

Using Streaming SIMD Extensions (SSE2) to Perform Big Multiplications

Version 2.0

7/00

Order Number 248606-001

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Pentium® II processors, Pentium III processors and Pentium 4 processors may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Third-party brands and names are the property of their respective owners.

Copyright © Intel Corporation 1999, 2000

Table of Contents

1	Introduction.....	5
2	Big Number Multiplication.....	5
2.1	Applications for Big Number Multiplication.....	6
2.2	Background of Big Number Multiplication.....	6
2.3	Implementing Big Number Multiplication.....	6
2.3.1	Techniques.....	7
3	Performance.....	9
3.1	Gains/Improvements.....	9
3.2	Considerations.....	9
4	Conclusion.....	9
5	C/C++ Coding Example.....	9
6	SSE2 Assembly Code Example.....	10
	Appendix A - Performance Data.....	A-1
	Performance Data Revision History.....	A-1
	Test Systems Configuration.....	A-2

Revision History

Revision	Revision History	Date
2.0	Pentium [®] 4 processor update	7/00
1.0	Original publication of document	9/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

[1] B. Schneier, *Applied Cryptography, Second Edition*, John Wiley & Sons, 1996.

[2] D. Knuth, *The Art of Computer Programming, Vol. II, Seminumerical Algorithms, Third Edition*, Addison Wesley, 1998.

[3] *Intel[®] Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*, Intel Corporation (Order Number 243191)

1 Introduction

The Streaming SIMD Extensions 2 (SSE2) technology introduces new Single Instruction Multiple Data (SIMD) double-precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel® architecture. The double-precision SIMD instructions extend functionality in a manner analogous to the single-precision instructions introduced with the Streaming SIMD Extensions (SSE). The 128-bit SIMD integer extensions are a full superset of the 64-bit integer SIMD instructions, with additional instructions to support more integer data types, conversion between integer and floating-point data types, and efficient operations between the caches and system memory. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, spatial (3D) audio, video encoding/decoding, encryption, and scientific application. This application note discusses how to use the 128-bit SIMD integer instructions to implement multiplication where the size of the multiplicands is very large. We include examples of code that exploit the SSE2 instructions.

Big Number Multiplication (as we will refer to it) is used for multiplying two numbers where the size of either multiplicand is larger than the native size of the target machine's instruction set. The basic algorithm is the same as that taught in grade schools for multiplying two numbers using a position-based notation, but is extended to handle very large integer numbers. This document describes an implementation for that algorithm using SSE2 instructions to provide significant performance enhancements.

For extremely large numbers, more sophisticated algorithms can be found in Knuth, 98 [2]. Some of these algorithms are built upon smaller multiplications that can use the techniques described here. Others have too large an overhead to become faster than this SSE2-optimized position-based notation algorithm until the size of the input numbers are extremely large.

2 Big Number Multiplication

The algorithm for multiplying two large numbers is well known to any grade school student who is proficient at arithmetic. 'Large numbers' in our case means that the numbers are so large that multiplication would cause overflow on the target machine's instruction set.

The component parts of the familiar 'multiply-add-carry' algorithm are visually displayed in Figure 1, and a naïve implementation in C code is given in Example 1 of Section 5. In Figure 1 the subscripts are numbered in the traditional order (most significant bytes to the left), but the actual implementation reverses the byte order to conform to a more natural order for little-endian machines. The C code captures the reversed order. The code implementation differs from the hand-computed version in that it does not record the partial sums explicitly. Instead, the partial sums are added to the intermediate total T as the algorithm proceeds, finally ending with the entire result in T.

The implementation discussed in this application note is a modified form of this same algorithm. It is changed somewhat to better take advantage of the performance gains offered by the SSE2 instructions. The specific changes are discussed in the Section 2.3 Implementing Big Number Multiplication.

