Freescale Semiconductor
Application Note

AN3038
Rev. 0, 09/2005

Using the ColdFire EMAC Unit to Improve RSA Performance

by: Jim Stephens
Freescale Semiconductor

The widely used RSA public key cryptographic algorithm requires modular exponentiation of large integers (typically 512 to 2048 bits). A significant amount of work has been done on efficient methods for making this calculation. A critical element in a high performance RSA implementation is the modular multiplication function. This application note describes a technique for speeding up the modular multiplication using the enhanced multiply accumulate (EMAC) unit present on most ColdFire processors. A low level function is described that can be incorporated in RSA implementations to achieve a 2.5x improvement in multiplication performance.[1]

At a top level, the modular exponentiation function can be efficiently implemented using the square and multiply method or similar algorithms (m-ary, sliding window). This breaks the problem down to a series of modular multiplications. The Montgomery multiplication algorithm is an efficient way of calculating the required modular multiplications and squarings. The

Table of Contents

1. Based on timing of a 256-bit multiply running a ColdFire V2 processor using the Green Hills compiler v3.6.

Montgomery algorithm practically eliminates the need to perform division, but still requires efficient implementation of regular multiple precision multiplication.

There are various ways to implement Montgomery multiplication. However, the performance of an implementation is mostly dependent on the low level multiplication function used to calculate and accumulate partial products. An efficient function, mul_add64(), does a 1 by k digit multiply and addition using 64-bit digits. This function can be used in multiple precision multiplication, squaring, and reduction functions. Use of the mul_add64() function is illustrated using a regular multiple precision multiplication function. Application to squaring and reduction is straightforward.

The core of a multiple precision multiply function that makes the calculation t = a*b is shown below. The full function is in the file mp_mul.c.

```
for(i=size-1; i>=0; i--)
{
  carry = 0;
  for(j=size-1;j>=0;j--)
  {
    z0   = (a[i]&0xffff) * (b[j]&0xffff);
    z1a  = (a[i]&0xffff) * (b[j]>>16);
    z1b  = (a[i]>>16) * (b[j]&0xffff);
    z2   = (a[i]>>16) * (b[j]>>16);
    temp =  z0;
    temp += (ULLONG)(z1a) <<16;
    temp += (ULLONG)(z1b) <<16;
    temp += (ULLONG)(z2)  <<32;
    temp += (ULLONG) t[i+j+1] + carry;
    t[i+j+1] = (temp & 0xffffffffL);
    carry  = temp>>32;
  }
  t[i]=carry;
}
```

Regular Multiple Precision Multiply

In this example, the input and output values are represented by arrays of unsigned long (32-bit) elements, with the most significant element first. The calculation is made using 32-bit digits, with the inner loop performing a full 32-bit multiply.

To improve performance, an EMAC-based multiply and add function, mul_add64(), is used to replace the inner loop of the multiply function, as shown below. The full function is in the file mp_mul64.c.

```
for(i=size-1; i>=0; i--)
{
  carry = 0;
  mul_add64(&a[i], &b[0], size, &carry, &t[i+1]);
  t[i] = carry;
}
```

EMAC Based Multiple Precision Multiply

To make better use of the EMAC's capabilities, the digit size is increased to 64-bits. The input and output arrays (a, b, and t) use unsigned, long (64-bit) elements. Therefore, the value of the size input is reduced by a factor of two. The mul_add64() function performs a series of 64-bit multiplies and adds the products, as shown in the pseudocode below.

```
For i = k-1 to 0
  {c,z} = x*y[i] + t[i] + c
  t[i] = z
```

Pseudocode for mul_add64()

Note that all values except the counter are 64-bits, and the carry (c) is both an input and an output.

The mul_add64() function is implemented using the Comba method in ColdFire assembly language. All of the code inside the loop is completely unrolled. A full 64-bit multiply is implemented using 16 mac.w instructions. Using the Comba method, partial products can be accumulated directly. The only requirement to do this is an extension register to store carry bits, which the EMAC unit has. To optimize performance, instructions are scheduled such that results are not immediately read from the accumulator after a mac.w instruction. The code for this function can be found in the file mul_add64.s.

Benchmark tests show that a multiple precision multiply using the mul_add64() function is more than twice as fast as the 32-bit C implementation. This can be attributed to using the Comba method to take advantage of the ColdFire EMAC unit features. The mul_add64() function can be used directly or as a reference for developers of high performance RSA implementations on ColdFire processors.

