Fast Machine Code for Modular Multiplication

Michael Scott
School of Computer Applications
Dublin City University
20th July 1995

Abstract

Techniques are described to speed up the calculation of modular products, which are required at the heart of software implementations of many cryptographic protocols such as the well-known RSA Public Key algorithm**.** By combining ideas from Comba [1] and Dusse and Kaliski [2], and exploiting Montgomery's method [4] for modular reduction, very efficient machine code routines can be generated. The method is illustrated for the popular 80386/80486/Pentium Family of processors, but uses only a very small subset of the instruction set, and so can easily be adapted to many other conventional processors with a word length of 16 bits or more.

Introduction

In exponential Public Key Cryptographic systems, such as the well known RSA method, there is a requirement for fast modular exponentiation, that is the calculation of

$$a = b^e \bmod n$$

where for acceptable levels of security a, b, e, and n are large multiprecision numbers. The algorithm of choice is the so-called m-ary method, described by Knuth [3]. This reduces the above calculation to a sequence of squarings and multiplications, modulo n, the total number of which will be not much larger than the number of bits in the binary representation of e. Therefore fast modular exponentiation can be achieved if we have access to fast methods for modular squaring and multiplication.

The problem can conveniently be divided into two phases; the squaring or multiplication of two large multiprecision numbers, and then their subsequent reduction to a remainder when divided by n.

Consider the multiplication of two m digit numbers, x and y to form a product z. Using subscripts for each individual digit, so that x_1 is the least significant digit of x, and x_m is the most significant digit, then the product can be represented as a pyramid of partial products thus:-

$$
\begin{aligned}
z_1 &= x_1y_1 \\
z_2 &= x_1y_2 + x_2y_1 \\
z_3 &= x_1y_3 + x_2y_2 + x_3y_1 \\
z_4 &= x_1y_4 + x_2y_3 + x_3y_2 + x_4y_1 \\
&\vdots \\
z_m &= x_1y_m + x_2y_{m-1} + \ldots\ldots + x_my_1 \\
&\vdots \\
z_{2m-1} &= x_my_m \\
z_{2m} &= 0
\end{aligned}
$$

If the numbers $\boldsymbol{x}$ and $\boldsymbol{y}$ are represented to base b, then, in general, the value obtained for each z_i may be greater than b. Indeed it is easy to see that the largest z_i can get nearly as big as $m.b^2$. To get $\boldsymbol{z}$ back into the form of a standard base b number, we must propagate or release the "carries". In the standard school-boy method this propagation of carries is normally carried out as the multiplication proceeds from right-to-left.

The most efficient way to implement this simple algorithm to multiply large numbers on a binary computer, is to make each digit as large as possible. To this end it is assumed in the sequel that the numbers $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ will be represented to a number base of $b=2^w$, where w is the wordlength of the computer in bits. This implies that each z_i, prior to releasing the carries, requires triple precision representation.

For the ultimate in speed, Comba [1] suggests implementing this algorithm directly to calculate the product. Key ideas are to completely unravel all program loops (many modern pipelining and cache memory mechanisms heavily penalise program jumps), and to minimise the number of memory accesses by only writing a digit of the result to memory when it has been completely evaluated. In this way a 50% speed advantage was obtained on a 16-bit 8086 processor over the more intuitive looping code (a double nested loop, each to be executed m times) that most programmers would instinctively use.

Three registers are reserved to store the accumulated z_i as we work through the columns from right to left. Comba provides actual code for the 8086 16-bit processor.

