗算プログラム

筆算によって多倍長数の乗算を行う関数 long_mul のプログラムを次に示します。この関数は多倍長数 a と b の乗算を行い、結果を a に格納します。

```
/*
  多倍長数の乗算
  a に結果を格納する
 */
void long_mul (uint16_t *a, uint16_t *b)
{
    int i, j;
    uint32_t x;
    uint16_t res[N];

    memset (res, 0, sizeof res) ;
    for  (i = 0; i < N; i++)  {
        if  (a[i] != 0)  {
            for  (j = 0; j < N; j++)  {
                if  (b[j] != 0 && i + j < N)  {
                    x = mul16 (a[i], b[j]) ;
                    add16 (res, i + j,  (x & 0xffff) ) ;
                    add16 (res, i + j + 1,  (x >> 16) ) ;
                }
            }
        }
    }
    memcpy (a, res, sizeof res) ;
}
```

関数 long_mul は、最初に結果を格納する変数 res をリセットしてから、被乗数 a と乗数 b の各桁を順番に掛け算して変数 res へ加算していきます。ただし、値が 0 の桁と結果が N 桁に納まらない場合は計算をスキップします。最後に変数 res に得られた結果を a にコピーします。

次に、関数 long_mul から呼び出している補助関数 add16 のプログラムを示します。この関数は多倍長数 a の i 桁目に符号無し 16 ビット整数 b を加算します。

```
/*
  符号無し 16 ビット整数 b を多倍長数 a の i 桁目に加算する
 */
static void add16 (uint16_t *a, int i, uint16_t b)
{
    uint32_t c;

    for  (c = b ; c > 0 && i < N; i++)  {
        c += a[i];
        a[i] = c;  // c の下位 16 ビットのみ転送される
        c >>= 16;  // c の上位 16 ビットには桁上りの値が入っている
    }
}
```

4.　多倍長数の四則演算

本章では、多倍長数の四則演算のうち前章で説明した乗算を除いた残りの 3 つの演算プログラムを示します。

- 加算
- 減算
- 除算

4.1　加算プログラム

本節では、多倍長数の加算関数 long_add のプログラムを示します。この関数は、多倍長数 a に多倍長数 b の値を加えた結果を a に格納します。なお,この関数はアセンブリ言語で記述されています。そのため#pragma inline_asm 宣言を使用します。

```
/*
  多倍長数の加算
  a に結果を格納する
 */
#pragma inline_asm long_add
void long_add (uint16_t *a, uint16_t *b)
{
       mov.l   #0,r4
       mov.l   #N,r5
  ?:
       movu.w  [r1],r3
       add     r3,r4
       movu.w  [r2+],r3
       add     r3,r4
       mov.w   r4,[r1+]
       shlr    #16,r4
       sub     #1,r5
       bnz     ?-
}
```

4.2　減算プログラム

本節では、多倍長数の減算関数 long_sub のプログラムを示します。この関数は、多倍長数 a から多倍長数 b の値を引いた結果を a に格納します。ただし、a ≥ b でなければなりません。なお、この関数はアセンブリ言語で記述されています。そのため #pragma inline_asm 宣言を使用します。

```
/*
  多倍長数の減算 (ただし、a >= b でなければならない)
  a に結果を格納する
 */
#pragma inline_asm long_sub
void long_sub (uint16_t *a, uint16_t *b)
{
       mov.l   #0,r4
       mov.l   #N,r5
  ?:
       movu.w  [r1],r3
       add     r3,r4
       movu.w  [r2+],r3
       sub     r3,r4
       mov.w   r4,[r1+]
       shar    #16,r4
       sub     #1,r5
       bnz     ?-
}
```

もう一つ減算に関連した演算として、多倍長数の比較関数 long_cmp のプログラムを次に示します。この関数は、2 つの多倍長数 a と b を比較し、a = b の場合に 0 を、a < b の場合に–1 を、a > b の場合に 1 を返します。

```
/*
  多倍長数の比較
  a > b なら 1、a == b なら 0, a < b なら -1 を返す
 */
int long_cmp (uint16_t *a, uint16_t *b)
{
    int i;
    int32_t c;

    for  (i = N - 1; i >= 0; i--)  {
        c =  (int32_t) a[i] -  (int32_t) b[i];
        if  (c < 0)  {
            return -1;
        }
        if  (c > 0)  {
            return 1;
        }
    }
    return 0;
}
```

