Analysis of MMX Technology for Implementing Number-Theoretic Cryptosystems

Final Report - February 12, 1997

Dr. Cetin K. Koc
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331
Koc@ece.orst.edu

1. Table of Contents

2. Abstract

We have analyzed the instruction set and related architectural features of the MMX technology for obtaining high-speed implementations of the number-theoretic cryptographic algorithms. In this study, we have mainly concentrated on the RSA and DSS algorithms, however, the findings of this study is equally applicable and highly relevant to other number-theoretic cryptographic systems. We have implemented the RSA and DSS algorithms on the MMX platform, and obtained timings on a 233 MHz Pentium Pro/MMX processor. These timings indicate that the number-theoretic cryptographic algorithms implemented using the MMX instructions are significantly *slower* than those implemented using Pentium instructions. This turns out to be mainly due to the fact that the MMX architecture performs signed arithmetic and lacks certain instructions such as 16-bit and 32-bit unsigned multiply and multiply-add. After carefully examining the kernel of operations for the number-theoretic cryptographic algorithms, we propose 8 new instructions for the next generation MMX technology. We also estimate the performance of the RSA and DSS algorithms assuming subsets of these proposed instructions were made available.

3. High-Speed Implementations of Cryptosystems

The privacy and authenticity of information (whether it is stored on a single computer or shared on a network of computers) requires implementation of cryptographic functions. The basic functions of information security services are very few, and almost invariant. These include public-key cryptosystems, digital signatures, message-digest functions, and secret-key cryptographic algorithms. The subject of this study is *algorithm engineering*, which refers to high-speed and cost effective hardware and software implementation of cryptographic algorithms [BG89]. Among these cryptographic algorithms, we are mainly interested in obtaining high-speed implementations of public-key cryptosystems and digital signatures based on number-theoretic methods. These algorithms require operations with elements of a large finite group, and need to be optimized on the chosen platform for high-speed implementation. As an example, the RSA cryptographic algorithm [RSA78] uses modular arithmetic operations (addition, multiplication, and exponentiation) with large integers, usually in the range of 512 to 1024 bits. Arithmetic with such large integers is time consuming, considering the fact that currently available processors have arithmetic logic units with wordsize up to 32 bits. The current fast implementations of the RSA signature algorithm with a 1024-bit key size require about 500 msec on a signal processor [DK90] and about 100 msec on the 120-Mhz Pentium PC using advanced algorithmic techniques and assembly language programming [K96]. Other cryptographic algorithms, for example, the Diffie-Hellman key exchange method [DH76], the ElGamal public-key cryptosystem and digital signature algorithm [E85], the Digital Signature Standard [N91] also require implementation of modular arithmetic operations with large integers.

Software implementations of these number-theoretic cryptographic algorithms are often desired because of their flexibility and cost effectiveness. Furthermore, certain applications are suitable for software implementations because of their very nature. However, the performance is always an issue, requiring the designer to optimize these cryptographic algorithms on the selected processor. In order to exploit the architectural and arithmetic-logic properties of the processor, the designer needs to analyze and reformulate the underlying algorithms. Almost inevitably, the programming is performed in assembly language in order to take advantage of the specific architectural properties of the processor, and thus, to obtain the desired performance.

We have been studying the number-theoretic cryptosystems in order to obtain their fast implementations on Intel Pentium Processors since August 1995 under a grant from Intel Research Council. Recently we have focused on investigating the use of MMX technology for obtaining high-speed implementations of number-theoretic cryptosystems, mainly the RSA and DSS algorithms. The findings of this study are also relevant in obtaining high-speed implementations of all number-theoretic cryptosystems with the exception of elliptic curve cryptosystems [K87,M93] based on the field GF(2^k). On the other hand, most secret-key cryptographic algorithms (DES, IDEA, RC4, RC5, etc.) and message-digest functions (MD5, SHA, etc.) do not use number-theoretic techniques, and thus, a separate study needs to be performed in order to examine the applicability of MMX technology for these algorithms.

4. Montgomery Multiplication

The most crucial routine in obtaining high-speed implementation of the RSA and DSS algorithm is the Montgomery product [KAK96], which involves the computation of $a*b*r^{-1}$ mod n given the k-bit integers a, b, and n, where $r=2^k$. The integer k is usually between 512 and 1024. The modulus is an odd number in the range 2^{k-1} to 2^k. The Montgomery product method involves modulo 2^k multiplications and division by 2^k which are intrinsically fast operations on digital computers. However, the Montgomery product algorithm requires the conversion of residues from mod n to mod 2^k, which are time-consuming operations. Thus, the Montgomery product computation algorithm is not useful when a single modular multiplication is to be performed. It is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute modular exponentiation. Using the binary or the m-ary methods for computing the powers, we replace the exponentiation operation by a series of square and multiplication operations modulo n.

In typical implementations, operations on large numbers are performed by breaking numbers into words. If w is the wordsize of the computer, then a number is thought of as a sequence of integers each represented in radix 2^w. If these multi-precision numbers require s words in radix W representation, then we take r as $r=2^{sw}$, where usually w=32.

A thorough study of several different algorithms for performing Montgomery product has been given in [KAK96]. We have included brief description of five algorithms in the Appendix. These algorithms are

- SOS: Separated Operand Scanning Method
- CIOS: Coarsely Integrated Operand Scanning Method
- FIOS: Finely Integrated Operand Scanning Method
- FIPS: Finely Integrated Product Scanning Method
- CIHS: Coarsely Integrated Hybrid Scanning Method

In the following, we describe the details of our implementations of these five algorithms on the MMX architecture.

5. MMX Implementation of Montgomery Multiplication Algorithms

5.1 Separated Operand Scanning (SOS) Method

The SOS algorithm exhibits a very regular structure. The first double-loop which performs the standard multiplication of two large numbers is simple, and consists of a major multiplication step:

(C,S) := t[i+j] + a[j]*b[i] + C

For an efficient implementation utilizing MMX instructions and registers, the word size should be carefully selected. In this selection, the instruction set and the algorithm should be considered. The MMX instruction set contains two multiplication instructions PMADD and PMUL(L/H). Both perform 16-bit *signed* multiplication. Due to this serious limitation only the lower half of the operands can be used. In the statement above we have a single word multiplication, therefore we prefer the PMUL(L/H) instruction. This instruction has two forms PMULLW and PMULHW. The PMULLW instruction multiplies the 16-bit fields of two MMX registers in parallel and writes the lower 16 bits of the result in the corresponding field of the destination register. The second form of the instruction does the same except that it writes the higher 16 bits of the result. Using the former we can use the MMX registers in full length. We are now ready to determine the word size. The easiest is to choose the word size to be 8 bits. Then in the inner loop 4 bytes can be multiplied in parallel. Before multiplying the operands must be prepared. The generation of the first operand mm0 is very easy and can be accomplished using two instructions:

```
movd mm0,a[j]       ; mm0 = | 0     0 | 0     0 | a[j+3] a[j+2] | a[j+1] a[j] |
punpcklbw mm0,mm7   ; mm0 = | 0 a[j+3]| 0 a[j+2]|     0  a[j+1] |    0   a[j] |
```

