Keywords: RSA, modular exponentiation, cryptography, secure microcontroller, MAXQ, MAA, modular arithmetic accelerator

APPLICATION NOTE 5430

Improving the Speed of Modular Exponentiation with DeepCover Secure Microcontrollers (MAXQ1050, MAXQ1850, and MAXQ1103)

Jun 18, 2012

Abstract: This application note describes how to improve the speed of modular exponentiation by more than 50% when using MAXQ® microcontrollers that have a modular arithmetic accelerator (MAA).

Introduction

Modular exponentiation, a^e modulo m, is a common operation in many cryptographic functions. The modular arithmetic accelerator (MAA) in MAXQ microprocessors can perform this operation with modulus sizes of up to 2048 bits. It is easy to load the memory areas with a, e, and m, and start the operation.

When the modulus is the product of two or more primes, we can use the results of the Chinese remainder theorem (CRT) to reduce the execution time by doing two smaller modular exponentiations instead of one large one. Specifically, we are using Garner's algorithm for this operation.

Description

In a typical RSA decryption operation, we recover our plain text (pt) from the cipher text (ct) by executing $pt = ct^d \bmod n$, where d and n form the private key. The value d is our decryption exponent, and n is the product of the primes p and q. In general, p and q will be the same number of bits in length, and n, pt, and ct will be twice that number of bits. For example, if p and q are 1024 bits long, then n will be a 2048 bit number about 60% of the time.

The CRT reduces our exponentiation to the following equations:

Let $c_1 = ct^d \bmod p$, $c_2 = ct^d \bmod q$, and let $m_1 = p(p^{-1} \bmod q)$ and $m_2 = q(q^{-1} \bmod p)$.

$$ct = (c_1 + m_1(c_2 - c_1)) \bmod n$$

or

$$ct = (c_2 + m_2(c_1 - c_2)) \bmod n.$$

Notice that now our modular exponentiations in the c_1 and c_2 terms will have half as many bits than with the ct^d mod n operation.

The terms m_1 and m_2 can both be precomputed. The p^{-1} mod q is some value, say y, such that p × y mod q = 1. For example, if p = 11 and q = 17, 11^{-1} mod 17 = 14, since 11 times 14 mod 17 = 1. These inverse values can be found using the extended Euclidian algorithm, or since p and q are primes, executing the function p^{q-2} mod q. This modular exponentiation to find an inverse is based on Fermat's little theorem.

The c_1 and c_2 terms are interesting because ct and d are both twice as large as their modulus values of p and q. The one thing the MAA cannot do is operate on values greater than the modulus size; we need to reduce both of these values before we can use the MAA to perform the modular exponentiation. The reduction of the exponent d relative to p and q can be precomputed. These new exponents are simply (d - 1) mod p and (d - 1) mod q. The reductions of the exponents is also based on Fermat's little theorem.

Reducing ct in both c_1 and c_2 is done at execution time with a modular multiply. For example, if the ct is 64 bits long and p and q are both 32 bits long, then we can perform the following multiplication: ct × 2^{32} mod (p × 2^{32}). This would be a 64-bit modular multiplication. This is actually simpler than the equation looks. We put the 64-bit, 4-word ct into the MAA register a and a clear everything in MAA register b. We then set bit 32 of register b, making the register equal 2^{32}. In the modulus we will write the bottom two words to zero, and then copy the value of p into the next two words. We then set MAWS to 64 and perform the modular multiplication. The reduced value we are looking for is in words 3 and 4 of the result.

To make things easier for the rest of the calculation, we find which term is the larger between c_1 and c_2, and then select the equation where we subtract the smaller from the larger to avoid getting a negative number. Now do a modulo multiplication, and then add this to c_1 (if we multiplied by m_1) or c_2 (if we multiplied by m_2).

Code Description

Listing 1 shows the initialization of unsigned long word pointers to each of the registers in the MAA. The hard-coded address for the MAA in the DeepCover® Secure Microcontroller (MAXQ1103) with secure RISC architecture is given.

Listing 1. Pointers to the MAA Registers

```
typedef unsigned long  int ulong;

// long word pointers to MAA memories in the MAXQ1103
ulong *maa_aw   = (ulong *) 0x8000;
ulong *maa_bw   = (ulong *) 0x8100;
ulong *maa_resw = (ulong *) 0x8200;
ulong *maa_expw = (ulong *) 0x8400;
ulong *maa_modw = (ulong *) 0x8500;
```

Listing 2 contains the four precomputed constants, piqtp (read as *p inverse q times p*), qiptq (read as *q* *inverse p times q*), dphip (read as *d phi of p*), and dphiq (read as d phi of q). The constants diptp and piqtq are the m_1 and m_2 terms from above. The constants dphip and dphiq are the reduced decryption exponents for the c_1 and c_2 terms from above. The two vectors ptp and ptq are for temporary storage. The other variables p, q, n, phi, e, d, pt, and ct describe all the values needed for RSA. The term phi is equal to (p - 1) × (q - 1).

