RX Family

Multiple precision Multiplication Program Making Use of the DSP Functions

R01AN0226EJ0100
Rev.1.00
Mar 14, 2011

Introduction

This application note explains how to use the instructions specific to the RX Family DSP functions. Coding examples of a multiple precision multiplication program using the DSP function specific instructions are also introduced.

Target Device

RX Family

Contents

1. General

Multiple precision arithmetic refers to numerical computations on numbers of a precision higher than that which can be handled directly with the hardware instructions of a computer. The range of numbers that can be handled directly by a 32-bit microcomputer such as the RX family is limited to from 0 to 2^{32} – 4294967295 (assuming that the numbers are unsigned integers). A program that carries out multiple precision arithmetic is required to make calculations beyond such hardware limitations. Generally, multiple precision arithmetic is used in applications in which the precision provided by hardware-implemented fixed-precision arithmetic would be inadequate or any overflows of computation results would bring about some problems. A typical application of multiple precision arithmetic is public key encryption. The public key encryption algorithm entails integer computations with a great number of digits.

This application note describes a multiple precision multiplication program as an application example of the multiply-accumulate instruction of the RX family CPU core (hereafter referred to as the RX). In addition to the multiplication program, the note also illustrates multiple precision arithmetic programs for addition, subtraction, and division. These programs may be combined and used as a four-function multiple precision arithmetic program.

For details on the RX's multiply-accumulate instructions, refer to RX Family User's Manual; Software (REJ09B0435). Reference should also be made to the document listed below for details on the multiple precision arithmetic algorithm.

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

2. Data Representation Multiple precision Numbers

Numbers of which precision exceeds the limits of values that a computer hardware can handle directly are called multiple precision numbers. This chapter describes the data representation of multiple precision numbers that are subject to multiple precision arithmetic.

The multiple precision numbers are assumed to be unsigned integers. The RX has a 16 bits × 16 bits multiply-accumulate instruction. To take advantage of this multiply-accumulate instruction to implement a multiplication program for multiple precision numbers, a multiple precision number is assumed to be represented by an array of 16-bit unsigned integers in this application note.

A 16-bit unsigned integer representation can represent 2^{16} numbers. By regarding each element of the array as a digit in a notation system of base 2^{16}, an array of 16-bit unsigned integers with a length of N can be used to represent an integer of N digits in a notation system of base 2^{16}. In addition, it is predefined that the elements of the array for a number of N digits are arranged in the ascending order toward the uppermost digit, with the element with a subscript of 0 being designated as the lowermost digit.

This is illustrated in figure 1.

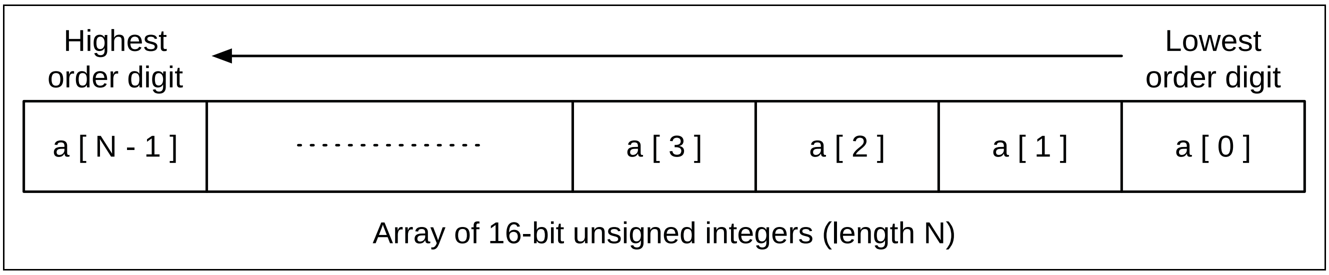

Figure 1 Data Representation of a Multiple precision Number Using an Array of 16-bit Unsigned Integers

The numerical value stored in the multiple precision number a[N] can be represented by the following expression:

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

An example of a C program fragment for a multiple precision number that corresponds to figure 1 is given below. In this program, the number of digits N is set to 32 and unsigned integers with a maximum of 2^{512} – 1 of magnitude are assumed to be handled.

```
#include <stdint.h>
#define N       32      /* Length of multiple precision number (for an array
of 16-bit unsigned integers) */
uint16_t a[N];
```

For example, 12345678901234567890 in decimal notation can be represented as a C language array with initial values as shown below. Note that the elements of the array for which no initial value is specified (upper digits) are all assumed to have a value of 0.