Here mm7 is zero. The second operand mm1 is not as easy to generate, and requires a broadcast operation.

```
movd mm1,b[i]        ; mm1 = | 0    0  | 0    0 | b[j+3] b[j+2] | b[j+1] b[j] |
punpcklbw mm1,mm1    ; mm1 = | 0    0  | 0    0 | b[j+2] b[j+2] |  b[j]  b[j] |
punpcklwd mm1,mm1    ; mm1 = | 0    0  | 0    0 |   b[j]   b[j] |  b[j]  b[j] |
punpcklbw mm1,mm7    ; mm1 = | 0  b[j] | 0  b[j]|      0   b[j] |     0   b[j] |
```

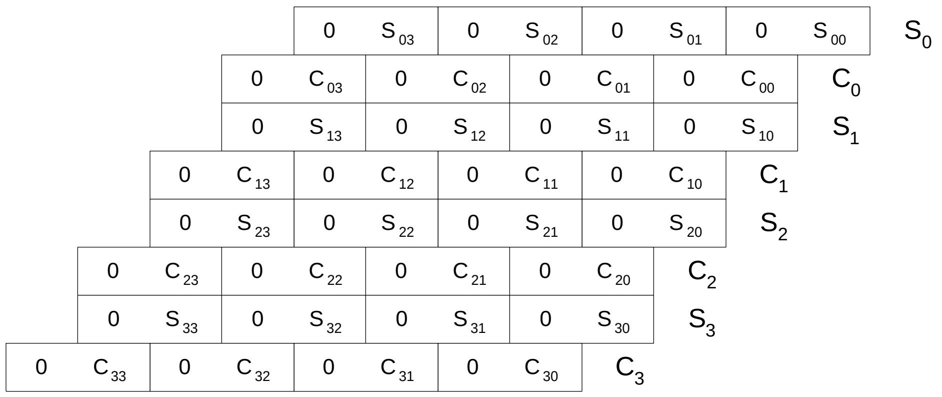
The multiplication operation yields the result as

```
pmullw mm0,mm1       ; mm0 = | a[j+3]*b[i] | a[j+2]*b[i] | a[j+1]*b[i] | a[j]*b[i]|
```

After completing the parallel multiplication, t[i+j] is expanded as it was done on a[j] above and added to mm0. Then we add the previous carry to the first 16 bits of mm0. The last step is to pack mm0. Note that we still have to propagate the carries by adding the higher bytes to the next lower bytes. This can be accomplished as follows:

```
movq mm1,mm0         ; mm1 = | C3  S3 | C2  S2 | C1 S1 | C0 S0 |
pand mm0,mm6         ; mm0 = |  0  S3 |  0  S2 |  0 S1 |  0 S0 |
psrlw mm1,8          ; mm1 = |  0  C3 |  0  C2 |  0 C1 |  0 C0 |
packuswb mm0,mm0     ; mm1 = |  0   0 |  0   0 | S3 S2 | S1 S0 |
packuswb mm1,mm1     ; mm1 = |  0   0 |  0   0 | C3 C2 | C1 C0 |
movd ebx,mm0         ; ebx = S3 S2 S1 S0
movd edx,mm1         ; edx = C3C2C1C0
movd ecx,edx
shr ecx,24           ; ecx  =    0  0  0 C3
shl edx,8            ; edx = C2 C1 C0   0
add ebx,edx          ; ebx  = S
adc ecx,0            ; ecx  = C
```

In this method, the preparation of the operands and the final part (where the carry propagation and packing are performed) are time-consuming and not sufficiently parallel. One way to reduce both is to select the word size to 32 bits, and extend the above algorithm so that in each iteration four multiplications are performed. This will increase the number of independent instructions and will allow us to propagate the carries in a collective fashion. In the new algorithm the addition of the previous carry and t[i+j] is left to the very end. This will give enough space to add the carries as they are generated. The figure below illustrates this step:


0 S_{03} 0 S_{02} 0 S_{01} 0 S_{00} S_0
0 C_{03} 0 C_{02} 0 C_{01} 0 C_{00} C_0
0 S_{13} 0 S_{12} 0 S_{11} 0 S_{10} S_1
0 C_{13} 0 C_{12} 0 C_{11} 0 C_{10} C_1
0 S_{23} 0 S_{22} 0 S_{21} 0 S_{20} S_2
0 C_{23} 0 C_{22} 0 C_{21} 0 C_{20} C_2
0 S_{33} 0 S_{32} 0 S_{31} 0 S_{30} S_3
0 C_{33} 0 C_{32} 0 C_{31} 0 C_{30} C_3

M_i denotes the i^th product. C and S_i denote the unpacked carry and sum of M_i, respectively. The sum of the first product S_0 is directly packed and added to the product. On the other hand the carry C_0 can be added to M_1 since it has the same order as S_1. Now S_1 can be packed, shifted and added to the product. The remaining part of M_1 which is C_1 is added to M_2. This process continues until the whole product is formed. Afterwards t[i+j] and the previous carry are added to the product. The higher word of the product will give the next carry and the lower word will give the sum.

5.2 Finely Integrated Product Scanning (FIPS) Method

The FIPS algorithm has a complex structure. It consists of two parts which have similar statements. The following code sequence appears in the inner loop of both parts.

```
(C,S) := t[0] + a[j]*b[i-j]
ADD(t[1],C)
(C,S) := S + m[j]*n[i-j]
```

Although the products can be calculated using the algorithm presented in SOS separately, this method fails to exploit the parallel nature of the code sequence. Thus, a new algorithm needs to be developed. We rewrite the sequence to a more parallel form as follows.

```
(C,S) := t[0] + a[j]·b[i-j] + m[j]·n[i-j]
ADD(t[1],C)
```

Note that C is now 2 words long. Now we will concentrate on the computation of the products in parallel. In the computation of a∗b + m∗n, the instruction PMADD will be used. Since PMADD performs signed 16-bit multiplication only the lower 8-bits are utilized. Effectively a single PMADD instruction will perform four 8-bit multiplications and two 16-bit additions. The operands are generated via the broadcast and unpack operations as done in the SOS algorithm. The following illustrates the multiplication. Using these constructs, a straightforward implementation with 8-bits word size and two iterations per multiplication is possible.