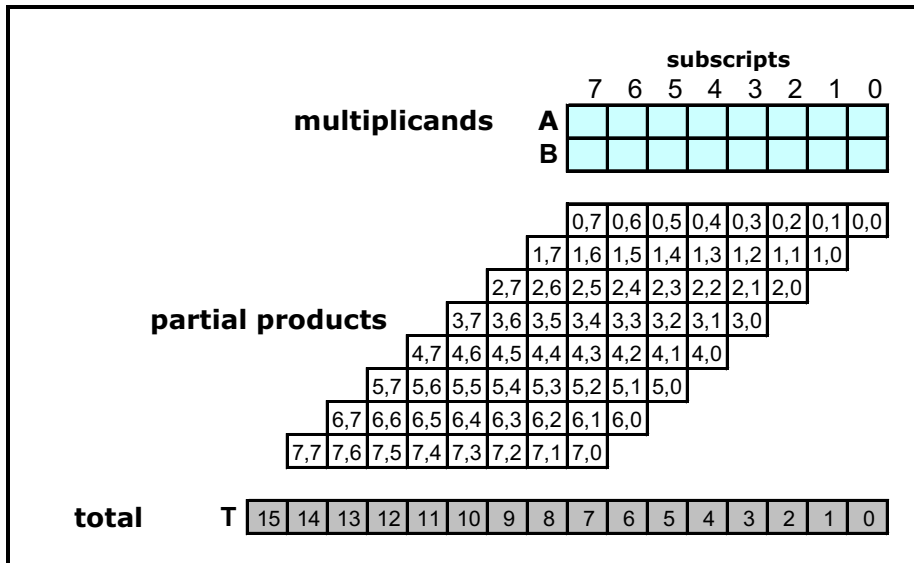


Figure 1: Visual representation of multiplication algorithm

2.1 Applications for Big Number Multiplication

This function is heavily used in number-theoretic cryptographic applications. In particular, it is heavily used in public-key cryptography. The importance of these cryptosystems has risen rapidly with the growth of the Internet. They provide the basis for secure communications between parties who would not otherwise have an agreed-upon channel for secure exchange of information. Some particular examples of algorithms and protocols that depend on large multiplications are:

- RSA public-key encryption and signatures
- Diffie-Hellman Key Exchange
- ElGamal public-key encryption
- DSA public-key signatures
- Elliptic curve public-key systems based on modular arithmetic over a prime Galois Field
- SSL (Secure Sockets Layer, widely used on the Internet for e-commerce), which is an indirect application of the previously mentioned cryptosystems.

See B. Schneier, 96 [1] for an excellent overview of the field.

The multiplication of large numbers is also important in scientific and research applications where extreme accuracy is important.

2.2 Background of Big Number Multiplication

This function has been used since ancient times. The originator is unknown.

2.3 Implementing Big Number Multiplication

The SSE2 instructions provide several new instructions that are useful in improving performance of this algorithm. These instructions are useful because they significantly reduce the number of computations

required to do the full multiplications. The formal properties of these instructions are provided in the *Intel® Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*.

- `pmuludq` – Packed Multiply Double-Word to Quad-Word Unsigned. This instruction multiplies two unsigned double-words in the destination register with two unsigned double-words in the source register. The full 64-bit results of both multiplications are written to the upper and lower halves of the destination register. The use of 32-bit (versus 16-bit) multiply allows the same multiplication with only one-fourth as many instructions.
- `paddq` – Packed Add Quad-word 64 bits. This instruction adds the lower and upper halves of the source register and destination register and deposits the result in the destination register

The instructions listed below provide the mechanism for taking advantage of the SSE2 instructions above in performing the multiplication.