The following representative code fragment is from Comba [1], modified in a trivial way for a 32-bit 80386.

```
;ebx -> x, esi -> y, edi-> z

          push edi        ;re-use this register
          xor  ecx,ecx    ;Note ch=0 always
          xor  edi,edi    ;sets edi=0
          xor  ebp,ebp    ;z_i is calculated in cl|edi|ebp
;first column
........
........
;fourth column

          mov  eax,[ebx]            ;get x_1, multiply by y_4
          mul  DWORD PTR [esi+12]   ;result in edx|eax
          add  ebp,eax              ;propagate carries
          adc  edi,edx
          adc  cl,ch

          mov  eax,[ebx+4]          ;get x_2, multiply by y_3
          mul  DWORD PTR [esi+8]
          add  ebp,eax
          adc  edi,edx
          adc  cl,ch

          mov  eax,[ebx+8]          ;get x_3, multiply by y_2
          mul  DWORD PTR [esi+4]
          add  ebp,eax
          adc  edi,edx
          adc  cl,ch

          mov  eax,[ebx+12]         ;get x_4, multiply by y_1
          mul  DWORD PTR [esi]
          add  ebp,eax
          adc  edi,edx
          adc  cl,ch

          mov  edx,edi
          pop  edi
          mov  [edi+12],ebp         ;store result z_4
          push edi
          mov  ebp,edx              ;carry to next column
          mov  edi,ecx
          xor  cl,cl

;etc
```

It should be pointed out that the same technique could be used in a very similar fashion on many other contemporary processors. Only a very small subset of the instruction set is being used.[1] The only significant requirement is that the processor should support a fast *wxw* bit unsigned multiply operation, which allows access to both the most and least significant *w* bits of the *2w* bit product generated. This will be true of most - but certainly not all - current processors.

[1] The `XOR` and `PUSH/POP` instructions could be implemented using `MOV`s (on an 80386 this will be slightly less efficient). So only four basic instructions are required - `MOV`, `ADD`, `ADC` and `MUL`.

Squaring can be carried out with just over half the number of partial products required for full multiplication. Rearranging the "pyramid of products" above, and setting $y_i=x_i$ then

$z_1 = x_1x_1$
$z_2 = 2.x_1x_2$
$z_3 = 2.x_1x_3 + x_2x_2$
$z_4 = 2.x_1x_4+2.x_2x_3$
$z_5 = 2.x_1x_5+2.x_2x_4+x_3.x_3$
z_6 = ...
.....

An obvious pattern appears. This can be exploited in various ways; the method used here avoids using any new instructions.

```
;ebx -> x, esi-> z
          xor ecx,ecx     ;Note ch=0 always
          xor edi,edi
          xor ebp,ebp     ;z_i is calculated in cl|edi|ebp

; first column
.......
.......
; fourth column

          mov  eax,[ebx]            ;get x_1, multiply by x_4
          mul  DWORD PTR [ebx+12]   ;result in edx|eax
          add  ebp,eax              ;propagate carries
          adc  edi,edx
          adc  cl,ch

          add  ebp,eax              ;double it
          adc  edi,edx
          adc  cl,ch

          mov  eax,[ebx+4]          ;get x_2 multiply by x_3
          mul  DWORD PTR [ebx+8]
          add  ebp,eax
          adc  edi,edx
          adc  cl,ch

          add  ebp,eax              ;double it
          adc  edi,edx
          adc  cl,ch

          mov  [esi+12],ebp         ;store result z_4
          mov  ebp,edi
          mov  edi,ecx              ;carry to next column
          xor  cl,cl

;etc
```

Montgomery's Method

Comba [1] goes on to describe a rather ad-hoc mechanism for reducing a product to its remainder modulo n. Here we advocate a different approach.

In [4] Montgomery describes a novel way of achieving modular reductions without requiring any division operation whatsoever. However this first requires the transformation of all the problem variables (except the exponent) into a special N-residue form. Then once the full calculation is complete the answer can be converted back to "normal" form.

Choose a value $\boldsymbol{R}$ as $1000....00_b$, which is just greater than the odd modulus $\boldsymbol{n}$. Calculate and store $\boldsymbol{n'} = -(1/\boldsymbol{n}) \bmod \boldsymbol{R}$ using the extended Greatest Common Divisor algorithm [3]. Numbers are then converted to the N-residue form by multiplying them by $\boldsymbol{R} \bmod \boldsymbol{n}$.