References

1. C. Koc, “High-Speed RSA Implementation,” RSA Laboratories, 1994
2. C. Koc, T. Acar, B. Kaliski, “Analyzing and Comparing Montgomery Multiplication Algorithms,” IEEE Micro, 1996
3. P. Comba, “Exponentiation Cryptosystems on the IBM PC,” IBM Systems Journal, 1990

Appendix A
mp_mul64.c Code

```
/*
    mp_mul64.c
    multiple precision multiply using mul_add64 function
    t = a * b
    The size input is the length of a and b which have
    64-bit elements. Data is most significant element first.
*/

#define ULONG  unsigned long
#define ULLONG unsigned long long

void mul_add64(ULLONG *x, ULLONG y[], int k, ULLONG *c, ULLONG t[]);

void  mp_mul64(ULLONG a[],
               ULLONG b[],
               ULLONG t[],
               int size)
{
 int         i;
 ULLONG      carry;

  for(i=size; i<size*2; i++)
  {
    t[i] = 0;
  }

  for(i=size-1;i>=0;i--)
  {
    carry = 0;
    mul_add64(&a[i],&b[0],size,&carry,&t[i+1]);
    t[i] = carry;
  }

}
```

Appendix B
mul_add64.s Assembly File

```
; mul_add64.s
; Multiply and add function with 64 bit values.
; ColdFire assembly implementation using the EMAC and the
; Comba method. This function assumes k > 0
;
; void mul_add64(ULLONG *x, ULLONG y[], int k, ULLONG *c, ULLONG t[])
;                   8          12       16      20         24
; For i = k-1 to 0
;   {c,z} = x*y[i] + t[i] + c
;   t[i]  = z
;
; {a0,a1}  = x
; {a2,a3}  = y
; {d0,d1}  = c
; {d2,d3}  = z
; a5       = scratch ptr
; a6       = frame ptr
; d4,d5,d6 = scratch data reg
; d7       = 16 (for shifts)

   .globl   mul_add64

mul_add64:
   link.w   %a6,#-40
   movem.l  %d2-%d7/%a2-%a5,(%sp)

   moveq    #0x40,%d6           ; unsigned integer mode
   move.l   %d6,%macsr
   movclr.l %acc0,%d6           ; clear acc and ext
   movclr.l %acc1,%d6
   moveq    #16,%d7         ; d7 used for shifts
   move.l   8(%a6),%a5          ; load x {a0,a1}
   move.l   (%a5)+,%a0
   move.l   (%a5),%a1
   move.l   16(%a6),%d5         ; d5 = k
   subq.l   #1,%d5              ; d5 = k-1
   lsl.l    #3,%d5              ; d5 = 8*(k-1)
   add.l    %d5,12(%a6)         ; y_ptr = &y[k-1]
   add.l    %d5,24(%a6)         ; t_ptr = &t[k-1]
   move.l   20(%a6),%a5         ; load c (moved to z in loop)
   move.l   (%a5)+,%d0
   move.l   (%a5),%d1

loop:
   move.l   12(%a6),%a5         ; load y[i] {a2,a3}
   move.l   (%a5)+,%a2
   move.l   (%a5),%a3
   moveq    #8,%d5
```

```
sub.l    %d5,12(%a6)          ; y_ptr--
mac.w    %a1.l,%a3.l          ; x1*y1
move.l   %d0,%d2              ; move c to z
move.l   %d1,%d3
clr.l    %d0                  ; clear c
clr.l    %d1
movclr.l %acc0,%d6            ; d6 =  {z2,z1}
mac.w    %a1.l,%a3.u          ; x1*y2
mac.w    %a1.u,%a3.l          ; x2*y1
add.l    %d6,%d3              ; d3 += {z2,z1}  (add col 1)
clr.l    %d5
addx.l   %d5,%d2              ; d2 += carry
addx.l   %d5,%d1              ; d1 = carry
move.l   %accext01,%d4
movclr.l %acc0,%d5            ; d4:d5 = {z4,z3,z2}
mac.w    %a1.l,%a2.l          ; x1*y3
mac.w    %a1.u,%a3.u          ; x2*y2
mac.w    %a0.l,%a3.l; x3*y1
move.l   %d5,%d6              ; add col 2
lsl.l    %d7,%d4              ; d4 =  {z4,0}
lsr.l    %d7,%d5              ; d5 =  {0,z3}
lsl.l    %d7,%d6              ; d6 =  {z2,0}
add.l    %d4,%d5              ; d5 =  {z4,z3}
add.l    %d6,%d3              ; d3 += {z2,0}
addx.l   %d5,%d2              ; d2 += {z4,z3} + carry
clr.l    %d6
addx.l   %d6,%d1              ; d1 += carry
move.l   %accext01,%d4
movclr.l %acc0,%d5            ; d4:d5 = {z4,z3,z2}
mac.w    %a1.l,%a2.u          ; x1*y4
mac.w    %a1.u,%a2.l          ; x2*y3
mac.w    %a0.l,%a3.u; x3*y2
mac.w    %a0.u,%a3.l; x4*y1
add.l    %d5,%d2              ; d2 += {z3,z2} (add col3)
addx.l   %d4,%d1              ; d1 += {0,z4}
move.l   %accext01,%d4
movclr.l %acc0,%d5            ; d4:d5 = {c2,c1,z4}
mac.w    %a1.u,%a2.u          ; x2*y4
mac.w    %a0.l,%a2.l          ; x3*y3
mac.w    %a0.u,%a3.u; x4*y2
move.l   %d5,%d6              ; add col 4
lsl.l    %d7,%d4              ; d4 =  {c2,0}
lsr.l    %d7,%d5              ; d5 =  {0,c1}
lsl.l    %d7,%d6              ; d6 =  {z4,0}
add.l    %d4,%d5              ; d5 =  {c2,c1}
add.l    %d6,%d2              ; d2 += {z4,0}
addx.l   %d5,%d1              ; d1 += {c2,c1} + carry
addx.l   %d0,%d0              ; d0 = carry
move.l   %accext01,%d4
movclr.l %acc0,%d5            ; d4:d5 = {c3,c2,c1}
mac.w    %a0.l,%a2.u          ; x3*y4
mac.w    %a0.u,%a2.l          ; x4*y3
add.l    %d5,%d1              ; d1 += {c2,c1} (add col 5)
```

```
addx.l   %d4,%d0             ; d0 += {0,c3}
move.l   %accext01,%d4
movclr.l %acc0,%d5           ; d4:d5 = {c4,c3,c2}
mac.w    %a0.u,%a2.u         ; x4*y4
move.l   %d5,%d6             ; add col 6
lsl.l    %d7,%d4             ; d4 =  {c4,0}
lsr.l    %d7,%d5             ; d5 =  {0,c3}
lsl.l    %d7,%d6             ; d6 =  {c2,0}
add.l    %d4,%d5             ; d5 =  {c4,c3}
add.l    %d6,%d1             ; d1 += {c2,0}
addx.l   %d5,%d0             ; d0 += {c4,c3}
movclr.l %acc0,%d5           ; d5 =  {c4,c3} (add col 7)
add.l    %d5,%d0             ; d0 += {c4,c3}
move.l   24(%a6),%a5         ; a5 = t_ptr
add.l    4(%a5),%d3          ; add t[i]
move.l   (%a5),%d5
addx.l   %d5,%d2
clr.l    %d5
addx.l   %d5,%d1
addx.l   %d5,%d0
move.l   %d2,(%a5)           ; t[i] = z
move.l   %d3,4(%a5)
subq.l   #8,%a5              ; t_ptr--
move.l   %a5,24(%a6)
moveq    #1,%d5
sub.l    %d5,16(%a6)         ; k--
bne.w    loop                ; loop k times

move.l   20(%a6),%a5         ; store c
move.l   %d0,(%a5)+
move.l   %d1,(%a5)

movem.l  (%sp),%d2-%d7/%a2-%a5
unlk     %a6
rts
```