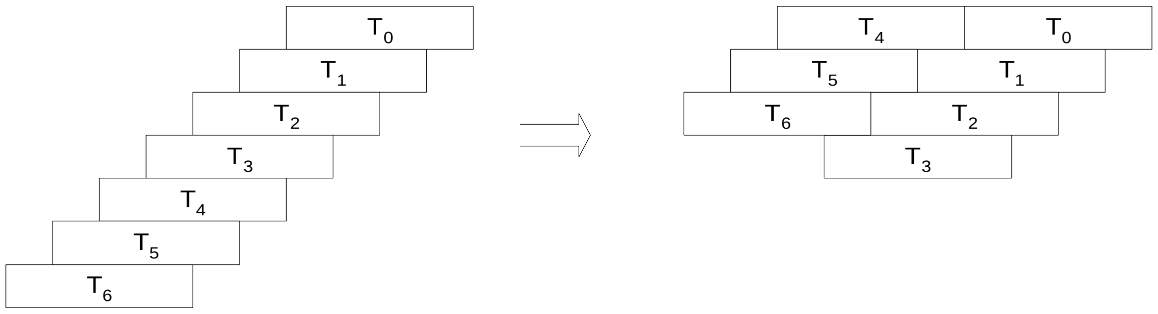
```
mm0 =  |  0   m[j+1]  |  0  a[j+1]  |  0  m[j]  |  0  a[j]  |
mm1 =  |  0     n[i]  |  0    b[i]  |  0  n[i]  |  0  b[i]  |

pmadd  mm0,mm1      ; |  0  m[j+1]*n[i]+a[j+1]b[i] | m[j]n[i]+a[j]b[i] |
```

However only the first bit of the higher 16-bit word is used. If the word size is selected as 32 bits, this space can be used to accumulate the carries. This brings us to an algorithm similar to the one used in SOS. The partial products are shown in the graph below. Each box corresponds to a 32-bit product and t is defined as t_{ij} = m[j]·n[i]+a[j]·b[i]. Each line is generated using two PMADD instructions and corresponds to two MMX registers. The boxes in one column have the same order and can be added in a single MMX register using the PADDD instruction. The resulting structure will have seven 32-bit registers which have to be packed and added properly to form the 63-bit product.

			t_{03}	t_{02}	t_{01}	t_{00}
		t_{13}	t_{12}	t_{11}	t_{10}	
	t_{23}	t_{22}	t_{21}	t_{20}		
t_{33}	t_{32}	t_{31}	t_{30}			
T_6	T_5	T_4	T_3	T_2	T_1	T_0

The final step performs the carry propagation. Since the order of t_{ij} is one less than the order of t_{ij+1} the generated partial products must be aligned byte-wise. The resulting structure is shown below on the left. This computation of this structure involves many shifts and additions. We can simplify this to the structure shown below on the right.



The MMX instruction set lacks a 64-bit addition instruction, thus, in order to accumulate these partial products we need to switch back to the integer registers and perform consecutive shift and add operations. The product can be accumulated in three 32-bit registers. To finish the statement t[0] is added to the product.

5.3 Coarsely Integrated Hybrid Scanning (CIHS) Method

The CIHS algorithm is a modified version of the SOS algorithm. The indexing method is different. But the single word multiplication statement in the inner loop remains the same. Therefore the very same method presented in SOS is used in the implementation. The speed of CIHS is less than the speed of SOS mostly due to the odd indexing behavior which is not suitable for the MMX architecture.

5.4 Coarsely Integrated Operand Scanning (CIOS) Method

The CIOS algorithm contains two inner loops. We used this algorithm in the ICL (Intel Cryptographic Library). This method is implemented in Pentium assembler and highly optimized for that processor. Therefore, it is not fair to compare the currently available MMX functions to this Pentium-CIOS function unless the MMX codes become more optimized.

CIOS algorithm on MMX utilizes PMULLW, PMULLH and UNPCKL instructions. Multiplications are carried out using 8-bit wordsize. Four multiplications are done in each inner loop. Theoretically, this should yield a similar timing with the non-MMX version of the same algorithm. But, lack of 64-bit addition instruction in the MMX instruction set degraded the performance. The MMX-CIOS algorithm is composed of six parts: *load, multiply, unpack, mask/shift*, and *add* parts. The following pseudocode illustrates these operations:

```
•   add
        mm4:=(a[j+3])(a[j+2])(a[j+1)(a[j])
        mm0:=(b[i])(b[i])(b[i])(b[i])
        mm1:=(b[i+1])(b[i+1])(b[i+1])(b[i+1])
        mm2:=(b[i+2])(b[i+2])(b[i+2])(b[i+2])
        mm3:=(b[i+3])(b[i+3])(b[i+3])(b[i+3]
•   multiply
        mm0:=mm0*mm4
        mm1:=mm1*mm4
        mm2:=mm2*mm4
        mm3:=mm3*mm4
•   unpack word to dword
        mm4:=mm0
        mm5:=mm1
        UnpackH mm0,mm2       ;I0
        UnpackL mm1,mm3       ;I1
        UnpackL mm4,mm2       ;I2
        UnpackH mm5,mm3       ;I3
•   even -  mask/shift
        mm2:=mm0 AND Mask0_31
        mm3:=mm1 AND Mask32_63
        mm6:=mm4 AND Mask0_31
        mm7:=mm5 AND Mask32_63
        mm2:=mm2 SRL 16
        mm3:=mm3 SRL 16
•   odd -  mask/shift
        mm0:=mm0 AND Mask32_63
        mm1:=mm1 AND Mask0_31
        mm4:=mm4 AND Mask32_63
        mm5:=mm5 AND Mask0_31
        mm7:=mm5 AND Mask32_63
        mm0:=mm0 SRL 8
        mm1:=mm1 SLL 8
        mm4:=mm4 SRL 24
        mm5:=mm5 SLL 24
•   add
        mm2:=mm2 + mm3
        mm6:=mm6 + mm7
        mm0:=mm0 + mm1
        mm4:=mm4 + mm5
        mm2:=mm2 + mm6
        mm0:=mm0 + mm4
        mm0:=mm0 + mm2
•   add - t[j] and C
        (C,eax):=eax + t[j]
        ebx:=mm0[31..0]
        mm0:=mm0 SRL 32
        edx:=mm0[31..0] + C
        (C,eax):=eax +ebx
        t[j]:=eax
        eax:=edx + C
```

We used multiplication, addition and memory access (read/write) instructions of MMX technology to implement the Montgomery multiplication algorithms. The MMX instructions utilized in the innermost loop of the current CIOS implementation are listed below:

MMX Instruction	Number of Uses
movd	1
movq	18
punpcklbw	1
pmullw	4
pand	8
psllq	3
psrlq	3
punpckhwd	2
punpcklwd	2

5.5 Finely Integrated Operand Scanning (FIOS) Method

The two inner loops in FIOS can be combined into a single inner loop with two carries as:

```
for j=0 to s-1
        $(Cx,C,S) := t[j] + a[j]\cdot b[i] + m*n[j] + C + Cx\cdot 2^w$
        $t[j-1] := S$
```

It is verified that the result always fits in three words:

$$(Cx\ C,\ S) \leq 2^w-1 + (2^w-1)(2^w-1) + (2^w-1)(2^w-1) + 2^w-1 + 2^w(2^w-1) < 4 \cdot 2^{2w} - 4 \cdot 2^w < 2^{2w+2}\ .$$

Assuming w>2, the result fits in three registers. The regular code deals with the double carry by the following pseudocode:

```
$(C1,S1) := a[j] \cdot b[i] + t[j]$
$(C2,S2) := m + n[j] + C$
$(C1,C) := (C1,S1) + S2$
$t[j-1] := C$
$(Cx,C) := C1 + C2 + Cx$
```