- `pshufd` – Packed Shuffle Double-Word. This instruction allows any permutation or combination of the words in the source register to be placed into the destination register.
- `pslldq`, `psrldq` – Packed Shift Left/Right Logical Double-Quadword. This instruction does a logical shift of the entire 128-bit register left or right by the specified number of bytes.
- `psllq`, `psrlq` – Packed Shift Left/Right Logical – This instruction logically shifts the bits each half of the 128-bit register left or right. The number of bits to shift is specified in the instruction.
- `movdqua` – Move Double QuadWord Aligned. This instruction loads or stores 16-bytes. They must be aligned on a 16-byte boundary.

2.3.1 Techniques

The following techniques are used in the optimized implementation of this algorithm.

- **Use `pmuludq` to reduce the number of multiplications, `paddq` to do the sums.**

This is the primary source of gain from using SSE2 instructions for big multiplications. Suppose we were required to use 16-bit multiplications without these instructions. In order to accomplish the work done by a single 32-bit by 32-bit multiplication we would have to do four multiplications and four additions, instead of the single operation performed by `pmuludq`.

- **Use 128-bit SSE2 instructions whenever feasible.**

The obvious benefit is that this allows multiple computations to be accomplished for the cost of a single instruction. The price that must be paid in this is that the computation does not have a ‘natural’ SIMD structure. Therefore there is some overhead in juggling data to take advantage of SIMD. The richness of the SSE2 data manipulation instructions makes this task less burdensome.

- **Break the big multiplication into smaller sub-calculations.**

The algorithm reads four double-words at a time from each multiplicand into two 16-byte registers. This reduces the number of times that each section of the data must be loaded. Figure 2 shows the area of data calculated for each sub-calculations in the inner loop of the algorithm.

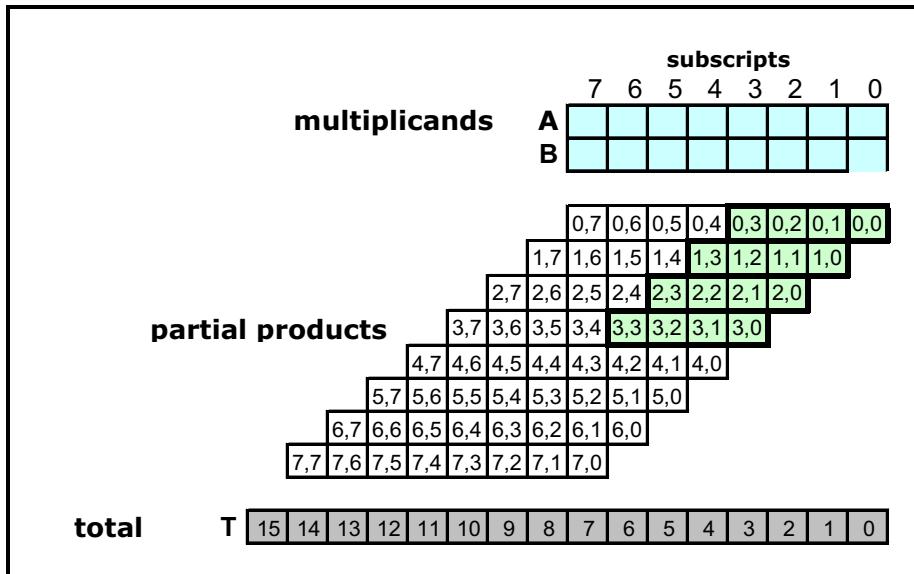


Figure 2 : Breaking into Sub-calculations

- **Compute all the partial products before doing the carry propagation.**

This technique reduces the number of juggling operations required for intermediate steps in a more straightforward implementation of the algorithm. The technique is as follows: As each partial product is computed, it is separated into a lower half (the sum), and an upper half (the carry). A temporary 64-bit each carry/sum pair is maintained for each column. After all the partial sums have been accumulated and summed in this fashion, a final pass is made to propagate the carries across all the columns, producing the final result.

By separating the carry/sum portions of the partial products and summing them separately we are able to avoid the overhead of propagating the carry across each row of partial products, and avoid potential overflow problems. The gain from doing the algorithm this way is possible because the `paddq` instruction allows straightforward addition of 64-bit quantities. This technique works for work for multiplicands of any size likely to be encountered in practice.