The term nwords is the number of words in n, the modulus of the keys. In this implementation it is assumed that both p and q will have exactly nwords × 16 bits, and that n will have exactly nwords × 32 bits.

The words in these vectors are saved from the least significant word to the most significant word. They are loaded into the MAA register from low word to high word. To be clear, here is an example of p × q = n using the constants from Listing 2 with alternate long words in bold.

```
   0xF22F213FE34B717B × 0xC9446776B381BFB9 =
0xBE67B781405A57697217C6CFBB2AC6E3
```

Listing 2. RSA Constants

```
int   nwords = 0x4;
ulong ptp[0x2];
ulong ptq[0x2];

// complete set of RSA constants
ulong p[0x2] =     { 0xE34B717B, 0xF22F213F };
ulong q[0x2] =     { 0xB381BFB9, 0xC9446776 };
ulong n[0x4] =     { 0xBB2AC6E3, 0x7217C6CF, 0x405A5769, 0xBE67B781 };
ulong phi[0x4] =   { 0x245D95B0, 0xB6A43E19, 0x405A5767, 0xBE67B781 };
// keys
ulong e[0x4] =     { 0x00000005, 0x00000000, 0x00000000, 0x00000000 };
ulong d[0x4] =     { 0xB6B1448D, 0xC55031AD, 0x337B791F, 0x9852F934 };
// sample plain text and corresponding cipher text
ulong pt[0x4] =    { 0x90ABCDEF, 0x12345678, 0x90ABCDEF, 0x12345678 };
ulong ct[0x4] =    { 0xDA3C591A, 0xC131AD9D, 0x40A51B30, 0x361958DF };

// the four pre-computed values used in crt computation.
ulong piqtp[0x4] = { 0x50995949, 0x4D355F7A, 0x907F8CC5, 0x1F0F60BF };
ulong qiptq[0x4] = { 0x6A916D9B, 0x24E26755, 0xAFDACAA4, 0x9F5856C1 };
ulong dphip[0x2] = { 0x5AEAFA31, 0x60DFA6E6 };
ulong dphiq[0x2] = { 0x6BB43FD5, 0x78C2A47A };
```

Listing 3 has the do_crt function, which is called with the number of words in p or q. The routine starts off by creating the terms c_1 and c_2 from above and saving these values in ptp and ptq, respectively. Then we determine which is larger of ptp or ptq, and then call the routine that will do the modular multiplication and addition. This leaves the pt in the maa_resw memory.

Listing 3. The do_crt Routine

```
void do_crt(int nwords)
{  // nwords is the number of 32 bit words in p or in q.
```

```
   int i;

   mod_reduction(ct, p, dphip, nwords, ptp);
   mod_reduction(ct, q, dphiq, nwords, ptq);

   for (i = nwords - 1; i >= 0; --i)
      if (ptp[i] > ptq[i])
      {
         sum_mul_sub(2*nwords, ptp, ptq, qiptq, n);
         break;
      }
      else
      {
         sum_mul_sub(2*nwords, ptq, ptp, piqtp, n);
         break;
      }
}
```

Listing 4 has the details for initializing the MAA. Here we clear all 384 words in the MAA, initialize the memory select register, MAMS, and then tell the MAA which is the most significant bit in the modulus, MAWS.

Listing 4. The init_maa Routine

```
void init_maa(int mod_size)
{
   int i;

   for (i = 0; i < 384; ++i)
      maa_aw[i] = 0;           // clear the entire MAA
   MAMS = 0x6420;              // memory select register
   MAWS = mod_size;            // position of the most significant bit of
modulus.
}
```

Listing 5 has the details for doing the modular reduction for expressions of the form ptp = ct^{dphip} mod p. The first thing this routine does is reduce ct to half its size by doing a modular multiply with a shifted modulus p, and letting the multiplicand be the shifted value. These routines use moves of long words rather than shifting several words one bit at a time. (A move by a distance of a long word is like shifting 32 times.) After doing the modular multiplication, our reduced answer is shifted left in the maa_resw register.

With the reduced ct, we next do the modular exponentiation to get the result this subroutine was meant to obtain, ptp.