The wide registers and the simultaneous multiply-and-add instruction of MMX allow us to formulate these operations as follows:

a) $(C1,S1) := a[j]*b[i] + m\cdot n[j]$
b) $(C2,S2) := (Cx,C) + t[j]$
c) $(C,S) := (C1,S1) + (C2,S2)$

where each a[j], b[i], t[j], m, n[j] are 8 bits, zero extended to 16 bits. All other variables are 16 bits. One instruction contains two of such operations (PMADDWD and PADDD). This offers various scheduling possibilities with eight 64-bit registers, The simultaneous multiplication capability of MMX allows a pair of addition of two 16-bit integer multiplications in a single instruction (PMADDWD). However, the multiplication uses signed arithmetic leading to 8-bit wordsize selection for Montgomery multiplication. The MMX pseudocode of the FIOS algorithm is illustrated below.

```
for i=0 to s-1 do
(C,S) := t[0] + a[0]*b[i]
```

```
m := S*n'[0] mod W
(Cx,C,S) := (C,S) + m*n[0]
for j=1 to s-1 do
   (C1,S1) := a[j]*b[i] + m*n[j]
   (C1',S1') := a[j+1]*b[i] + m*n[j+1]
   (C2,S2) := t[j] + (Cx,C)
   (C,S) := (C1,S1) + (C2,S2)
   (C',S') := t[j+1] + (C1',S1')
   t[j-1] := S
   (C,S) := (C,S) >> 8
   (C,S) := (C',S') + (C,S)
   t[j] := S
   (Cx,C) := (C,S) >> 8
(Cx,C,S) := a[s-1]×b[i]+m×n[s-1]+(Cx,C)+(t[s],t[s-1])
t[s-2] := S
t[s-1] := C
t[s] := Cx
```

The last Cx and C are the new carries which are used in the next iteration. The wordsize is 8 bits. The data structure is as follows:

- Each 8 bit is represented in a 16 bit space, higher 8 bits zero extended.
- Each byte of t must be stored in a 32-bit space, zero extended.
- (a[j],b[i]) pairs must be prepared in this order in memory.
- Each a[j] and b[i] in 16 bits, making the total length of each pair 32 bit.
- (b[i],m) pairs have the similar structure with (a[j],b[i]) pairs.
- m is updated in each outer loop which is carried out by regular Pentium instructions.

MMX Instruction	Number of Uses
movd	4
movq	1
pmaddwd	1
paddd	1
por	1
psllq	1
psrlq	1

We have implemented the FIOS algorithm for Montgomery multiplication on Pentium Pro/MMX in assembly language. Any (C,S) pair or (Cx,C,S) triple is stored in one MMX register.

6. Timing Results

We have implemented and tested these five Montgomery multiplication methods as the core routine of the RSA and DSS algorithms on a 233 MHz Pentium Pro/MMX machine. The following table gives the timings of the RSA Public & Private and DSS Sign & Verify operations in milliseconds. The size of signature operands is 1024 bits. These timing values are average values obtained by running the RSA and DSS operations 32 times.

We have developed C++ and MMX Assembler implementations of all five algorithms. As was mentioned, we had also previously developed a highly optimized Pentium Assembler implementation of the CIOS algorithm as part of the work for developing Intel Cryptographic Library. Because the MS Visual C++ compiler generates Pentium code, we are able to compare the timings of the C code running only on the Pentium Pro to the timings of MMX Assembler code (which also contains C and Pentium Assembler parts) running on Pentium Pro and MMX.

Furthermore, the timings of the optimized CIOS code written in the Pentium Assembler provides a comparison of C versus Assembler code running on the Pentium Pro. However, the comparison of the optimized CIOS algorithm on the Pentium Pro to all five algorithms on the MMX is not a fair one since we had much more time to reformulate and program the CIOS algorithm for the Pentium Processor.

		RSA Public	RSA Private	DSS Sign	DSS Verify
MS VC++ 4.2	**CIOS**	18	121	79	154
	SOS	20	133	88	167
	FIOS	20	117	78	151
	FIPS	17	113	76	145
	CIHS	20	127	84	159
MMX Assembler	**CIOS**	32	291	137	260
	SOS	22	180	121	230
	FIOS	25	194	146	274
	FIPS	21	171	113	214
	CIHS	25	214	140	267
Pentium Assembler	**CIOS**	10	60	45	84

As can be seen from this table, the MMX assembler code is approximately 1.5 times slower than the C code compiled by Microsoft Visual C++ 4.2 and executed on the Pentium Pro. Furthermore, the MMX assembler code is about 3 times slower than the optimized Pentium assembler code executed on the Pentium Pro.

It turns out that the fastest RSA & DSS Sign/Verify operations on the MMX are obtained when the Montgomery multiplication algorithm is selected to be either the FIPS or the SOS method. However, the fastest Montgomery algorithm on the Pentium Processor was the CIOS method.

It is our observation that the lack of unsigned arithmetic, 32-bit multiplication and 64-bit addition instructions are the main reasons behind the loss of performance on the MMX platform. We summarize our findings and conclusions in the last section.

7. Crucial Operations for Number-Theoretic Cryptography

In the following, we describe a set of arithmetic and data movement operations which are crucial in obtaining high-speed implementations of the Montgomery product algorithms on the MMX architecture. We also give MMX code segments implementing these operations in order to observe and clarify the use of MMX architectural features in obtaining the highest possible performance.

7.1 Unsigned Multiply Operation

The core of the Montgomery multiplication algorithm is the multiply-add operation. The following statement appears in all five algorithms:

$$(C,S) := t[i+j] + a[i]*b[j] + C$$

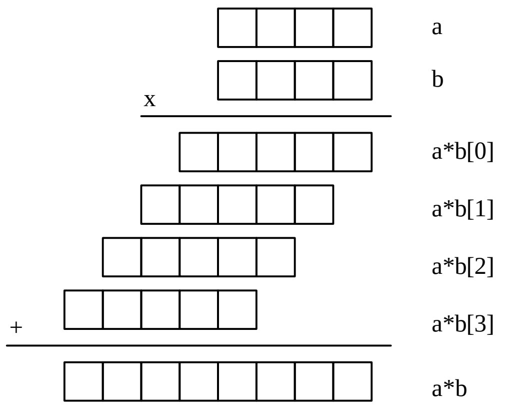
where t, a, b, c are 8-bit or 16-bit or 32-bit unsigned integers. This operation needs to be performed using unsigned arithmetic. Currently the MMX technology provides only 16-bit signed multiplication which is used to emulate 8-bit unsigned multiplication. The instruction PMULLW is used to obtain the 16-bit product where the high 8-bits of the input operands are set to zero.

```
Bits:      63—-48  47--32  31--16  15---0
Input1:     0 a3    0 a2    0 a1    0 a0
Input2:     0 b3    0 b2    0 b1    0 b0
Output:    a3*b3   a2*b2   a1*b1   a0*b0        [PMULLW]
```

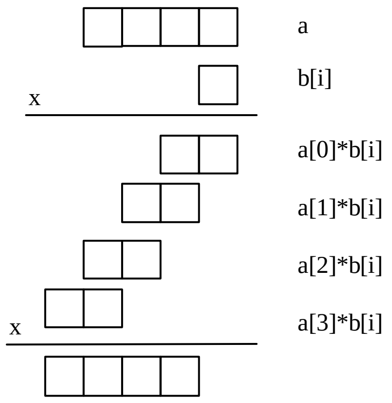
In the following we give the detailed MMX code implementing the multiplication operation in which two 32-bit unsigned integers are multiplied to obtain a 64-bit unsigned integer:

(C,S) := a ∗ b

Let these numbers be represented as a=(a[3] a[2] a[1] a[0]) where a[3] and a[0] are the most and least significant 8-bits of 32-bit number a. The multiplication of a*b is visualized as follows:



The MMX instruction set limits us to 8-bit by 8-bit unsigned multiplication only. We obtain each partial product (a∗b[i]) in the above figure using a PMULLW instruction as follows:



Below we summarize the MMX code for obtaining the product (C,S) using the PMULLW instruction.

-
  ```
  add
      mm4:=(a[3])(a[2])(a[1])(a[0])
      mm0:=(b[0])(b[0])(b[0])(b[0])
      mm1:=(b[1])(b[1])(b[1])(b[1])
      mm2:=(b[2])(b[2])(b[2])(b[2])
      mm3:=(b[3])(b[3])(b[3])(b[3])
  ```

-
  ```
  multiply
      mm0 := mm0*mm4
      mm1 := mm1*mm4
      mm2 := mm2*mm4
      mm3 := mm3*mm4
  ```

-
  ```
  even - mask/shift
      mm2 := mm0 AND Mask0_31
      mm3 := mm1 AND Mask32_63
      mm6 := mm4 AND Mask0_31
      mm7 := mm5 AND Mask32_63
      mm2 := mm2 SRL 16
  ```

```
        mm3 := mm3 SRL 16
```

- odd - mask/shift
```
        mm0 := mm0 AND Mask32_63
        mm1 := mm1 AND Mask0_31
        mm4 := mm4 AND Mask32_63
        mm5 := mm5 AND Mask0_31
        mm0 := mm0 SRL 8
        mm1 := mm1 SLL 8
        mm4 := mm4 SRL 24
        mm5 := mm5 SLL 24
```

- add
```
        mm2 := mm2 + mm3
        mm6 := mm6 + mm7
        mm0 := mm0 + mm1
        mm4 := mm4 + mm5
        mm2 := mm2 + mm6
        mm0 := mm0 + mm4
        mm0 := mm0 + mm2
```

Note that the additions in the final add block are accomplished using Pentium instructions. This is due to the fact that the MMX does not provide an addition with carry instruction to obtain the sum of a vector.

The availability of a 16-bit or 32-bit unsigned multiplication instruction would considerably speedup the above code segment. In order to overcome these obstacles, we propose two unsigned multiplication instructions: PMULUWD and PMULUDQ.

The proposed 16-bit unsigned multiplication instruction PMULUWD multiplies x=a*b and y=c*d as follows:

Instruction: PMULUWD

Input1: 0(15..0) c(15..0) 0(15..0) a(15..0)

Input2: 0(15..0) d(15..0) 0(15..0) b(15..0)

Output: y(31..0) x(31..0)

The code segment using this new instruction to obtain the product (C,S) is given as:

```
pxor      mm7,mm7
movd      mm0,a
movd      mm1,b
movd      mm2,mm1
psrl      mm2,16     ; mm2 = |   0    |    0    |    0    |    b[1]   |
punpcklwd mm0,mm7    ; mm0 = |   0    |  a[1]   |    0    |    a[0]   |
punpcklwd mm1,mm1    ; mm1 = |   x    |    x    |  b[0]   |    b[0]   |
punpcklwd mm2,mm2    ; mm2 = |   x    |    x    |  b[1]   |    b[1]   |
punpcklbw mm1,mm7    ; mm0 = |   0    |  b[0]   |    0    |    b[0]   |
punpcklwd mm2,mm7    ; mm0 = |   0    |  b[1]   |    0    |    b[1]   |
pmuluwd   mm1,mm0    ; mm1 = |     a[1]*b[0]    |      a[0]*b[0]      | ; new
pmuluwd   mm2,mm0    ; mm1 = |     a[1]*b[1]    |      a[0]*b[1]      | ; new


movd      ebx,mm1
psrl      mm1,32
movd      edi,mm2
psrl      mm2,32
movd      esi,mm1
movd      ecx,mm2
add       ebx,esi
adc       edi,ecx
// Product is now in   | edi | ebx |
```

The proposed 32-bit unsigned multiply instruction PMULUDQ computes x=a*b as

Instruction: PMULUDQ

Input1: 0(31..0) a(31..0)

Input2: 0(31..0) b(31..0)

Output: x(63..0)

Thus, we can use this instruction compute (C,S)=a∗b in 3 lines:

```
movd mm1,a
movd mm2,b
pmuludq mm1,mm2      ; New instruction
```

7.2 Unsigned Multiply-Add Operation

The multiply-add operation takes four 1-word inputs a, b, c, d and produces two 1-word outputs C, S and the carry bit Cx as follows:

(Cx,C,S) := a∗b + c∗d

The MMX instruction PMADDWD takes 16-bit signed inputs a, b, c, d and produces the signed 32-bit output, where the carry bit is truncated. This instruction is used in two different operations:

(C,S) := a∗b + c

(C,S) := a∗b + c∗d

where a, b, c are 8-bit values, and thus there is no Cx. The second operation is more general, from which the first operation can be obtained by taking d=1. In order to compute the second result:

(C,S) := a∗b + c∗d

we take 8-bit inputs a and c, extend them to 16-bits by padding zeros, and place them in the lower 32 bits of a 64-bit MMX register. Similarly, b and d are placed in the lower 32-bits of another MMX register. The higher 32-bits of these registers will contain another 4 such numbers. After the execution of PMADDWD, we obtain the 17-bit result (carry,C,S) in the lower 32-bit of the MMX register.

We propose the instruction PMADDUWD which is the unsigned version of PMADDWD as illustrated below:

Instruction: PMADDUWD

Input1: g(15..0) e(15..0) c(15..0) a(15..0)

Input2: h(15..0) f(15..0) d(15..0) b(15..0)

Output: y(31..0) x(31..0)

where x=a*b+c*d mod 2^{32} and y=e*f+g*h mod 2^{32}, the carry-out bits are being ignored. If the instruction PMADDUWD is available, then we can perform two simultaneous operations of type (C,S) := a∗b + c by taking d=1, where a, b, c, C, and S are 16-bit unsigned integers.

Furthermore, if the unsigned 32-bit version PMADDUDQ of the instruction PMADDWD is available, then we can proceed 32 bits at a time:

Instruction: PMADDUDQ

Input1: c(31..0) a(31..0)

Input2: d(31..0) b(31..0)

Output: x(63..0)

where x=a*b+c*d mod 2^{64}, where the carry-out bit is being provided.