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

3. Multiplications of Multiple precision Numbers

This chapter discusses the multiplications of multiple precision numbers using the RX's multiply-accumulate instruction.

3.1 Multiplication of 16-bit Unsigned Integers

Since multiple precision numbers are represented by arrays of 16-bit unsigned integers, multiplication of 16-bit unsigned integers is required as one of the basic arithmetic operations for implementing multiple precision multiplications. The RX are provided with a 32 bits × 32 bits multiplication instruction and a 16 bits × 16 bits multiply-accumulate instruction but these are signed multiplication instructions. On the other hand, since multiple precision numbers are unsigned integers, multiplication of 16 bits × 16 bits unsigned integers is necessary. Accordingly, 16 bits × 16 bits unsigned integer multiplications are implemented using the RX's multiply-accumulate instruction.

The basic idea about this implementation is shown in figure 2. The point is to divide each of the 16-bit unsigned integer multiplicand and multiplier into the uppermost bit (b15 only) and the part other than the uppermost bit (b14 to b0). That is, we consider a multiplication of the 16-bit unsigned integer multiplicand a and multiplier b as the sum of the following four parts:

1. Product of the lower 15 bits of a and the lower 15 bits of b
2. If the uppermost bit of a is 1, add the value that is obtained by shifting the lower 15 bits of b 15 bits to the left (equal to the product of the uppermost bit of a and the lower 15 bits of b).
3. If the uppermost bit of b is 1, add the value that is obtained by shifting the lower 15 bits of a 15 bits to the left (equal to the product of the uppermost bit of b and the lower 15 bits of a).
4. IF the uppermost bit of both a and b is 1, add 0x40000000 (equal to the product of the uppermost bit of a and the uppermost bit of b).

Execute the product of the lower 15 bits of a and the lower 15 bits of b described in step 1 above using the RX's multiply-accumulate instruction MULLO. The MULLO instruction performs a 16-bit signed multiplication but causes no problem because both operands are 15-bit unsigned integers.

Given below is a sample program for the 16-bit unsigned integer multiplication function mul16. This function returns a 32-bit unsigned integer as the results of multiplying the 16-bit unsigned integers a and b. This function is coded in assembly language. Accordingly, the *#pragma inline_asm* declaration is used.

```
/*
  Multiplies 16-bit unsigned integers.
  Returns a 32-bit unsigned integer as the results.
 */
#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
}
```