However, for specialized cases, there is an additional technique that can give even larger performance gains under proper conditions. If the length of the multiplicands is known in advance, we can avoid the splitting into carry/sum pairs by using a suitable recoding so that a reduced number of bits are placed into each digit. This allows the addition of several partial products without overflow. The particular case we have chosen to illustrate is for 1024 by 1024 bit multiplication. We split each 1024 bit quantity into 29 bit parts. This requires 36 digits to represent the number. But we can add 64 of the resulting products together with no possibility of overflow. By arranging the intermediate results judiciously, we are able to do two multiplications and then use `paddq` to add directly without any need to split into the carry-sum pairs. This increases the total number of multiplications needed (from 1024 to 1296 in this case), but it turns out that increased performance is greater than the cost for this effort since the cost of operations for splitting is fairly high.

3 Performance

3.1 Gains/Improvements

There are significant performance gains for the Big Number Multiplication from using SSE2 instructions. There are two main sources for the performance gains:

- 32-bit multiplications and 64-bit additions allow bigger chunks of the problem to be solved by a single computation.
- Increased parallelism is available through the use of SIMD instructions.

3.2 Considerations

In order to take maximum advantage of the performance gains offered by the new architectural enhancements, it is necessary to alter the algorithm substantially from the naïve implementation.

4 Conclusion

In summary, there are significant benefits to using SSE2 instructions to perform the Big Multiplication algorithm. Since this function is heavily used in public-key cryptosystems, SSE2 instructions should provide significant performance benefits for those applications.

5 C/C++ Coding Example

The following code captures the naïve implementation of this algorithm. It does not capture the optimizations that are detailed in other sections of this document.

Example 1: Naïve Algorithm

```
// A[0..i-1] = Multiplicand 1
// B[0..j-1] = Multiplicand 2
// T[0..i+j-1] = Result
// WORDS = number of words in multiplicand
// Assumptions: T is initialized to zero
//             A, B are in LSB to MSB order

unsigned short A[WORDS];
unsigned short B[WORDS];
unsigned short T[WORDS];
BigMultiply ()
{
    unsigned short i,j,carry;
    unsigned long result;
    for(i = 0; i < WORDS; i++)
    {
        for(carry = 0, j = 0; j < WORDS, j++)
        {
            result = A[i] * B[j] + t[i+j] + carry;
            t[i+j] = LoWord(result);
            carry = HiWord(result);
        }
    }
    T[i+j] = carry;
}
```

6 SSE2 Assembly Code Example

This code implements the optimizations discussed in this document. The code assumes that the multiplicands and the results are 16-byte aligned. It further assumes that the size of each multiplicand is a multiple of 128 bits. It is a simple exercise to enhance the function to handle different sized multiplicands. If the length of the multiplicands is not a multiple of 128 bits, some additional code would be required to handle the anomalous conditions.

The conversion from 32-bit to 29-bit form (and back) is not shown here. For many applications, the overhead for this optimization can be amortized over many iterations. Note also that the code uses two arrays for maintaining the partial products. This was done to avoid loads and stores into the 128-bit registers that were not aligned on 16-byte boundaries. There is a significant performance penalty in doing otherwise.