7.3 64-Bit Addition Operation

In computing the statement (C,S) := t[i+j] + a[i]*b[j] + C to accumulate the partial results, it is necessary to switch to 32-bit integer registers since the carry at the 32-bit boundary cannot be detected. In order to obtain the result, lengthy shift and move instructions are used since the access to the higher words of the MMX registers is limited. This process usually takes longer than the multiplication itself. Another important disadvantage is that the integer registers are occupied, which should normally be used for array indexing only. This can be eliminated by adding a 64-bit addition instruction PADDQ to the MMX instruction set:

Instruction: PADDQ

Input1: a(63..0)

Input2: b(63..0)

Output: x(63..0)

where x=a+b, and the carry-out bit is being ignored. The add-with-carry version of this

instruction (PADCQ) is also being proposed:

Instruction: PADCQ

Input1: a(63..0)

Input2: b(63..0)

Input3: Carry-In

Output: x(63..0), Carry-Out

We used a four-operand addition S := a + b + c + d in the CIOS method, where a, b, c, d and S are 64 bits each. This is a full 64 bit addition without carry-in and carry-out. Also, a, b, c and d contain 16-bit numbers which are not necessarily aligned at 16-bit boundaries. An actual situation appears in CIOS as given as follows. Assume that the MMX registers contain

```
mm0:   I5 I3 I4 I2
mm1:   I4 I2 I3 I1
mm4:   I3 I1 I2 I0
mm5:   I6 I4 I5 I3
```

where each `Ii` is 16 bits. We want to add seven 16-bit shifted Ki values as shown below:

```
      K0
     K1
    K2
   K3
  K4
 K5
K6
```

```
---------------------
S3,S2,S1,S0
```

where S3, S2, S1 and S0 contains the 64-bit sum, and

```
K0:=mm4[I0]
K1:=mm1[I1]+mm4[I1]
K2:=mm0[I2]+mm1[I2]+mm4[I2]
K3:=mm0[I3]+mm1[I3]+mm4[I3]+mm5[I3]
K4:=mm0[I4]+mm1[I4]+mm5[I4]
K5:=mm0[I5]+mm5[I5]
K6:=mm5[I6]
```

Currently, this addition is performed using both MMX and Pentium registers. Four MMX registers are shifted to align the 16-bit numbers at 16-bit boundaries. This requires use of eight MMX registers. Then these aligned registers are dumped to 64-bit memory locations. Final addition of these distributed 16-bit numbers are done using Pentium registers. Assume the MMX registers mm0, mm1, mm4 and mm5 contain numbers as shown above, and we want to produce the final result in mm0 register. We use the following pseudocode to achieve this operation:

```
• even - mask/shift
      mm2 := mm0 AND Mask0_31
      mm3 := mm1 AND Mask32_63
      mm6 := mm4 AND Mask0_31
      mm7 := mm5 AND Mask32_63
      mm2 := mm2 SRL 16
      mm3 := mm3 SRL 16
• odd - mask/shift
      mm0 := mm0 AND Mask32_63
      mm1 := mm1 AND Mask0_31
      mm4 := mm4 AND Mask32_63
      mm5 := mm5 AND Mask0_31
      mm0 := mm0 SRL 8
      mm1 := mm1 SLL 8
      mm4 := mm4 SRL 24
      mm5 := mm5 SLL 24
• add
      mm2 := mm2 + mm3
      mm6 := mm6 + mm7
      mm0 := mm0 + mm1
      mm4 := mm4 + mm5
      mm2 := mm2 + mm6
      mm0 := mm0 + mm4
      mm0 := mm0 + mm2
```

Note that, the final addition block uses 64-bit memory locations for eight MMX registers, and addition is performed using Pentium registers and these memory locations. We currently use the following actual code to perform the last block of additions:

```
; add 16-bit words
    movq    qword ptr [rmm0],mm0
    movq    qword ptr [rmm1],mm1
    movq    qword ptr [rmm2],mm2
    movq    qword ptr [rmm3],mm3
    movq    qword ptr [rmm4],mm4
    movq    qword ptr [rmm5],mm5
    movq    qword ptr [rmm6],mm6
    movq    qword ptr [rmm7],mm7
    add     eax,dword ptr [rmm0]
    adc     edx,dword ptr [rmm0+4]
    add     eax,dword ptr [rmm1]
    adc     edx,dword ptr [rmm1+4]
    add     eax,dword ptr [rmm2]
    adc     edx,dword ptr [rmm2+4]
    add     eax,dword ptr [rmm3]
    adc     edx,dword ptr [rmm3+4]
    add     eax,dword ptr [rmm4]
    adc     edx,dword ptr [rmm4+4]
    add     eax,dword ptr [rmm5]
    adc     edx,dword ptr [rmm5+4]
    add     eax,dword ptr [rmm6]
```

```
        adc     edx,dword ptr [rmm6+4]
        add     eax,dword ptr [rmm7]
        adc     edx,dword ptr [rmm7+4]
```

However, we have not observed any performance gain when the above instruction sequence is replaced with shift/mask/add MMX instructions instead of eight 64-bit memory locations. If the PADDQ instruction were available (64-bit addition), it would add eight MMX registers in the last addition block above. Furthermore, it would be very useful if this 64-bit addition can be performed using an input carry, and generating an output carry.

7.4 64-bit Unsigned Multiply Operation

This is a monster operation taking two 64-bit unsigned integers giving the 128-bit product (C,S) := a*b where each a, b, S and C is 64-bits wide.

Input1: a(63..0)

Input2: b(63..0)

Output: x(127..0)

We currently use PMULLW instruction to multiply four 8-bit words in all of five algorithms. Compacting four 16-bit partial products is achieved by different methods based on shift/add and move instructions. If registers mm0 and mm1 contain two 64-bit values a and b respectively, the PMULUQ instruction would multiply two registers and produce a 128-bit product as:

```
movd mm0,a
movd mm1,b
pmuluq mm1,mm0     ;  mm1,mm0 (a*b)
```

7.5 Broadcast Operation

The broadcast operation is used in the current MMX implementation of all five Montgomery multiplication algorithms. It takes a single byte value b[0], zero extends it to 16 bits, and creates 4 copies of it in a 64-bit register We use the following MMX code to realize this operation.

```
movd mm1,b           ; mm1 = |  0   0  |  0   0  | b[3] b[2] | b[1] b[0] |
punpcklbw mm1,mm1    ; mm1 = |  0   0  |  0   0  | b[2] b[2] | b[0] b[0] |
punpcklwd mm1,mm1    ; mm1 = |  0   0  |  0   0  | b[0] b[0] | b[0] b[0] |
punpcklbw mm1,mm7    ; mm1 = |  0  b[0]|  0  b[0]|  0   b[0] |  0   b[0] |
```

This code segment is used four times in a single 32-bit multiplication. If this operation is implemented as a single instruction PBROAD, then we can use this instruction as

```
pbroad mm1,b
```

to replace four lines. We currently utilize this instruction to broadcast a single byte.

We summarize the proposed 8 instructions below:

PMULUWD	**Input**: Four 16-bit unsigned integers a, b, c, d
	Input1: 0 c 0 a
	Input2: 0 d 0 b
	Output: Two 32-bit unsigned integers x, y

Output: y x

where x=a*b, y=c*d

PMULUDQ

Input: Two 32-bit unsigned integers a, b

Input1: 0 a

Input2: 0 b

Output: One 64-bit unsigned integer x

Output: x

where x=a*b

PMADDUWD

Input: Eight 16-bit unsigned integers a, b, c, d, x, y, z, w

Input1: g e c a

Input2: h f d b

Output: Two 32-bit unsigned integers x, y

Output: y x

where x=a*b+c*d, y=e*f+g*h mod 2^{32}

The CarryOut bits are ignored.