In this form there is a quick algorithm for the computation of $\boldsymbol{t}=\boldsymbol{TR}^{-1} \bmod \boldsymbol{n}$, for a number $\boldsymbol{T} < \boldsymbol{Rn}$:-

> Find a number $\boldsymbol{k}<\boldsymbol{R}$ such that $\boldsymbol{t}=(\boldsymbol{T}+\boldsymbol{kn})/\boldsymbol{R}$ is a whole number.
> (So $\boldsymbol{tR} \equiv \boldsymbol{T} \bmod \boldsymbol{n}$, and $\boldsymbol{t}=\boldsymbol{TR}^{-1} \bmod \boldsymbol{n}$.)
> If $\boldsymbol{t} \geq \boldsymbol{n}$ then return $(\boldsymbol{t}-\boldsymbol{n})$, else return $\boldsymbol{t}$

Such a number $\boldsymbol{k}$ can be quickly calculated as $\boldsymbol{k}=\boldsymbol{Tn'} \bmod \boldsymbol{R}$. The modular reduction here is cheap because of the simple form of $\boldsymbol{R}$.

For N-residues, a modular multiplication can be obtained using this algorithm. Converting $\boldsymbol{x}$ and $\boldsymbol{y}$ to N-residue form gives $\boldsymbol{xR} \bmod \boldsymbol{n}$ and $\boldsymbol{yR} \bmod \boldsymbol{n}$. Their product $\boldsymbol{T}$, calculated in the normal way, is congruent to $\boldsymbol{xyR}^2 \bmod \boldsymbol{n}$. Applying the above algorithm, yields $\boldsymbol{t}=(\boldsymbol{xy})\boldsymbol{R} \bmod \boldsymbol{n}$, which is clearly the N-residue form of the product mod $\boldsymbol{n}$.

This method for modular multiplication has the remarkable property that it requires no division whatsoever (once $\boldsymbol{n'}$ is calculated and the numbers involved are converted to N-residue form). The modular reduction is achieved at the cost of two multiplications of large numbers, one in the calculation of $\boldsymbol{k}$ and the other in the calculation of $\boldsymbol{kn}$.

However it is possible to do even better if using multiprecision arithmetic to base b. Montgomery [4] goes on to suggest a variation that requires the calculation of only m^2+m partial products for an m-digit $\boldsymbol{n}$. Here we suggest an alternative Comba-style variation which will be even quicker on many processors.

Consider the calculation of $\boldsymbol{t}=(\boldsymbol{T}+\boldsymbol{kn})/\boldsymbol{R}$. To illustrate, consider $\boldsymbol{R}=10000_b$

					n_4	n_3	n_2	n_1
					k_4	k_3	k_2	k_1
					k_1n_4	k_1n_3	k_1n_2	k_1n_1
				k_2n_4	k_2n_3	k_2n_2	k_2n_1	0
			k_3n_4	k_3n_3	k_3n_2	k_3n_1	0	0
		k_4n_4	k_4n_3	k_4n_2	k_4n_1	0	0	0
	T_8	T_7	T_6	T_5	T_4	T_3	T_2	T_1
	c_7	c_6	c_5	c_4	c_3	c_2	c_1	
t_5	t_4	t_3	t_2	t_1	0	0	0	0

Here the c_i denote a possible carry from the previous column, and t_5 is either a 0 or 1.

To calculate k_1 we must ensure that $T_1+n_1k_1 \equiv 0$ mod b. Therefore set $k_1 = T_1.n'$ mod b. Note that this single precision number n' (= $-(1/n_1)$ mod b) may be precalculated once and for all from the least significant digit of $\boldsymbol{n}$. In the second column we must select k_2 such $c_1 + T_2+ k_1n_2 + k_2n_1 \equiv 0$ mod b. So in this case $k_2 = (c_1 + T_2+ k_1n_2)n'$ mod b. Once k_1, k_2, k_3 and k_4 are calculated in this way, the remaining columns containing t_1, t_2, t_3, t_4 and t_5 are calculated in a straightforward fashion, proceeding as for a normal multiprecision multiplication (but not forgetting to add in the T_i).

In general, when $\boldsymbol{n}$ is an m digit number, each of the first m columns are processed by calculating the associated k_i to ensure that the column sum (after the carry) is 0. Once $\boldsymbol{k}$ has been fully evaluated in this way, the required answer $\boldsymbol{t}$ is obtained from the most significant columns.