A brief summary of the algorithm (in pseudocode, ignoring many details) is as follows:

Example 2 : PseudoCode Big Multiply

```
// Some variables and functions shown below do not exist per se in
// the actual code, but are inserted here for clarity

For each set of four values in A
  For each set of four values in B
    For each possible pair A[i],B[j] (taken two at time)
      Compute A[i]*B[j]
      Split product into 32-bit carry/sum portions
      Add product to the appropriately aligned set of partial products
Carry = 0
For each column of output
  Add the corresponding two columns of partial products
  Put the lower 29 bits into the output column
  Carry = Upper 35 bits of the results
Output the carry
```

Example 3 : Big Multiply Assembly Code

```
#define DIGITS 36
#define DIGITSTIMESTWO 72
#define DIGITSMINUSFOUR 32
```

```

// This version uses delayed carry and 29-bit multiplies.
// Specifically, no carries are done until all of the partial sums have
// completed.  By using 29 bits only, we guarantee that the sum of all the
// partial sums will fit in 64 bits.  This makes it much cleaner to support
// the sequence of
//   PSHUFD (align two multiplicand digits in halves of two XMM registers)
//   PMULUDQ (do the multiply)
//   PADDQ  (store the result in the correct partial sum)
// Also, we use two result arrays that we can always do 16-byte aligned
// stores and load.  At the end, the two result arrays are 'stitched'
// together at the same time the carry is being propagated.
// The added number of multiplications required for the shortening from
// 32 to 29 bits is easily absorbed by the speedup in the rest of the
// algorithm.
//
int multiply ( int *a, int *b, int *t, int *u)
{
    _asm {
        // t and u are intermediate results.
        // t will be used for the final results at the end
        mov eax, dword ptr [a]
        mov ebx, dword ptr [b]
        mov ecx, dword ptr [t]
        mov edx, dword ptr [u]
        mov eax,0
    outer:
        push ebx
        mov ebx, dword ptr [b]
        // load b[i]..b[i+4]
        movdqa xmm1, [ebx + eax*4]
        pop ebx
        movdqa xmm4, [ecx + eax*8+16]
        movdqa xmm5, [edx + eax*8+16]
        movdqa xmm6, [edx + eax*8]
        movdqa xmm7, [ecx + eax*8]
        mov ebx,0
    inner:
        mov edi, eax

```

```
add edi, ebx
push eax
mov eax, dword ptr [a]

// load a[j]..b[j+4]
movdqa xmm0, [eax + ebx*4]
pop eax

// There is an intricate little dance below to:
// 1)Do the minimal number of PSHUFDs
// 2)Do the minimal number of loads, stores
// 3)Put the maximal distance between obviously
//   dependent instructions.
// 4)Make sure the result array pieces are 16-byte
//   aligned
// We are operating on permutations and combinations
// of a 4x4 section of the multiplication
//
// The technique for minimizing PSHUFD is to
// alternate the use of the target registers
// for the PMULUDQ instruction so that we are
// temporarily through with a register before
// we step on it.
//
pshufd xmm3, xmm1, 0x0
pshufd xmm2, xmm0, 0x32
pmuludq xmm3, xmm2
paddq xmm4, xmm3
pshufd xmm3, xmm1, 0x11
pmuludq xmm2, xmm3
paddq xmm5, xmm2
pshufd xmm2, xmm0, 0x10
pmuludq xmm3, xmm2
paddq xmm6, xmm3
movdqa [edx + edi*8],xmm6
movdqa xmm6, [edx + edi*8 + 32]
pshufd xmm3, xmm1, 0x0
pmuludq xmm3, xmm2
```

```

    paddq xmm7, xmm3
    movdqa [ecx + edi*8],xmm7
    movdqa xmm7, [ecx + edi*8 + 32]
    pshufd xmm3, xmm1, 0x22
    pmuludq xmm3, xmm2
    paddq xmm4, xmm3
    movdqa [ecx + edi*8+16],xmm4
    movdqa xmm4, [ecx + edi*8+48]
    pshufd xmm3, xmm1, 0x33
    pmuludq xmm2, xmm3
    paddq xmm5, xmm2
    movdqa [edx + edi*8+16],xmm5
    movdqa xmm5, [edx + edi*8+48]
    pshufd xmm2, xmm0, 0x32
    pmuludq xmm3, xmm2
    paddq xmm6, xmm3
    pshufd xmm3, xmm1, 0x22
    pmuludq xmm3, xmm2
    paddq xmm7, xmm3
    add ebx,4
    cmp ebx, DIGITS
    jl inner

    movdqa [edx + edi*8 + 32], xmm6
    movdqa [ecx + edi*8 + 32], xmm7
    add eax,4
    cmp eax, DIGITS
    jl outer

// Finally, we do the 29 bit carry calculation
// and add the two result arrays
pxor mm6,mm6
mov eax, 0x1fffffff
movd mm7, eax
movq mm0, [ecx]
movq mm6, mm0
pand mm0, mm7
movd [ecx], mm0
psrlq mm6, 29
mov ebx,1