Listing 5. The mod_reduction Routine

```
void mod_reduction(ulong *ct, ulong *p, ulong *dphip, int nwords, ulong *ptp)
{
   int i;
   // nwords as passed is the length of p. (rather than n)
   // we are going to do a modmul, with a shifted p as the modulus
   // init_maa is initializing MAWS with the correct modulus size.
   init_maa(nwords*64);

   // reducing ct mod p by doing the modmul ct * 2^(nwords*32) mod (p *
(2^(nword*32))
   // load a with ct
   for (i = 0; i < 2*nwords; ++i)
      maa_aw[i] = ct[i];
   // load b with 2^(nwords*32) which is simply a bit set
   maa_bw[nwords] = 1;
   // load modulus with p*2^(nwords*32) which is simply a load shifted by
nwords.
   for (i = 0; i < nwords; ++i)
      maa_modw[i + nwords] = p[i];

   // this multiply gives us the reduction in ct and
   // the answer in maa_resw shifted by nwords.
   MACT = 0x05;      // mod multiply and start
   while (MACT & 1)  // wait for the multiply to finish
      ;
   // load registers to do ct^dphip mod p
   // notice that we are coping the shifted result of maa_resw to maa_aw.
   for (i = 0; i < nwords; ++i)
   {
      maa_aw[i] = maa_resw[nwords + i];
      maa_bw[i] = 0;
      maa_expw[i] = dphip[i];
      maa_modw[i] = p[i];
   }
   maa_b[0] = 1;      // the b reg is always 1 for modexp
   MAWS = 32*nwords;  // the most important step is setting MAWS to the
correct size

   MACT = 0x1;        // mod exp and start
   while (MACT & 1)
      ;
   // copy our result to the ptp argument.
   for (i = 0; i < nwords; ++i)
       ptp[i] = maa_resw[i];
}
```

Listing 6 describes the function that puts everything together. We are going to compute the equation ct = (c_1 + m_1(c_2 - c_1)) mod n if c_2 is larger than c_1 or ct = (c_2 + m_2(c_1 - c_2)) mod n otherwise. It starts off with the subtraction. Notice we are using n as our modulus, and we will be at the full key length for the multiply and add. After the subtraction, we move the result from maa_resw to maa_aw, and copy our multiplier, m_1 or m_2, our argument c into maa_bw, and start the modular multiply. In the last step, we copy the result of the multiply from maa_resw to maa_aw, and copy c_1 or c_2, our argument b, into maa_bw and do the modular addition. When it is done, our plain text is in maa_resw.

Listing 6. The sum_mul_sub Routine

```
void sum_mul_sub(int nwords, ulong *a, ulong *b, ulong *c, ulong *n)
{
   int i;

   // prepare to subtract b from a
   for (i = 0; i < nwords/2; ++i)
   {
       maa_aw[i]   = a[i];
       maa_bw[i]   = b[i];
       maa_modw[i] = n[i];
   }
   // clear the upper words of maa_a and maa_b and copy the rest of n
   for (i = nwords/2; i < nwords; ++i)
   {
       maa_aw[i]   = 0;
       maa_bw[i]   = 0;
       maa_modw[i] = n[i];
   }
   // this is a full size operation.
   // start the subtraction
   MAWS = 32*nwords;
   MACT = 0xB;    // subtract and start
   while (MACT & 1)
      ;
   // copy the result over to maa_aw and
   // put or multiplicand into maa_bw
   for (i = 0; i < nwords; ++i)
   {
       maa_aw[i] = maa_resw[i];
       maa_bw[i] = c[i];
   }
   MACT = 5;    // multiply and start
   while (MACT & 1)
      ;
   for (i = 0; i < nwords/2; ++i)
   {
      maa_aw[i] = maa_resw[i];
      maa_bw[i] = b[i];
   }
   for (i = nwords/2; i < nwords; ++i)
   {
      maa_aw[i] = maa_resw[i];
      maa_bw[i] = 0;
   }
   MACT = 0x9;    // add and start
   while (MACT & 1)
      ;
}
```

Performance and Conclusion

Tables 1 and **2** give an indication of the speed improvements that can be realized by using the algorithm discussed above. We get greater reductions in time as the modulus size increases.

The C implementation presented here is intended to demonstrate that a speed increase is possible using this algorithm. The code also shows how the MAA is manipulated. Many things can be done to increase the speed of the algorithm, including loop unrolling, loop optimization, using the compilers optimized data movement routines, and using assembly language.

The quickest and easiest improvement in speed would be to manipulate the MAA's MAMS register, which will eliminate some data movement. The MAMS register lets us rename memory segments. It was not done in this application, for simplicity.

Theoretically, it should be possible to get close to a timing ratio of 4 to 1 in the improved speed by using this algorithm.