It is easy to see that if $\boldsymbol{T<Rn}$, and $\boldsymbol{k<R}$, then $(\boldsymbol{T+kn}) < 2$. . Therefore $\boldsymbol{t}<2.\boldsymbol{n}$. So one extra subtraction of $\boldsymbol{n}$ from $\boldsymbol{t}$ may be needed to get $\boldsymbol{t}$ into the range $0 \le \boldsymbol{t} < \boldsymbol{n}$. After this step $t_5 = 0$. Alternatively [2] by choosing an $\boldsymbol{n}$ such that $\boldsymbol{n < R/4}$, then we can tolerate numbers in the redundant range $[\boldsymbol{0,2n-1}]$ as their product $\boldsymbol{T}$ will always be in the valid range $\boldsymbol{0 \le T < 4n^2 < Rn}$. In this case t_5 would be 0 always, and the extra subtraction is unnecessary, except perhaps at the completion of the exponentiation process when we need to normalise the final result into the range $[\boldsymbol{0,n-1}]$.

In the Comba-style implementation given below, the digits of $\boldsymbol{T}$ are overwritten to save space, first by the digits of $\boldsymbol{k}$, as they are calculated, and then by the digits of $\boldsymbol{t}$. The following fragment shows the code for the 4th and 5th columns of the modular reduction (just as we pass the mid-point of the calculation and change gear from calculating $\boldsymbol{k}$ to calculating $\boldsymbol{t}$), with $\boldsymbol{R}=10000_b$.

Note that in this case we run out of registers on the rather register-deficient 80386. So the precalculated value for n' is now taken instead from a memory location.

```
; ebx -> T, esi -> n

          xor  edi,edi
          xor  ecx,ecx              ;Note ch=0 always
          mov  ebp,[ebx]            ;get T1

; first column
........
........
; fourth column

          mov  eax,[ebx]            ;get k1 multiply by n4
          mul  DWORD PTR [esi+12]   ;result in edx|eax
          add  ebp,eax              ;propagate carries
          adc  edi,edx
          adc  cl,ch

          mov  eax,[ebx+4]          ;get k2 multiply by n3
          mul  DWORD PTR [esi+8]
          add  ebp,eax
          adc  edi,edx
          adc  cl,ch

          mov  eax,[ebx+8]          ;get k3 multiply by n2
          mul  DWORD PTR [esi+4]
          add  ebp,eax
          adc  edi,edx
          adc  cl,ch


          mov  eax,ebp              ;multiply by n' mod b
          mul  DWORD PTR ndash      ;calculate k4
          mov  [ebx+12],eax         ;store k4
          mul  DWORD PTR [esi]      ;get k4 multiply by n1
          add  ebp,eax
          adc  edi,edx
          adc  cl,ch

          mov  ebp,edi              ;carry to next column
          mov  edi,ecx
          xor  ecx,ecx
          add  ebp,[ebx+16]         ;add T5
          adc  edi,ecx

; 5th column

          mov  eax,[ebx+4]          ;get k2 multiply by n4
          mul  DWORD PTR [esi+12]   ;result in edx|eax
          add  ebp,eax              ;propagate carries
          adc  edi,edx
          adc  cl,ch

          mov  eax,[ebx+8]          ;get k3 multiply by n3
          mul  DWORD PTR [esi+8]
          add  ebp,eax
          adc  edi,edx
          adc  cl,ch
```

```
            mov  eax,[ebx+12]         ;get k4 multiply by n2
            mul  DWORD PTR [esi+4]
            add  ebp,eax
            adc  edi,edx
            adc  cl,ch

            mov  [ebx+16],ebp         ;store t1
            mov  ebp,edi              ;carry to next column
            mov  edi,ecx
            xor  ecx,ecx
            add  ebp,[ebx+20]         ;add T6
            adc  edi,ecx

; etc
```

This code will only take very slightly longer than a 4-digit by 4-digit multiprecision multiplication. Only one extra `mul` and a couple of extra `mov` and `add` instructions are required for the first 4 columns. For the final 4 columns, only two extra `add` instructions are needed. No extra instructions are required over and above those already used in the multiplication/squaring code.