```

```
cloop:
    movq mm0, [ecx + ebx*8]
    movq mm1, [edx + ebx*8-8]
    paddq mm0, mm1
    paddq mm0, mm6
    movq mm6, mm0
    pand mm6, mm7
    movd [ecx + ebx * 4], mm6
    psrlq mm0, 29

    movq mm6, [ecx + ebx*8+8]
    movq mm1, [edx + ebx*8-0]
    paddq mm6, mm1
    paddq mm6, mm0
    movq mm0, mm6
    pand mm0, mm7
    movd [ecx + ebx * 4+4], mm0
    psrlq mm6, 29
    add ebx,2
    cmp ebx,DIGITSTIMESTWO-1
    jl cloop
    sub ebx,1
    movd [ecx + ebx * 4], mm6
}
return 0;
}
```

Appendix A - Performance Data

Performance Data Revision History

Table 1: Performance Data for Big Number Implementations

Performance Data in Microseconds			
	Number/size of multiplication operation	Pentium® III Processor (733 MHz)	Pentium 4 Processor (1.2 GHz)
Standard Algorithm in C	1 16x16	37.5	30.0
Streaming SIMD Extensions 2 (SSE2) Fully Optimized ASM	2 32x32	N/A	2.80

Table 2: Speedups from Table 1 Performance Data

Implementations and Platforms	Speedup
Pentium 4 Processor (SSE2 ASM vs. Standard C Algorithm)	10.7
SSE2 ASM on a Pentium 4 Processor vs. Standard C on a PentiumIII Processor	13.4

Performance was measured assuming a normal cache environment. Caching is not a significant factor in the performance of this algorithm. Performance was measured using a 733 MHz Pentium® III processor and a 1.2 GHz Pentium 4 processor. See Test Systems Configuration on page A-2, for a detailed description of the test systems.

This table summarizes the relative performance of two algorithms for computing the Big Number Multiplication. The first uses 16-bit multiplications. It is included to demonstrate the performance of the standard algorithm with no special optimizations. The second algorithm is a fully optimized version using the technique of delaying the carry propagations until after all partial products are computed. It also used the 29-bit technique described in the application note.

If we assume that doubling the operand size will quadruple the performance, and that doing two such operations in parallel using SIMD will again double the performance, we might expect a theoretical improvement of 8x. In fact, we were able to do better than that. Here are some of the reasons:

- Even though doubling the size of the multiplications from 16x16 to 32x32 should theoretically give a four times boost in performance, the actual result is better because the percentage of overhead instructions (e.g. loading and storing) is larger for the shorter multiplications.
- The optimized algorithm is more carefully tuned to the architecture of the Pentium 4 Processor.

Test Systems Configuration

Table 3: Pentium III Configuration

Processor	Pentium III Processor at 733 MHz
System	Intel [®] Desktop Board VC820
Bios Version	VC82010A.86A.0028.P10
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster [†] Annihilator [†] Pro AGP nVidia GeForce256 [†] DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows [†] 2000 Build 2195

Table 4: Pentium 4 Configuration

Processor	Pentium 4 Processor at 1.2 GHz
System	Intel Desktop Board D850GB
Bios Version	GB85010A.86A.0014.D.0007201756
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows 2000 Build 2195