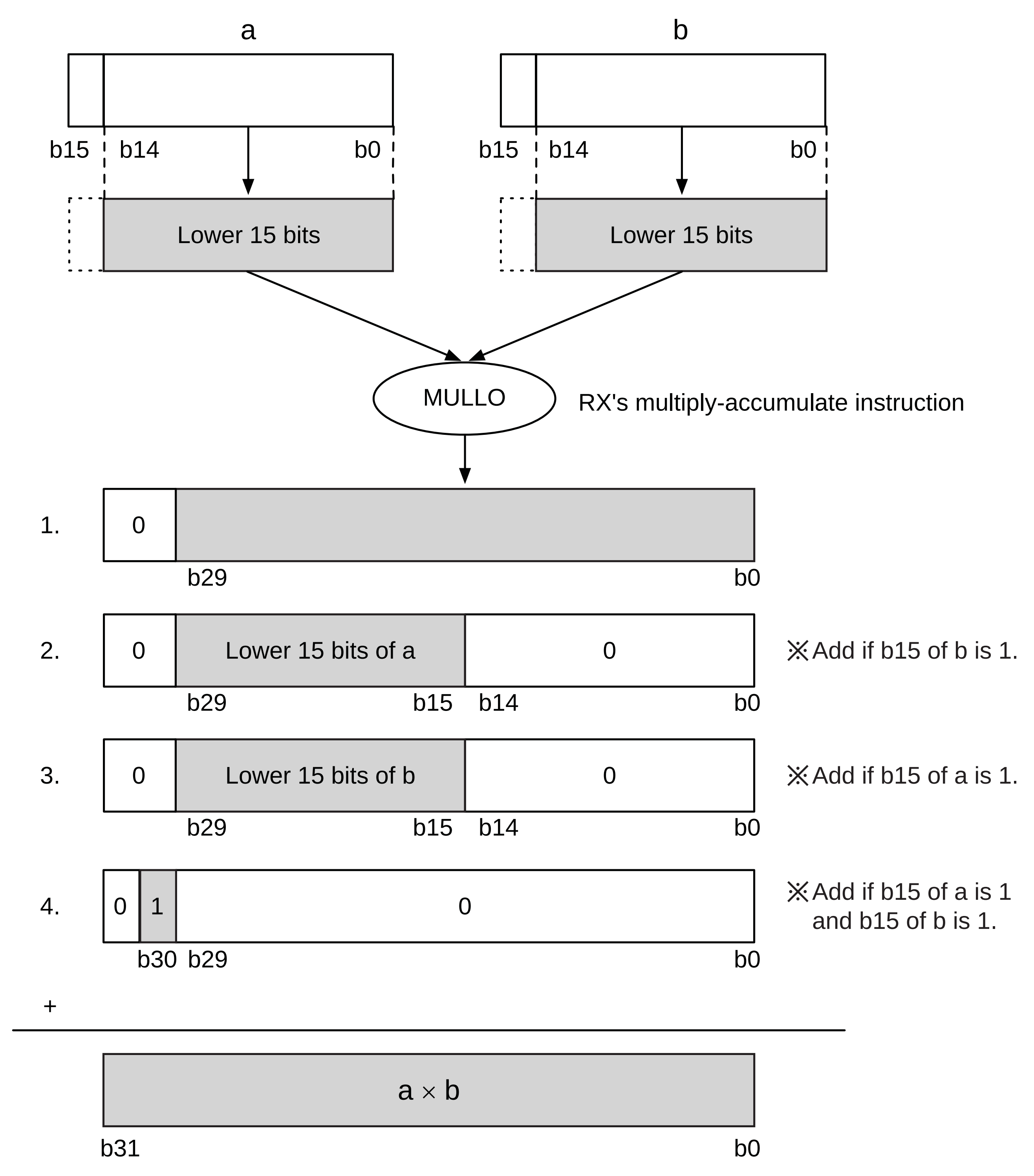

Figure 2 16 Bits × 16 Bits Unsigned Integer Multiplication

3.2 Multiplication of Multiple precision Numbers on Paper

This section explains the multiplication of multiple precision numbers on paper. Considering multiple precision numbers as 16-bit unsigned integers of N digits, add together the results of multiplying each digit of the multiplicand by each digit of the multiplier (32-bit unsigned integers) sequentially at their required digit position. Figure 3 shows an example of on-paper multiplication of 4-digit multiple precision numbers.