Implementation

The routines outlined above have been implemented on a 60Mhz Pentium-based PC. Since no looping is carried out, a very large amount of code can be generated for a fixed length modulus ***n***. However the features of a modern macro-assembler can be exploited to alleviate the tedium of having to write so much repetitive code.

In the RSA algorithm [3] the modulus ***n*** is the product of two prime factors ***p*** and ***q***. The most time consuming operation is a decryption, or signature, operation which requires the calculation of a full length exponentiation. However since the decryption is being carried out by the possessor of the factors of the modulus ***n*** - the private key - the Chinese Remainder theorem (see Knuth [3]) can be exploited so that only two half-length exponentiations are required, assuming the two factors of ***n*** are about the same size. So for example using a 1024-bit modulus which is the product of two 512-bit prime factors, the decryption time would mostly consist of the time for two modular exponentiations of two 512-bit numbers, with 512-bit exponents and a 512-bit modulus.

The exponentiation method used is Knuth's *m*-ary method. The sliding-windows variation as described by Comba [1] is used, with a maximum window size of 3 bits. This has not been highly optimised, and might be somewhat improved.

Exploiting the Chinese Remainder Theorem, the following timings were obtained for some popular modulus sizes. There is a small overhead incurred by the C language control code used. This becomes insignificant for the larger moduli. Nevertheless it is felt that these times are close to the best possible that can be achieved in software with this processor and with a modulus that is not of a special form.

RSA Modulus	Decrypt Time (s)	Throughput (bps)	Object file size
512	0.047 seconds	10894 bits/sec	4673 bytes
768	0.125	6144	8646
1024	0.26	3938	14033
1536	0.78	1969	29130
2048	1.73	1170	49894

Discussion

The method described for calculating modular products can be easily modified to work with many other 16, 32 and 64-bit processors. The simple subset of instructions used should make conversion to another processor a relatively easy task. However it will require a lot of memory (ROM).

As is well known the school-boy algorithm for multiplication and squaring is not asymptotically optimal. For example a popular recursive algorithm due to Karatsuba [3] requires the calculation of $O(m^{lg3})$ partial products for multiplying two m digit numbers, compared to the $O(m^2)$ complexity of the school-boy algorithm. However Comba reports that in practice Karatsuba does not confer an advantage until numbers at least 28 digits in length are being multiplied. This is due to the unavoidable control overhead and the many extra additions and subtractions that are required to implement the algorithm. However this implies that if the numbers are much greater than 28 digits in length, the method described here will no longer be the fastest possible. Note that for a 2048 bit RSA modulus on a 32 bit processor (using the Chinese Remainder Theorem), the numbers involved are 32 digits in length. In this case using Karatsuba's method for the multiplication/squaring algorithms might yield a very small speed advantage of a few percent at most.

The performance cross-over obviously depends on the Computer word-length. On a 64-bit computer, all other things being equal, Karatsuba's method would not break-even until an RSA modulus of 4096 bits or more were in use.

Note that there appears to be no Karatsuba-like way to speed up the variant of the Montgomery reduction algorithm described above. However for extremely large moduli the Karatsuba algorithm might be applied to the original Montgomery algorithm which required two full length multiplications.

Most current implementations of the RSA public key system recommend a modulus size of 512 to 1024 bits. So in practise the approach described here may in fact be close to optimal for many processors which have a word length of 16-bits or more.

References

[1] COMBA, P.G. Exponentiation Cryptosystems on the IBM PC. IBM Systems Journal, Vol. 29, No. 4 1990, 526-538

[2] DUSSE, S.R. and KALISKI, B.S. A Cryptographic Library for the Motorola DSP56000, In Advances in Cryptology, Eurocrypt '90, Lecture Notes in Computer Science, Vol. 473, Springer-Verlag 1991, 230-244

[3] KNUTH, D.E. The Art of Computer Programming. Vol. 2: Seminumerical Algorithms. Addison-Wesley, Reading, Mass., 1981

[4] MONTGOMERY, P. Modular Multiplication Without Trial Division. Math. Comp. , 44, (April 1985), 519-521