Appendix C
mp_mul.c Code

```
/*
    mp_mul.c
    multiple precision multiply using long data type
    t = a * b
    The size input is the length of a and b.
    Data is most significant element first.
*/

#define ULONG  unsigned long
#define ULLONG unsigned long long

void  mp_mul(ULONG a[],
             ULONG b[],
             ULONG t[],
             int size)
{
 int         i,j,k;
 ULONG       carry;
 ULONG       z0,z1a,z1b,z2;
 ULLONG      temp;

  for(k=size; k<size*2; k++)
  {
    t[k] = 0;
  }

  for(i=size-1;i>=0;i--)
  {
    carry = 0;
    for(j=size-1;j>=0;j--)
    {
      z0   = (a[i]&0xffff) * (b[j]&0xffff);
      z1a  = (a[i]&0xffff) * (b[j]>>16);
      z1b  = (a[i]>>16) * (b[j]&0xffff);
      z2   = (a[i]>>16) * (b[j]>>16);
      temp =  z0;
      temp += (ULLONG)(z1a) <<16;
      temp += (ULLONG)(z1b) <<16;
      temp += (ULLONG)(z2)  <<32;
      temp += (ULLONG) t[i+j+1] + carry;
      t[i+j+1] = (temp & 0xffffffffL);
      carry  = temp>>32;
    }
    t[i]=carry;
  }

}
```

This page intentionally left blank.

This page intentionally left blank.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN3038
Rev. 0, 09/2005

freescale™
semiconductor