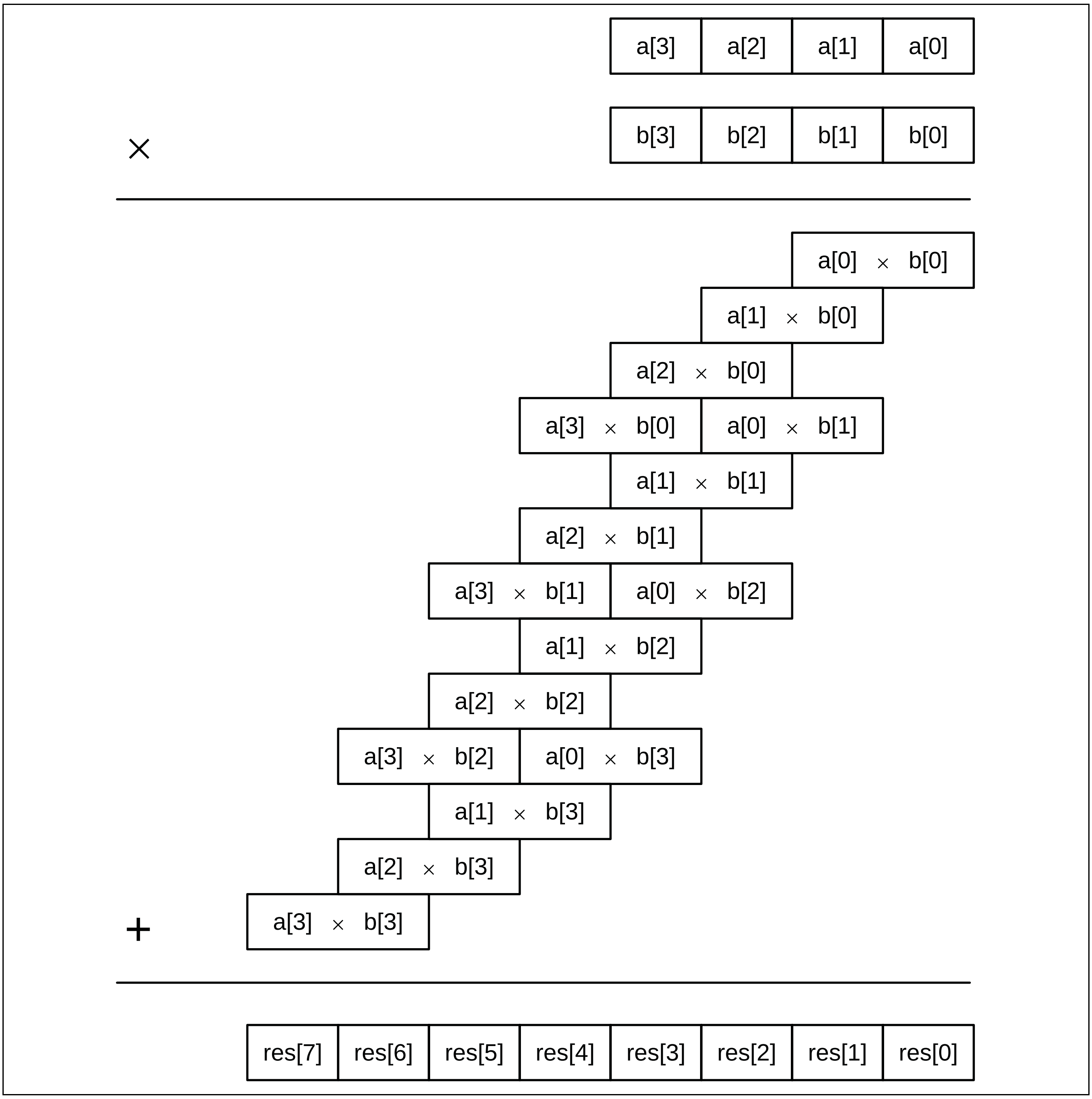

Figure 3 Example of Multiple precision Multiplication on Paper (4 Digits × 4 Digits)

3.3 Multiplication Programs

Given below is a sample program for the function long_mul that performs on-paper multiplications on multiple precision numbers. This function performs a multiplication on the multiple precision numbers a and b and places the results in a.

```
/*
  Multiplies multiple precision numbers.
  The results are placed in a.
 */
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);
}
```

The function long_mul initially resets the variable res for storing the results, then sequentially performs multiplications over the digits of multiplicand a and multiplier b, one digit at a time, and adds the intermediate results to the variable res. The function, however, skips any computation on the digit whose value is 0 or if the results will not fit in N digits. Finally, the function copies the results from the variable res to a.