PMADDUDQ

Input: Four 32-bit unsigned integers a, b, c, d

Input1: c a

Input2: d b

Output: One 64-bit unsigned integer x

Output: (CarryOut, x)

where x=a*b+c*d mod 2^{64}

The CarryOut bit is being provided.

PADDQ

Input: Two 64-bit unsigned integers a, b

Input1: a

Input2: b

Output: One 64-bit unsigned integer x

Output: x

where x=a+b mod 2^{64}

The CarryOut bit is being ignored.

PADCQ

Input: Two 64-bit unsigned integers a, b and the CarryIn bit

Input1: a

Input2: b

Output: One 64-bit unsigned integer x and the CarryOut bit

Output: (CarryOut, x)

where (CarryOut,x) = a+b+CarryIn

PMULUQ	**Input**: Two 64-bit unsigned integers a, b
	Input1: a
	Input2: b
	Output: Two 64-bit unsigned integers x, y
	Output: y, x
	where (y,x)=a*b
PBROAD	**Input**: One 8-bit integer a
	Input: 0 0 0 0 0 0 0 a
	Output: One 64-bit integer x
	Output: x
	where x = 0 a 0 a 0 a 0 a

8. Performance Estimates with Proposed Instructions

In this section, we present the performance estimates for the RSA and DSS algorithms assuming a subset of the proposed instructions were made available. This is performed by writing a `pretend code' which performs some computation using certain existing instructions which are comparable in difficulty and timing to the proposed instructions. The resulting arithmetic values are not correct, but the timings are close to the expected timings if these proposed instructions existed on the MMX platform.

We consider the following 4 cases:

- FIPS_1: The FIPS algorithm with instruction PMULUWD
- FIPS_2: The FIPS algorithm with instructions PMADDUDQ, PADDQ, and PADCQ
- CIOS_1: The CIOS algorithm with instruction PADDQ
- CIOS_2: The CIOS algorithm with instructions PMULUDQ and PADDQ

In place of these proposed instructions, we have placed the following existing MMX instructions in the `pretend' code of the FIPS and CIOS algorithms:

Proposed Instruction	**Existing Instruction**
PADDQ	PADDD
PADCQ	PADDD
PMULUWD	PMULLW
PMULUDQ	PMULLW
PMADDUDQ	PMADDWD

The timing results (in milliseconds) of the `pretend' code are tabulated in the first 4 rows of the following table. For comparison, we also give the timings of our fastest MMX implementation, and the timings of the highly optimized Pentium Assembler code running on the Pentium Pro in the last 2 rows.

	RSA Public	RSA Private	DSS Sign	DSS Verify
FIPS_1	21	149	100	180
FIPS_2	5	27	23	34
CIOS_1	18	132	89	181
CIOS_2	5	39	30	48
Fastest MMX	21	171	113	214
Pentium ASM	10	60	45	84

Note that the CIOS_2 implementation of the RSA Private computation with 1024-bit modulus corresponds to a data rate of 25 Kbits/sec. The fastest RSA hardware implementation ever reported was 185 Kbits/sec for 970-bit modulus [RSA97]. We also tabulate the 1024-bit Montgomery multiplication timings below for comparison:

FIPS on MMX	440 usec	FIPS_1 on MMX	384 usec
FIOS on MMX	510 usec	FIPS_2 on MMX	40 usec
SOS on MMX	472 usec	CIOS_1 on MMX	324 usec
CIOS on MMX	560 usec	CIOS_2 on MMX	82 usec
CIHS on MMX	549 usec	CIOS on PRO	143 usec

9. REFERENCES

[BG89] T. Beth and D. Gollmann. Algorithm engineering for public-key algorithms. *IEEE Jour. on Selected Areas in Comm.*, 7(4):458-466, 1989.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644-654, 1976.

[DK90] S. R. Dusse and B. S. Kaliski, Jr. A cryptographic library for the Motorola DSP56000. *EUROCRYPT 90*, pages 230--244. Springer, 1990.

[E85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. . *IEEE Transactions on Information Theory*, 31(4):469-472, 1985.

[N91] National Institute for Standards and Technology. Digital signature standard (DSS). Federal Register, 56:169, August 1991.

[K87] N. Koblitz. *A Course in Number Theory and Cryptography*. Springer, 1987

[K94] C. K. Koc. High-Speed RSA Implementation. Technical Report, RSA Laboratories, RSA Data Security, Inc., November 1994.

[KAK96] C. K. Koc, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication. *IEEE Micro*, 16(3):26-33, June 1996.

[K96] C. K. Koc. Intel Cryptographic Library (ICL). Component of Common Data Security Architecture (CDSA) Software Development Kit.

[M93] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer, 1993.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. of the ACM*, 21(2):120-126, 1978.

[RSA97] RSA Laboratories. *Answers to Frequently Asked Questions about Today's Cryptography*. Version 3.0, RSA Data Security, Inc., 1997

10. APPENDIX. Montgomery Multiplication Algorithms

The Montgomery algorithm computes MonPro(a,b) = $a * b * r^{-1} \bmod n$.

```
function MonPro (a, b)
Step 1. t := a· b
Step 2. u := (t + (t * n’ mod r) * n) / r
Step 3. If u ≤ n then return u-n else return u
```

given a,b<n and r such that gcd(n,r)=1. The modulus n is a k-bit integer, i.e., $2^{k-1} \leq n < 2^k$, and $r = 2^k$. Multiplication modulo r and division by r are both intrinsically fast operations, since r is a power of 2. The Montgomery product algorithm is potentially faster than ordinary computation of a∗b mod n, which involves division by n. However, since conversion from an ordinary residue to an n-residue, computation of n’, and conversion back to an ordinary residue are time-consuming, it is not a good idea to use the Montgomery product computation algorithm when a single modular multiplication is to be performed. It is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute modular exponentiation. Using the binary method for computing the powers, we replace the exponentiation operation by a series of square and multiplication operations modulo n.

In typical implementations, operations on large numbers are performed by breaking numbers into words. If w is the wordsize of the computer, then a number is thought of as a sequence of integers each represented in radix $W=2^w$. If these *multi-precision* numbers require s words in radix W representation, then we take r as $r=2^{sw}$. We take w=32 for most platforms.

A thorough description of several different algorithms for performing the Montgomery product is given in [KAK96]. Here, we give brief descriptions and pseudocodes of five algorithms. We may roughly organize the algorithms based on two factors. The first factor is whether multiplication and reduction are *separated* or *integrated*. In the separated approach, we first multiply a and b, then perform a Montgomery reduction. In the integrated approach, we alternate between multiplication and reduction. This integration can be either *coarse-grained* or *fine-grained*, depending on how often we switch between multiplication and reduction (specifically, after processing an array of words, or just one word).

The second factor is the general form of the multiplication and reduction steps. One form is the *operand scanning*, where an outer loop moves through words of one of the operands; another form is *product scanning*, where the loop moves through words of the product itself. It is also possible for multiplication to have one form and reduction to have the other form, even in the integrated approach.