4.3　除算プログラム

本節では、多倍長数の除算関数 long_div のプログラムを示します。この関数は、多倍長数 a の値を多倍長数 b の値で割り算をして、商を q、剰余を r に格納します。ただし、b > 0 でなければなりません。

```
static uint32_t guess (uint16_t *a, uint16_t *b, int c, int d) ;
/*
  多倍長数の除算 (ただし、b > 0 でなければならない)
  q に商を、r に剰余をそれぞれ返す
 */
void long_div (uint16_t *a, uint16_t *b, uint16_t *q, uint16_t *r)
{
    int i, m, n, shift;
    uint32_t u, quot;
    uint16_t c[N], d[N], e[N];

#define ZERO (x)         memset (x, 0, sizeof (uint16_t)  * N)
#define COPY (x, y)      memcpy (x, y, sizeof (uint16_t)  * N)

    /* initialize */
    ZERO (e) ;
    ZERO (q) ;
    COPY (r, a) ;
    n = llen (b)  - 1;
    if  (long_cmp (a, b)  < 0 || n < 0)  {
        return;
    }
    /* normalize */
    for  (shift = 0, u = b[n];  (u & 0x8000)  == 0; u <<= 1)  {
        shift++;
    }
    lshl (r, shift) ;
    lshl (b, shift) ;
    /* loop */
    while  (long_cmp (r, b)  >= 0)  {
        m = llen (r)  - 1;
        if  (r[m] >= b[n])  {
            ZERO (c) ;
            for  (i = 0; i <= n; i++)  { c[m - n + i] = b[i]; }
            if  (long_cmp (r, c)  >= 0)  {
                q[m - n] = 1;
                long_sub (r, c) ;
                continue;
            }
        }
        quot = guess (r, b, m, n) ;
        ZERO (c) ;
        for  (i = 0; i <= n; i++)  { c[m - n - 1 + i] = b[i]; }
        COPY (d, c) ;
        e[0] = quot;
        long_mul (c, e) ;
        while  (long_cmp (r, c)  < 0)  {
            long_sub (c, d) ;
            quot--;
        }
        q[m - n - 1] = quot;
        long_sub (r, c) ;
```

```
    }
    /* unnormalize */
    lshr (r, shift) ;
    lshr (b, shift) ;

#undef  ZERO
#undef  COPY
}

#pragma inline_asm guess
static uint32_t guess (uint16_t *a, uint16_t *b, int c, int d)
{
       shll    #01h,r3,r5
       add     r1,r5
       movu.w  [r5],r1
       sub     #02h,r5
       shll    #10h,r1
       add     [r5].uw,r1
       movu.w  [r4,r2],r5
       divu    r5,r1
       cmp     #0ffffh,r1
       bleu    ?+
       mov.l   #0ffffh,r1
  ?:
}
```

なお、上の除算プログラムは、既に説明した乗算関数、減算関数、比較関数の他に次の 3 つの補助関数を使用します：

- 多倍長数の左ビットシフト (lshl)
- 多倍長数の右ビットシフト (lshr)
- 多倍長数の桁数の取得 (llen)

最初に、多倍長数の左シフト関数 lshl のプログラムを示します。この関数は、多倍長数 a を左に n ビットシフトします。ただし、0 ≤ n ≤ 15 でなければなりません。

```
/*
  多倍長数 a を左へ n ビットシフト
  ただし、0 <= n <= 15 でなければならない
 */
static void lshl (uint16_t *a, int n)
{
    int i;
    uint32_t c = 0;
    uint32_t t;

    if  (n == 0)  {
        return;
    }
    for  (i = 0; i < N; i++)  {
        t =  (uint32_t) a[i];
        t <<= n;
        t |= c;
        a[i] = t;
        c =  (t >> 16) ;
    }
}
```

次に、多倍長数の右シフト関数 lshr のプログラムを示します。この関数は、多倍長数 a を右に n ビットシフトします。ただし、0 ≤ n ≤ 15 でなければなりません。

```
/*
  多倍長数 a を右へ n ビットシフト
  ただし、0 <= n <= 15 でなければならない
 */
static void lshr (uint16_t *a, int n)
{
    int i;
    uint16_t c = 0;
    uint16_t t;

    if  (n == 0)  {
        return;
    }
    for  (i = N - 1; i >= 0; i--)  {
        t = a[i];
        a[i] =  (c |  (t >> n) ) ;
        c =  (t <<  (16 - n) ) ;
    }
}
```

最後に、多倍長数の桁数を返す関数 llen のプログラムを示します。この関数は、多倍長数 a の桁数を返します。ただし、a = 0 の場合は 0 を返します。

```
/*
  多倍長数の桁数を返す
  ただし、a == 0 のときは 0 を返す
 */
static int llen (uint16_t *a)
{
    int i;

    for  (i = N - 1; i >= 0; i--)  {
        if  (a[i] != 0)  {
            return i + 1;
        }
    }
    return 0;
}
```

5. サンプルプログラム

多倍長演算の簡単なサンプルとして 35 の階乗を求めるプログラムを次に示します。

```
void main (void)
{
    int i;
    uint16_t a[N];
    uint16_t b[N];
    uint16_t c[N];

    memset (a, 0, sizeof a) ;
    memset (b, 0, sizeof b) ;
    memset (c, 0, sizeof b) ;
    a[0] = 1;
    b[0] = 2;
    c[0] = 1;
    for  (i = 0; i < 35 - 1; i++)  {
        long_mul (a, b) ;
        long_add (b, c) ;
    }
    /* a <- 35! = 10333147966386144929666651337523200000000 */
}
```

ホームページとサポート窓口

ルネサス エレクトロニクスホームページ
http://japan.renesas.com/

お問合せ先
http://japan.renesas.com/inquiry

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2011.03.14	—	初版発行

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 未使用端子の処理
   【注意】未使用端子は、本文の「未使用端子の処理」に従って処理してください。
   CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。未使用端子は、本文「未使用端子の処理」で説明する指示に従い処理してください。
2. 電源投入時の処置
   【注意】電源投入時は，製品の状態は不定です。
   電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。
   外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。
   同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. リザーブアドレス（予約領域）のアクセス禁止
   【注意】リザーブアドレス（予約領域）のアクセスを禁止します。
   アドレス領域には、将来の機能拡張用に割り付けられているリザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
4. クロックについて
   【注意】リセット時は、クロックが安定した後、リセットを解除してください。
   プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。
   リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
5. 製品間の相違について
   【注意】型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。
   同じグループのマイコンでも型名が違うと、内部 ROM、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ輻射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA機器、通信機器、計測機器、AV機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制するRoHS指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注2. 本資料において使用されている「当社製品」とは、注1において定義された当社の開発、製造製品をいいます。

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口 http://www.renesas.com

※営業お問合せ窓口の住所・電話番号は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス販売株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル） (03)5201-5307

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：http://japan.renesas.com/inquiry

Colophon 1.0