Given below is a sample program for the auxiliary function add16 which is called by the function long_mul. This function adds the 16-bit unsigned integer b to the ith digit of multiple precision number a.

```
/*
  Adds 16-bit unsigned integer b to ith digit of multiple precision number a
 */
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;  // Only lower 16 bits of c are transferred.
        c >>= 16;  // Upper 16 bits of c hold the value of the carry.
    }
}
```

4. Four Arithmetic Operations on Multiple precision Numbers

This chapter introduces sample arithmetic operation programs that perform three of the four arithmetic operations on multiple precision numbers; except the multiplication which is discussed in the preceding chapter.

- Addition
- Subtraction
- Division

4.1 Addition Program

This section explains a sample program for the addition function long_add for multiple precision numbers. This function places the results of adding multiple precision number b to multiple precision number a in a. This function is coded in assembly language. Accordingly, the *#pragma inline_asm* declaration is used.

```
/*
  Adds together multiple precision numbers.
  Results are placed in a.
 */
#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 Subtraction Program

This section explains a sample program for the subtraction function long_sub for multiple precision numbers. This function places the results of subtracting multiple precision number b from multiple precision number a in a. However, the inequality a ≥ b must be observed. This function is coded in assembly language. Accordingly, the *#pragma inline_asm* declaration is used.

```
/*
  Subtraction on multiple precision numbers (a >= b must be observed)
  Results are placed in a.
 */
#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     ?-
}
```

Shown below is another sample program for the comparison function long_cmp for multiple precision numbers which is used to carry out the operation that is performed in conjunction with a subtraction. This function compares two multiple precision number a and b and returns 0 if a = b, –1 if a < b, and 1 if a > b.

```
/*
  Compares between multiple precision numbers
  Returns 1 if a > b, 0 if a == b, and -1 if a < b.
 */
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 Division Program

This section contains a sample program for the division function long_div for multiple precision numbers. This function divides the value of multiple precision number a by multiple precision number b and places the quotient in q and the remainder in r. However, the inequality b > 0 must be observed.

```
static uint32_t guess(uint16_t *a, uint16_t *b, int c, int d);
/*
  Performs division on multiple precision numbers (inequality b > 0 must be
observed).
  Places the quotient in q and the remainder in r.
 */
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
  ?:
}
```

The above division program uses the following three auxiliary functions in addition to the already-discussed multiplication, subtraction, and comparison functions:

- Bit-shift multiple precision number left (lshl)
- Bit-shift multiple precision number right (lshr)
- Get number of digits of multiple precision number (llen)

Firstly, a sample program for the left shift function lshl for multiple precision numbers is shown below. This function shifts multiple precision number a n bits to the left. The inequality $0 \leq n \leq 15$ must be observed.

```
/*
  Shifts multiple precision number a n bits to the left.
  0 <= n <= 15 must be observed.
 */
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);
    }
}
```

Given below is a sample program for the right shift function lshr for multiple precision numbers. This function shifts multiple precision number a n bits to the right. The inequality $0 \le n \le 15$ must be observed.

```
/*
 Shifts multiple precision number a n bits to the right.
  0 <= n <= 15 must be observed.
 */
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));
    }
}
```

Finaly, a sample program for the function llen for getting the length of multiple precision numbers is shown below.This function returns the number of digits of multiple precision number a. The function returns 0 if a = 0.

```
/*
  Returns number of digits of a multiple precision number.
  A 0 is returned if a == 0.
 */
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. Sample Program

Given below is an example of a simple multiple precision arithmetic program that finds the factorial of 35.

```
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 */
}
```

Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Mar 14, 2011	—	First edition issued

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Handling of Unused Pins

   Handle unused pins in accord with the directions given under Handling of Unused Pins in the manual.

   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of an MPU or MCU in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

RENESAS

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-65030, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141