Separated Operand Scanning (SOS) Method

The first method described for computing MonPro(a,b) is what we call the Separated Operand Scanning method. In this method we first compute the product t = a*b. The final value obtained is the 2s-word integer t residing in words t[0]…t[2s-1]. Then we compute u using the formula u=(t+m*n)/r, where m=t*n’ mod r. In order to compute u, we first take u=t, and then add m*n to it using the standard multiplication routine, and finally divide it by $r=2^{sw}$ which is accomplished by ignoring the lower s words of u. Since m=t*n’ mod r, and the reduction process proceeds word by word, we can use $n_0’= n’ \bmod 2^w$ instead of n’. This observation applies to all five methods. Finally we obtain the number u in s+1 words. The multi-precision subtraction in Step 3 of MonPro is then performed to reduce u if necessary.

```
for i=0 to s-1
    C := 0
    for j=0 to s-1
        (C,S) := t[i+j] + a[j]*b[i] + C
        t[i+j] := S
    t[i+s] := C
for i=0 to s-1
    C := 0
    m := t[i]*n'[0] mod W
    for j=0 to s-1
        (C,S) := t[i+j] + m*n[j] + C
        t[i+j] := S
    ADD (t[i+s],C)
for j=0 to s
    u[j] := t[j+s]
```

Step 3 is performed in the same way for all algorithms described in this chapter, and thus, we will not repeat this step in the description of the algorithms:

```
B := 0
        for i=0 to s-1
                (B,D) := u[i] - n[i] - B
                t[i] := D
        (B,D) := u[s] - B
        t[s] := D
        if B=0 then
                return t[0], t[1], ... , t[s-1]
        else
                return u[0], u[1], ... , u[s-1]
```

ADD function adds the second parameter (carry) to the word given in the first parameter. If another carry occurs in this addition, the new carry is propagated onto following next high order words. Generated carries are propagated until no further carry is generated.

The value n_0', which is defined as the inverse of the least significant word of n modulo 2^w, i.e., $n_0' = -n_0^{-1} \bmod 2^w$, can be computed using a very simple algorithm. Furthermore, the reason for separating the product computation a*b from the rest of the steps for computing u is that when a=b, we can optimize the Montgomery multiplication algorithm for squaring. The optimization of squaring is achieved because almost half of the single-precision multiplications can be skipped since $a_i * a_j = a_j * a_i$.

Coarsely Integrated Operand Scanning (CIOS) Method

The next method, the Coarsely Integrated Operand Scanning method, improves on the first one by integrating the multiplication and reduction steps. Specifically, instead of computing the entire product a∗b, then reducing, we alternate between iterations of the outer loops for multiplication and reduction. We can do this since the value of m in the reduction loop depends only on one word of t. which is completely computed in the outer multiplication loop. We say that the integration in this method is *coarse* because it alternates between iterations of the outer loop. In the next method, we will alternate between iterations of the inner loop.

```
for i=0 to s-1
        C := 0
        for j=0 to s-1
                (C,S) := t[j] + a[j]*b[i] + C
                t[j] := S
        (C,S) := t[s] + C
        t[s] := S
        t[s+1] := C
        m := t[0]*n'[0] mod W
        (C,S) := t[0] + m*n[0]
        for j=1 to s-1
                (C,S) := t[j] + m*n[j] + C
                t[j-1] := S
        (C,S) := t[s] + C
        t[s-1] := S
        t[s] := t[s+1] + C
```

Finely Integrated Operand Scanning (FIOS) Method

This method integrates the two inner loops of the CIOS method into one by computing the multiplications and additions in the same loop. Multiplication of one word of a and b, and multiplication of m and one word of n are computed in same loop, and then added to form the final t. In this case, lowest word of t must be computed before entering into the loop since m depends on this value. The major difference between the CIOS method and this method is that the FIOS method has only one inner loop. However, carry propagation in the FIOS method requires an extra carry bit, outbalancing the loop elimination.

```
for i=0 to s-1
      (C,S) := t[0] + a[0]*b[i]
      ADD(t[1],C)
      m := S*n'[0] mod W
      (C,S) := S + m*n[0]
      for j=1 to s-1
            (C,S) := t[j] + a[j]*b[i] + C
            ADD(t[j+1],C)
            (C,S) := S + m*n[j]
            t[j-1] := S
      (C,S) := t[s] + C
      t[s-1] := S
      t[s] := t[s+1] + C
      t[s+1] := 0
```

Finely Integrated Product Scanning (FIPS) Method

Like the previous one, this method interleaves the computations a*b and m*n, but here both computations are in the product-scanning form. The method keeps the values of t and u in the same s-word array t. The first loop given below computes one part of the product a*b and then adds t*n to it. The three-word array m, is used as the partial product accumulator for the products a * b and t * n. Note that, there are two i-loops running through the words of b and n. The pseudocode for the FIPS algorithm is given on the left. The final subtraction, as always, should be appended at the end of the given pseudocode.

```
for i=0 to s-1
      for j=0 to i-1
            (C,S) := m[0] + a[j]*b[i-j]
            ADD(m[1],C)
            (C,S) := S + t[j]*n[i-j]
            m[0] := S
            ADD(m[1],C)
      (C,S) := m[0] + a[i]*b[0]
      ADD(m[1],C)
      t[i] := S*n'[0] mod W
      (C,S) := S + t[i]*n[0]
      ADD(m[1],C)
      m[0] := m[1]
      m[1] := m[2]
      m[2] := 0
for i=s to 2s-1
      for j=i-s+1 to s-1
            (C,S) := m[0] + a[j]*b[i-j]
            ADD(m[1],C)
            (C,S) := S + t[j]*n[i-j]
            m[0] := S
            ADD(m[1],C)
      t[i-s] := m[0]
      m[0] := m[1]
      m[1] := m[2]
      m[2] := 0
```

Coarsely Integrated Hybrid Scanning (CIHS) Method

This method is a modification of the SOS method, illustrating yet another approach to Montgomery multiplication. We call it a *hybrid scanning* method because it mixes the product-scanning and operand-scanning forms of multiplication. (Reduction is just in the operand-scanning form.) First, we split the computation of a*b into two loops. The second loop shifts the intermediate result one word at a time at the end of each iteration. The splitting of multiplication is possible because m is computed by multiplying the i^th word of t by n_0'. Thus, the multiplication a∗b can be simplified by postponing the word multiplications required for the most significant half of t to the second i-loop. The multiplication loop can be integrated into the second main i-loop, computing one partial product in each iteration. In the first stage, (n-j) words of the j^th partial product of a∗b are computed and added to t. The multiplication of m∗n is then interleaved with the addition a∗b + m∗n. The division by r is performed by shifting one word at a time within the i-loop.

```
for i=0 to s-1
      C := 0
      for j=0 to s-i-1
            (C,S) := t[i+j] + a[j]*b[i] + C
            t[i+j] := S
      (C,S) := t[s] + C
      t[s] := S
      t[s+1] := C for i=0 to s-1
      m := t[0]*n'[0] mod W
      (C,S) := t[0] + m*n[0]
      for j=1 to s-1
            (C,S) := t[j] + m*n[j] + C
            t[j-1] := S
      (C,S) := t[s] + C
      t[s-1] := S
      t[s] := t[s+1] + C
      t[s+1] := 0
      for j=i+1 to s-1
            (C,S) := t[s-1] + b[j]*a[s-j+i]
            t[s-1] := S
            (C,S) := t[s] + C
            t[s] := S
            t[s+1] := t[s+1]+C
```