Always use the crypto ring as your crypto clock source. This is accomplished by clearing the Master Crypto Source Select (MCSS, PMR.7) in the Power Management Register (PMR). When decrypting, it is recommended that the Optimized Calculation Control (OCALC, MACT.4) of the Modular Arithmetic Accelerator Control Register (MACT) be cleared to zero, disabling the function.

Table 1. MAXQ1103 at 25MHz with the MAA Running with Crypto Ring

Size	ModExp (ms)	CRT (ms)	Ratio
2048	549	166	3.3
1024	82.0	29.6	2.8
512	14.2	7.25	2.0
256	3.37	3.08	1.1

Table 2. DeepCover Secure Microcontroller (MAXQ1050) at 24MHz with the MAA Running with Crypto Ring

Size	ModExp (ms)	CRT (ms)	Ratio
2048	1760	492	3.6
1024	244	75.7	3.2
512	37.1	14.3	2.6
256	6.80	4.49	1.5

Numerical Example of RSA Using the CRT to Recover the Plain Text

This example goes through the steps of constructing the public and private keys for RSA, then taking a sample message, encrypting it, and decrypting it. Then we show how the same encrypted message can be decrypted using the CRT.

We start by finding a couple of prime numbers. Let p = 0xE747 and q = 0xC7A5. Both are 16-bit prime numbers.

This gives us n = p × q = 0xB45D41C3 and phi = (p - 1)(q - 1) = 0xB45B92D8. Both are 32 bits.

We can pick e = 0x10001 since gcd(e, phi) = 1. This gives us our public key. Our private key is picked so e × d mod phi = 1. Using the extended Euclidian algorithm, we compute d = 0x9B111CC9.

We arbitrarily pick our plain text, pt = 0xABCDEF12. Our cipher text, ct = pt^{e} mod n = 0x87CCFE27. To recover the plain text, execute ct^{d} mod n. This is a 32-bit modular exponentiation. This in principle is the RSA encryption/decryption process.

Now we will recover the plain text by using Chinese remainder theorem.

Offline we precompute four constants. The first is piqtp = 0x9E1D261C, which is (p^{-1}mod q) × p. The second is qiptq = 0x16401BA8, which is (q^{-1}mod p) times q. These are both 32 bits long. You can use the extended Euclidian algorithm or execute p^{q-2} mod q and q^{p-2} mod p to find the inverses. We also need dphip = d mod phi(p) = 0x9B111CC9 mod (0xE747 - 1) = 0x4AAB and dphiq = d mod phi(q) = 0x9B111CC9 mod (0xC7A5 - 1) = 0x9A0D. These numbers are both 16 bits, half the size of n.

The online calculation starts by reducing the cipher text by mod p. The cipher text is 32 bits long and our modulus is 16 bits long. To use the MAA to do the reduction, we shift the modulus, p, left 16 bits and multiply the cipher text by 216. That looks like 0x87CCFE27 × 0x10000 mod 0xE7470000 = 0x36B00000. Now we shift the answer right 16 bits or just grab the upper 16 bits of the word. We need this value to do the 16-bit modular exponentiation of $0x36B0^{dphip}$ mod p, which looks like $0x36B0^{0x4AAB}$ mod 0xE747 = 0x6425 = ptp.

The second modular reduction with cipher text mod q expands as 0x87CCFE27 × 0x10000 mod 0xC7A50000 = 0x543D0000. Shift the answer right 16 bits, or just grab the upper word, getting the reduction, 0x543D. Use this result and perform the modular exponentiation $0x543D^{dphiq}$ mod q, which looks like $0x543D^{0x9A0D}$ mod 0xC7A5 = 0x1671 = ptq.

If ptp is greater than ptq, we do the calculation (ptq + (ptp - ptq) × qiptq) mod n, otherwise we calculate (ptp + (ptq - ptp) × piqtp) mod n. Remember that piqtp and qiptq were precomputed and are 32 bits long.

We see that ptp is larger than ptq. The difference, ptp - ptq = 0x4DB4. The product of ((ptp - ptq) and qiptq) mod n, and (0x4DB4 × 0x16401BA8) mod 0xB45D41C3 is 0xABCDD8A1. Adding ptq, 0x1671 gives us back our plain text, 0xABCDEF12, and we are finished.

Related Parts		
MAXQ1050	DeepCover Secure Microcontroller with USB and Hardware Cryptography	
MAXQ1103	DeepCover Secure Microcontroller with Rapid Zeroization Technology and Cryptography	
MAXQ1850	DeepCover Secure Microcontroller with Rapid Zeroization Technology and Cryptography	Free Samples

More Information
For Technical Support: http://www.maximintegrated.com/support
For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact