AP-930

Streaming SIMD Extensions - Matrix Multiplication

June 1999

Order Number: 245045-001

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® III processor, Pentium II processor, Pentium Pro processor and Intel® Celeron™ processor 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.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained bycalling 1-800-548-4725 or by visiting Intel’s website at http://www.intel.com

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

Table of Contents

Revision History

Revision	Revision History	Date
1.0	First external publication	3/99
0.99	Internal publication	1/99

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. *Using the RDTSC Instruction for Performance Monitoring,*
   http://www.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM

1 Introduction

This application note describes the multiplication of two matrices using Streaming SIMD Extensions.

The performance of the assembler code using Streaming SIMD Extensions, which implements the multiplication of two 6x6-matrices, is approximately 2.1x times better than an implementation using FPU instructions (See section 3.1).

2 Implementation

Each SIMD floating point register of the Pentium® III processor can hold 4 single precision float numbers, which may be processed effectively using SIMD commands. Let's denote the result of multiplication of two matrices *B* and *C* as *A*:

$$A_{mxk} = B_{mxn} \times C_{nxk}$$

Because each row of matrix *A* depends on all rows of matrix *C*, but only on one row of matrix *B*, it is advantageous to store some (or all) data of matrix *C* in Pentium® III registers and, when necessary, to load the elements of matrix *B* one by one.

Based on the matrix dimensions m, n and k, specific implementation may require splitting original matrices into pieces to take into account the size of the SIMD floating point registers (see section 2.1). If all dimensions are not greater then 4, it is possible to process all data at once.

In order to minimize latency of the instructions, unrolling of all loops is highly desirable.

One important case requires special consideration. It is multiplication of a matrix by a vector, which is frequently used in computer graphics. In the case of multiplication of a 4x4-matrix with a 4x1-vector, we may represent it as

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \times \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}$$

Applying 4 **mulps** instructions, we may easily get 4 **xmm** (SIMD floating point) registers containing (see section 4.3):

register				
xmm0	b_{11}*c_1	b_{12}*c_1	b_{13}*c_1	b_{14}*c_1
xmm1	b_{21}*c_2	b_{22}*c_2	b_{23}*c_2	b_{24}*c_2
xmm2	b_{31}*c_3	b_{32}*c_3	b_{33}*c_3	b_{34}*c_3
xmm3	b_{41}*c_4	b_{42}*c_4	b_{43}*c_4	b_{44}*c_4

If we execute

```
        addps   xmm0, xmm1
        addps   xmm2, xmm3
        addps   xmm0, xmm2
```

register **xmm0** will contain $B_{4x4}{}^{T} \times C_{4x1}$! Computation of $B_{4x4} \times C_{4x1}$ would require some additional **shufps** instructions to effectively transform the matrix. This example shows that computation of

B^T x C may be considerably faster than computation of B x C (see section 3.1). If an application contains a lot of vector transformations, it may be beneficial to change matrix representation from row order to column order.

2.1 Multiplication of Two 6x6 Matrices

Since each SIMD floating point register of the Pentium® III processor can hold 4 floating-point numbers, we need to split the original 6x6-matrices to process them effectively using SIMD commands.

The described method of evaluating the product of matrices $A_{6 \div 6} = B_{6x6} \times C_{6x6}$ is implemented in two stages using the block method.

Let us denote $A_{6x6} = \left[A_{6x4} \quad A_{6x2} \right]$, $C_{6x6} = \left[C_{6x4} \quad C_{6x2} \right]$.

1. Evaluate the matrix $A_{6x4} = B_{6x6} \times C_{6x4}$
2. Evaluate the matrix $A_{6x2} = B_{6x6} \times C_{6x2}$

Because each row of matrix A_{6x4} depends on all rows of matrix C_{6x4} but only on one row of matrix B_{6x6}, it is advantageous to store the whole matrix C_{6x4} in 6 Pentium® III registers and, when necessary, to load the elements of matrix B_{6x6} one by one. Thus, the whole row of matrix A_{6x4} is evaluated simultaneously. Matrix A_{6x2} is evaluated in a similar way.

3 Performance

The performance of matrix multiplication can be increased if we use Streaming SIMD Extensions commands.

Streaming SIMD Extensions allow an increase in performance as compared with the scalar floating-point code due to single-instruction-multiple-data processing. When the data is stored in a row or column basis, one instruction can operate on 4 data elements. This allows processing 4 elements of the matrix row or matrix column in one instruction.

Table 1 compares performance (in processor cycles) of matrix multiplication for different sizes using:

- Assembler code and FPU.
- Streaming SIMD Extensions (Pentium® III).

Processor cycles were measured by using the **rdtsc** instruction (see http://www.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM).

Numbers in this table represent warm cache performance including access to the matrix data, but exclude overhead associated with parameter passing to the function (see section 4.3). It may be slightly improved using the **__fastcall** calling convention.

Table 1: Performance Gains Using Streaming SIMD Extensions [1]

Matrix Operation	FPU (cycles)	SIMD (cycles)
B_{3x3} x C_{3x1}	31	29
B_{3x3}^{T} x C_{3x1}	31	23
B_{3x3} x C_{3x3}	79	59
B_{4x4} x C_{4x1}	53	31
B_{4x4}^{T} x C_{4x1}	53	27
B_{4x4} x C_{4x4}	172	90
B_{6x6} x C_{6x1}	113	60
B_{6x6} x C_{6x6}	652	307

4 Source Code

Three different code examples are represented below. The first example is multiplication of 6x6 matrices using the floating point unit; the second example is multiplication of 6x6-matrices using Streaming SIMD Extensions. The last code example represents a comparison of matrix multiplication performance using Pentium® II and Pentium® III instructions for various matrix sizes. Performance figures from this example were used in Table 3.1.

These examples require the Intel® C/C++ Compiler (http://support.intel.com/support/performancetools/c/).

4.1 Assembler Code with FPU

```
//    Parameters for the macros:
//    w    -    width (# of columns) for the particular matrix
//    t    -    1 for transpose, 0 for not (aka a, b for 1st and 2nd matrices)
//    I    -    statement(s) for m[i][j]
//    j    -    will be generated
//    k    -    kind of third index in the multiplication
//    l    -    width of the result
//    m    -    width of the first matrix
//    n    -    width of the second matrix
//    a    -    1 for transpose, 0 for not (matrix A)
//    b    -    1 for transpose, 0 for not (matrix B)


// Offset for m[i][j], w is an row width, t == 1 for transposed access.
#define mi(w, t, i, j)  4 * ((i * w + j) * (1-t) + (j * w + i) * t)

// Load & multiply.
#define flm(k, i, j, m, n, a, b)                         \
    __asm    fld    dword ptr [ebx + mi(m, a, i, k)] \
    __asm    fmul   dword ptr [ecx + mi(n, b, k, j)]

#define e6(i, j, l, m, n, a, b)  \
    flm(0, i, j, m, n, a, b)    \
```

[1] These measurements are based on tests run on a 450MHz, 64MB SDRAM, 100MHz bus Pentium® III processor. This is the first Pentium® III processor release. Performance on future releases of Pentium® III processor may vary.

```
    flm(1, i, j, m, n, a, b)    \
    flm(2, i, j, m, n, a, b)    \
    flm(3, i, j, m, n, a, b)    \
    flm(4, i, j, m, n, a, b)    \
    flm(5, i, j, m, n, a, b)    \
    __asm faddp st(1), st(0)    \
    __asm fxch  st(2)           \
    __asm faddp st(1), st(0)    \
    __asm faddp st(1), st(0)    \
    __asm fxch  st(2)           \
    __asm faddp st(1), st(0)    \
    __asm faddp st(1), st(0)    \
    __asm fstp  dword ptr [eax + mi(l, 0, i, j)]


// Parameters:
// input:
//      m1 - pointer to array of 36 floats (source matrix 1)
//      m2 - pointer to array of 36 floats (source matrix 2)
// output:
//      dst - pointer to array of 36 floats (m1 * m2)

void fpu_Mult00_6x6_6x6(float *m1, float *m2, float *dst)
{
    __asm mov    ebx, DWORD PTR m1
    __asm mov    ecx, DWORD PTR m2
    __asm mov    eax, DWORD PTR dst
    e6(0, 0, 6, 6, 6, 0, 0)
    e6(0, 1, 6, 6, 6, 0, 0)
    e6(0, 2, 6, 6, 6, 0, 0)
    e6(0, 3, 6, 6, 6, 0, 0)
    e6(0, 4, 6, 6, 6, 0, 0)
    e6(0, 5, 6, 6, 6, 0, 0)
    e6(1, 0, 6, 6, 6, 0, 0)
    e6(1, 1, 6, 6, 6, 0, 0)
    e6(1, 2, 6, 6, 6, 0, 0)
    e6(1, 3, 6, 6, 6, 0, 0)
    e6(1, 4, 6, 6, 6, 0, 0)
    e6(1, 5, 6, 6, 6, 0, 0)
    e6(2, 0, 6, 6, 6, 0, 0)
    e6(2, 1, 6, 6, 6, 0, 0)
    e6(2, 2, 6, 6, 6, 0, 0)
    e6(2, 3, 6, 6, 6, 0, 0)
    e6(2, 4, 6, 6, 6, 0, 0)
    e6(2, 5, 6, 6, 6, 0, 0)
    e6(3, 0, 6, 6, 6, 0, 0)
    e6(3, 1, 6, 6, 6, 0, 0)
    e6(3, 2, 6, 6, 6, 0, 0)
    e6(3, 3, 6, 6, 6, 0, 0)
    e6(3, 4, 6, 6, 6, 0, 0)
    e6(3, 5, 6, 6, 6, 0, 0)
    e6(4, 0, 6, 6, 6, 0, 0)
    e6(4, 1, 6, 6, 6, 0, 0)
    e6(4, 2, 6, 6, 6, 0, 0)
    e6(4, 3, 6, 6, 6, 0, 0)
    e6(4, 4, 6, 6, 6, 0, 0)
    e6(4, 5, 6, 6, 6, 0, 0)
    e6(5, 0, 6, 6, 6, 0, 0)
    e6(5, 1, 6, 6, 6, 0, 0)
    e6(5, 2, 6, 6, 6, 0, 0)
    e6(5, 3, 6, 6, 6, 0, 0)
    e6(5, 4, 6, 6, 6, 0, 0)
    e6(5, 5, 6, 6, 6, 0, 0)
}
```

4.2 C Code with Streaming SIMD Extensions

```
// Parameters:
// input:
//         m1 - pointer to array of 36 floats (source matrix 1)
//         m2 - pointer to array of 36 floats (source matrix 2)
// output:
//         dst - pointer to array of 36 floats (m1 * m2)

void sse_Mult00_6x6_6x6(float *m1, float *m2, float *dst)
{
    __m128    b0, b1, b2, b3, b4, b5;
    __m128    row, rslt, tmp;

    //    Loading first 4 columns of m2.

    b0 = _mm_loadh_pi(_mm_loadl_pi(b0, &m2[ 0]), &m2[ 2]);
    b1 = _mm_loadh_pi(_mm_loadl_pi(b1, &m2[ 6]), &m2[ 8]);
    b2 = _mm_loadh_pi(_mm_loadl_pi(b2, &m2[12]), &m2[14]);
    b3 = _mm_loadh_pi(_mm_loadl_pi(b3, &m2[18]), &m2[20]);
    b4 = _mm_loadh_pi(_mm_loadl_pi(b4, &m2[24]), &m2[26]);
    b5 = _mm_loadh_pi(_mm_loadl_pi(b5, &m2[30]), &m2[32]);

    //    Calculating first 4 elements in the first row of the destination matrix.

    row     = _mm_set_ps1(m1[0]);
    rslt    = _mm_mul_ps(row, b0);

    row     = _mm_set_ps1(m1[1]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b1));

    row     = _mm_set_ps1(m1[2]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b2));

    row     = _mm_set_ps1(m1[3]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b3));

    row     = _mm_set_ps1(m1[4]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b4));

    row     = _mm_set_ps1(m1[5]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b5));
```

```
_mm_store_ps(&dst[0], rslt);

//    Calculating first 4 elements in the second row of the destination matrix.

row     = _mm_set_ps1(m1[6]);
rslt    = _mm_mul_ps(row, b0);

row     = _mm_set_ps1(m1[7]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b1));

row     = _mm_set_ps1(m1[8]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b2));

row     = _mm_set_ps1(m1[9]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b3));

row     = _mm_set_ps1(m1[10]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b4));

row     = _mm_set_ps1(m1[11]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b5));

_mm_storel_pi((__m64*)&dst[6], rslt);
_mm_storeh_pi((__m64*)&dst[8], rslt);

//    Calculating first 4 elements in the third row of the destination matrix.

row     = _mm_set_ps1(m1[12]);
rslt    = _mm_mul_ps(row, b0);

row     = _mm_set_ps1(m1[13]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b1));

row     = _mm_set_ps1(m1[14]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b2));

row     = _mm_set_ps1(m1[15]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b3));

row     = _mm_set_ps1(m1[16]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b4));

row     = _mm_set_ps1(m1[17]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b5));

_mm_store_ps(&dst[12], rslt);

//    Calculating first 4 elements in the fourth row of the destination matrix.

row     = _mm_set_ps1(m1[18]);
rslt    = _mm_mul_ps(row, b0);
```

```
    row     = _mm_set_ps1(m1[19]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b1));

    row     = _mm_set_ps1(m1[20]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b2));

    row     = _mm_set_ps1(m1[21]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b3));

    row     = _mm_set_ps1(m1[22]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b4));

    row     = _mm_set_ps1(m1[23]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b5));

    _mm_storel_pi((__m64*)&dst[18], rslt);
    _mm_storeh_pi((__m64*)&dst[20], rslt);

    //    Calculating first 4 elements in the fifth row of the destination matrix.

    row     = _mm_set_ps1(m1[24]);
    rslt    = _mm_mul_ps(row, b0);

    row     = _mm_set_ps1(m1[25]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b1));

    row     = _mm_set_ps1(m1[26]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b2));

    row     = _mm_set_ps1(m1[27]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b3));

    row     = _mm_set_ps1(m1[28]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b4));

    row     = _mm_set_ps1(m1[29]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b5));

    _mm_store_ps(&dst[24], rslt);

    //    Calculating first 4 elements in the sixth row of the destination matrix.

    row     = _mm_set_ps1(m1[30]);
    rslt    = _mm_mul_ps(row, b0);

    row     = _mm_set_ps1(m1[31]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b1));

    row     = _mm_set_ps1(m1[32]);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b2));

    row     = _mm_set_ps1(m1[33]);
```

```
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b3));

row     = _mm_set_ps1(m1[34]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b4));

row     = _mm_set_ps1(m1[35]);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b5));

_mm_storel_pi((__m64*)&dst[30], rslt);
_mm_storeh_pi((__m64*)&dst[32], rslt);

//    Calculating last 2 columns of the destination matrix.

b0      = _mm_loadh_pi(_mm_loadl_pi(b0 , &m2[ 4]), &m2[10]);
b2      = _mm_loadh_pi(_mm_loadl_pi(b2 , &m2[16]), &m2[22]);
b4      = _mm_loadh_pi(_mm_loadl_pi(b4 , &m2[28]), &m2[34]);
b5      = _mm_shuffle_ps(b4 , b4 , 0x4E);

row     = _mm_loadh_pi(_mm_loadl_pi(row, &m1[ 0]),&m1[ 6]);
row     = _mm_shuffle_ps(row, row, 0xF0);

rslt    = _mm_mul_ps(row, b0);
row     = _mm_loadh_pi(_mm_loadl_pi(row, &m1[ 0]),&m1[ 6]);
b1      = _mm_shuffle_ps(b0 , b0 , 0x4E);
row     = _mm_shuffle_ps(row, row, 0xA5);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b1));

row     = _mm_loadh_pi(_mm_loadl_pi(row, &m1[ 2]),&m1[ 8]);
tmp     = row;
row     = _mm_shuffle_ps(row, row, 0xF0);

rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b2));
b3      = _mm_shuffle_ps(b2 , b2 , 0x4E);
tmp     = _mm_shuffle_ps(tmp, tmp, 0xA5);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(tmp, b3));

row     = _mm_loadh_pi(_mm_loadl_pi(row, &m1[ 4]),&m1[10]);
tmp     = _mm_shuffle_ps(row, row, 0xA5);
row     = _mm_shuffle_ps(row, row, 0xF0);
rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b4));
rslt    = _mm_add_ps(rslt, _mm_mul_ps(tmp, b5));

row     = _mm_loadh_pi(_mm_loadl_pi(row, &m1[12]),&m1[18]);
tmp     = _mm_shuffle_ps(row, row, 0xA5);
row     = _mm_shuffle_ps(row, row, 0xF0);

_mm_storel_pi((__m64*)&dst[ 4], rslt);
_mm_storeh_pi((__m64*)&dst[10], rslt);

rslt    = _mm_add_ps(_mm_mul_ps(row, b0),_mm_mul_ps(tmp,b1));

row     = _mm_loadh_pi(_mm_loadl_pi(row, &m1[14]),&m1[20]);
```

```
    tmp     = _mm_shuffle_ps(row, row, 0xA5);
    row     = _mm_shuffle_ps(row, row, 0xF0);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b2));
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(tmp, b3));

    row     = _mm_loadh_pi(_mm_loadl_pi(row, &m1[16]),&m1[22]);
    tmp     = _mm_shuffle_ps(row, row, 0xA5);
    row     = _mm_shuffle_ps(row, row, 0xF0);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b4));
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(tmp, b5));

    _mm_storel_pi((__m64*)&dst[16], rslt);
    _mm_storeh_pi((__m64*)&dst[22], rslt);

    row     = _mm_loadh_pi(_mm_loadl_pi(row, &m1[24]),&m1[30]);
    tmp     = _mm_shuffle_ps(row, row, 0xA5);
    row     = _mm_shuffle_ps(row, row, 0xF0);
    rslt    = _mm_add_ps(_mm_mul_ps(row, b0),_mm_mul_ps(tmp,b1));

    row     = _mm_loadh_pi(_mm_loadl_pi(row, &m1[26]),&m1[32]);
    tmp     = _mm_shuffle_ps(row, row, 0xA5);
    row     = _mm_shuffle_ps(row, row, 0xF0);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b2));
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(tmp, b3));

    row     = _mm_loadh_pi(_mm_loadl_pi(row, &m1[28]),&m1[34]);
    tmp     = _mm_shuffle_ps(row, row, 0xA5);
    row     = _mm_shuffle_ps(row, row, 0xF0);
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(row, b4));
    rslt    = _mm_add_ps(rslt, _mm_mul_ps(tmp, b5));

    _mm_storel_pi((__m64*)&dst[28], rslt);
    _mm_storeh_pi((__m64*)&dst[34], rslt);
}
```

4.3 Various Matrix Multiplication Examples

// mmtest.cpp

```
#include <windows.h>

#include <stdio.h>
#include <stdarg.h>
#include <time.h>
#include <math.h>

#include <xmmintrin.h>

#define   SAMPLES  100

long start = 0;
long end   = 0;
long save_ebx;
#define RecordTime(var)    \
        __asm cpuid        \
        __asm rdtsc        \
        __asm mov var, eax

#define StartRecordTime    \
        __asm mov save_ebx, ebx   \
        RecordTime(start)

#define StopRecordTime     \
        RecordTime(end)  \
        __asm mov ebx, save_ebx


int  i = 0;
long base = 0;
long tick = 0;
long ticks[SAMPLES];

int Duration(int sz = SAMPLES)
{
        long nclocks = 0;
        for (int i = 0 ; i < sz; i++) {
                if (!nclocks || ticks[i] < nclocks)
                        nclocks = ticks[i];
        }
        return int(nclocks - base);
}


void report(char* format, ...)
{
   va_list marker;
   char buf[500];
   vsprintf(buf, format, va_start(marker, format));
   puts(buf);
   // OutputDebugString(buf); OutputDebugString("\n");
}


#define START_MEASUREMENTS \
        SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL); \
        for (i = 0 ; i < SAMPLES; i++) {

#define END_MEASUREMENTS \
                ticks[i] = end - start; \
        } \
        SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_NORMAL); \
        report("Duration for %s:\t%i", testname, Duration());


// Offset for mat[i][j], w is an row width, t == 1 for transposed access.
#define mi(w, t, i, j)  4 * ((i * w + j) * (1-t) + (j * w + i) * t)

// Load & multiply.
#define flm(k, i, j, m, n, a, b) \
        __asm fld       dword ptr [edx + mi(m, a, i, k)] \
        __asm fmul      dword ptr [ecx + mi(n, b, k, j)]

// Load, multiply & add.
#define flma(k, i, j, m, n, a, b) flm(k, i, j, m, n, a, b) __asm faddp ST(1), ST(0)

#define e3(i, j, l, m, n, a, b) \
        flm (0, i, j, m, n, a, b)   \
        flma(1, i, j, m, n, a, b)   \
        flma(2, i, j, m, n, a, b)   \
        __asm fstp      dword ptr [eax + mi(l, 0, i, j)]

void PII_Mult_3x3_3x3(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm mov  edx, DWORD PTR src1
        __asm mov  ecx, DWORD PTR src2
        __asm mov  eax, DWORD PTR dst
        e3(0, 0, 3, 3, 3, 0, 0) e3(0, 1, 3, 3, 3, 0, 0) e3(0, 2, 3, 3, 3, 0, 0)
        e3(1, 0, 3, 3, 3, 0, 0) e3(1, 1, 3, 3, 3, 0, 0) e3(1, 2, 3, 3, 3, 0, 0)
        e3(2, 0, 3, 3, 3, 0, 0) e3(2, 1, 3, 3, 3, 0, 0) e3(2, 2, 3, 3, 3, 0, 0)
        StopRecordTime;
}

#define e4(i, j, l, m, n, a, b) \
        flm(0, i, j, m, n, a, b)    \
        flm(1, i, j, m, n, a, b)    \
        flm(2, i, j, m, n, a, b)    \
        flm(3, i, j, m, n, a, b)    \
        __asm faddp      st(1), st(0)    \
        __asm fxch       st(2)           \
        __asm faddp      st(1), st(0)    \
        __asm faddp      st(1), st(0)    \
        __asm fstp       dword ptr [eax + mi(l, 0, i, j)]


void PII_Mult00_4x4_4x4(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm mov  edx, DWORD PTR src1
        __asm mov  ecx, DWORD PTR src2
        __asm mov  eax, DWORD PTR dst
        e4(0, 0, 4, 4, 4, 0, 0)
        e4(0, 1, 4, 4, 4, 0, 0)
        e4(0, 2, 4, 4, 4, 0, 0)
        e4(0, 3, 4, 4, 4, 0, 0)
        e4(1, 0, 4, 4, 4, 0, 0)
        e4(1, 1, 4, 4, 4, 0, 0)
        e4(1, 2, 4, 4, 4, 0, 0)
        e4(1, 3, 4, 4, 4, 0, 0)
        e4(2, 0, 4, 4, 4, 0, 0)
        e4(2, 1, 4, 4, 4, 0, 0)
        e4(2, 2, 4, 4, 4, 0, 0)
        e4(2, 3, 4, 4, 4, 0, 0)
        e4(3, 0, 4, 4, 4, 0, 0)
        e4(3, 1, 4, 4, 4, 0, 0)
        e4(3, 2, 4, 4, 4, 0, 0)
        e4(3, 3, 4, 4, 4, 0, 0)
        StopRecordTime;
}

void PII_Mult00_4x4_4x1(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm mov  edx, DWORD PTR src1
        __asm mov  ecx, DWORD PTR src2
        __asm mov  eax, DWORD PTR dst
        e4(0, 0, 1, 4, 1, 0, 0)
        e4(1, 0, 1, 4, 1, 0, 0)
        e4(2, 0, 1, 4, 1, 0, 0)
        e4(3, 0, 1, 4, 1, 0, 0)
        StopRecordTime;
}


#define e6(i, j, l, m, n, a, b)       \
        flm(0, i, j, m, n, a, b)    \
        flm(1, i, j, m, n, a, b)    \
        flm(2, i, j, m, n, a, b)    \
        flm(3, i, j, m, n, a, b)    \
        flm(4, i, j, m, n, a, b)    \
        flm(5, i, j, m, n, a, b)    \
        __asm faddp st(1), st(0)    \
        __asm fxch  st(2)           \
        __asm faddp st(1), st(0)    \
        __asm faddp st(1), st(0)    \
        __asm fxch  st(2)           \
        __asm faddp st(1), st(0)    \
        __asm faddp st(1), st(0)    \
        __asm fstp  dword ptr [eax + mi(l, 0, i, j)]

void PII_Mult00_6x6_6x6(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm mov  edx, DWORD PTR src1
        __asm mov  ecx, DWORD PTR src2
        __asm mov  eax, DWORD PTR dst
        e6(0, 0, 6, 6, 6, 0, 0)
        e6(0, 1, 6, 6, 6, 0, 0)
        e6(0, 2, 6, 6, 6, 0, 0)
        e6(0, 3, 6, 6, 6, 0, 0)
        e6(0, 4, 6, 6, 6, 0, 0)
        e6(0, 5, 6, 6, 6, 0, 0)
        e6(1, 0, 6, 6, 6, 0, 0)
        e6(1, 1, 6, 6, 6, 0, 0)
        e6(1, 2, 6, 6, 6, 0, 0)
        e6(1, 3, 6, 6, 6, 0, 0)
        e6(1, 4, 6, 6, 6, 0, 0)
        e6(1, 5, 6, 6, 6, 0, 0)
        e6(2, 0, 6, 6, 6, 0, 0)
        e6(2, 1, 6, 6, 6, 0, 0)
        e6(2, 2, 6, 6, 6, 0, 0)
        e6(2, 3, 6, 6, 6, 0, 0)
        e6(2, 4, 6, 6, 6, 0, 0)
        e6(2, 5, 6, 6, 6, 0, 0)
        e6(3, 0, 6, 6, 6, 0, 0)
        e6(3, 1, 6, 6, 6, 0, 0)
        e6(3, 2, 6, 6, 6, 0, 0)
        e6(3, 3, 6, 6, 6, 0, 0)
        e6(3, 4, 6, 6, 6, 0, 0)
        e6(3, 5, 6, 6, 6, 0, 0)
        e6(4, 0, 6, 6, 6, 0, 0)
        e6(4, 1, 6, 6, 6, 0, 0)
        e6(4, 2, 6, 6, 6, 0, 0)
        e6(4, 3, 6, 6, 6, 0, 0)
        e6(4, 4, 6, 6, 6, 0, 0)
        e6(4, 5, 6, 6, 6, 0, 0)
        e6(5, 0, 6, 6, 6, 0, 0)
        e6(5, 1, 6, 6, 6, 0, 0)
        e6(5, 2, 6, 6, 6, 0, 0)
        e6(5, 3, 6, 6, 6, 0, 0)
        e6(5, 4, 6, 6, 6, 0, 0)
        e6(5, 5, 6, 6, 6, 0, 0)
        StopRecordTime;
}

void PII_Mult00_6x6_6x1(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm mov  edx, DWORD PTR src1
        __asm mov  ecx, DWORD PTR src2
        __asm mov  eax, DWORD PTR dst

        e6(0, 0, 1, 6, 1, 0, 0)
        e6(1, 0, 1, 6, 1, 0, 0)
        e6(2, 0, 1, 6, 1, 0, 0)
        e6(3, 0, 1, 6, 1, 0, 0)
        e6(4, 0, 1, 6, 1, 0, 0)
        e6(5, 0, 1, 6, 1, 0, 0)

        StopRecordTime;
}


void PII_Mult00_3x3_3x1(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm {
                mov     edx, dword ptr src1
                mov     ecx, dword ptr src2
                mov     eax, dword ptr dst
                fld     dword ptr [ecx]
                fmul    dword ptr [edx+24]
                fld     dword ptr [ecx]
                fmul    dword ptr [edx+12]
                fld     dword ptr [ecx]
                fmul    dword ptr [edx]
                fld     dword ptr [ecx+4]
                fmul    dword ptr [edx+4]
                fld     dword ptr [ecx+4]
                fmul    dword ptr [edx+16]
                fld     dword ptr [ecx+4]
                fmul    dword ptr [edx+28]
                fxch    ST(2)
                faddp   ST(3),ST
                faddp   ST(3),ST
                faddp   ST(3),ST
                fld     dword ptr [ecx+8]
                fmul    dword ptr [edx+8]
                fld     dword ptr [ecx+8]
                fmul    dword ptr [edx+20]
                fld     dword ptr [ecx+8]
                fmul    dword ptr [edx+32]
                fxch    ST(2)
                faddp   ST(3),ST
                faddp   ST(3),ST
                faddp   ST(3),ST
                fstp    dword ptr [eax]
                fstp    dword ptr [eax+4]
                fstp    dword ptr [eax+8]
        }
        StopRecordTime;
}


__declspec(naked) void PIII_Mult00_3x3_3x3(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm {
                mov     ecx,  dword ptr [esp+8]        ; src2
                mov     edx,  dword ptr [esp+4]        ; src1
                mov     eax,  dword ptr [esp+0Ch]      ; dst

                movss   xmm2, dword ptr [ecx+32]
                movhps  xmm2, qword ptr [ecx+24]

                movss   xmm3, dword ptr [edx]
                movss   xmm4, dword ptr [edx+4]

                movss   xmm0, dword ptr [ecx]
                movhps  xmm0, qword ptr [ecx+4]
                shufps  xmm2, xmm2, 0x36
                shufps  xmm3, xmm3, 0

                movss   xmm1, dword ptr [ecx+12]
                movhps  xmm1, qword ptr [ecx+16]

                shufps  xmm4, xmm4, 0
                mulps   xmm3, xmm0
                movss   xmm5, dword ptr [edx+8]
                movss   xmm6, dword ptr [edx+12]
                mulps   xmm4, xmm1
                shufps  xmm5, xmm5, 0
                mulps   xmm5, xmm2
                shufps  xmm6, xmm6, 0
                mulps   xmm6, xmm0
                addps   xmm3, xmm4

                movss   xmm7, dword ptr [edx+16]
                movss   xmm4, dword ptr [edx+28]

                shufps  xmm7, xmm7, 0
                addps   xmm3, xmm5
                mulps   xmm7, xmm1

                shufps  xmm4, xmm4, 0

                movss   xmm5, dword ptr [edx+20]
                shufps  xmm5, xmm5, 0
                mulps   xmm4, xmm1

                mulps   xmm5, xmm2
                addps   xmm6, xmm7

                movss   xmm1, dword ptr [edx+24]

                movss   dword ptr [eax]  , xmm3
                movhps  qword ptr [eax+4], xmm3

                addps   xmm6, xmm5
                shufps  xmm1, xmm1, 0

                movss   xmm5, dword ptr [edx+32]
                mulps   xmm1, xmm0
                shufps  xmm5, xmm5, 0

                movss   dword ptr [eax+12], xmm6
                mulps   xmm5, xmm2
                addps   xmm1, xmm4
                movhps  qword ptr [eax+16], xmm6
                addps   xmm1, xmm5
                shufps  xmm1, xmm1, 0x8F

                movhps  qword ptr [eax+24], xmm1
                movss   dword ptr [eax+32], xmm1
        }
        StopRecordTime;
        __asm ret
}
```

```
__declspec(naked) void PIII_Mult00_4x4_4x4(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm {
                mov      edx, dword ptr [esp+4]   ; src1
                mov      eax, dword ptr [esp+0Ch] ; dst
                mov      ecx, dword ptr [esp+8]   ; src2
                movss    xmm0, dword ptr [edx]
                movaps   xmm1, xmmword ptr [ecx]
                shufps   xmm0, xmm0, 0
                movss    xmm2, dword ptr [edx+4]
                mulps    xmm0, xmm1
                shufps   xmm2, xmm2, 0
                movaps   xmm3, xmmword ptr [ecx+10h]
                movss    xmm7, dword ptr [edx+8]
                mulps    xmm2, xmm3
                shufps   xmm7, xmm7, 0
                addps    xmm0, xmm2
                movaps   xmm4, xmmword ptr [ecx+20h]
                movss    xmm2, dword ptr [edx+0Ch]
                mulps    xmm7, xmm4
                shufps   xmm2, xmm2, 0
                addps    xmm0, xmm7
                movaps   xmm5, xmmword ptr [ecx+30h]
                movss    xmm6, dword ptr [edx+10h]
                mulps    xmm2, xmm5
                movss    xmm7, dword ptr [edx+14h]
                shufps   xmm6, xmm6, 0
                addps    xmm0, xmm2
                shufps   xmm7, xmm7, 0
                movlps   qword ptr [eax], xmm0
                movhps   qword ptr [eax+8], xmm0
                mulps    xmm7, xmm3
                movss    xmm0, dword ptr [edx+18h]
                mulps    xmm6, xmm1
                shufps   xmm0, xmm0, 0
                addps    xmm6, xmm7
                mulps    xmm0, xmm4
                movss    xmm2, dword ptr [edx+24h]
                addps    xmm6, xmm0
                movss    xmm0, dword ptr [edx+1Ch]
                movss    xmm7, dword ptr [edx+20h]
                shufps   xmm0, xmm0, 0
                shufps   xmm7, xmm7, 0
                mulps    xmm0, xmm5
                mulps    xmm7, xmm1
                addps    xmm6, xmm0
                shufps   xmm2, xmm2, 0
                movlps   qword ptr [eax+10h], xmm6
                movhps   qword ptr [eax+18h], xmm6
                mulps    xmm2, xmm3
                movss    xmm6, dword ptr [edx+28h]
                addps    xmm7, xmm2
                shufps   xmm6, xmm6, 0
                movss    xmm2, dword ptr [edx+2Ch]
                mulps    xmm6, xmm4
                shufps   xmm2, xmm2, 0
                addps    xmm7, xmm6
                mulps    xmm2, xmm5
                movss    xmm0, dword ptr [edx+34h]
                addps    xmm7, xmm2
                shufps   xmm0, xmm0, 0
                movlps   qword ptr [eax+20h], xmm7
                movss    xmm2, dword ptr [edx+30h]
                movhps   qword ptr [eax+28h], xmm7
                mulps    xmm0, xmm3
                shufps   xmm2, xmm2, 0
                movss    xmm6, dword ptr [edx+38h]
                mulps    xmm2, xmm1
                shufps   xmm6, xmm6, 0
                addps    xmm2, xmm0
                mulps    xmm6, xmm4
                movss    xmm7, dword ptr [edx+3Ch]
                shufps   xmm7, xmm7, 0
                addps    xmm2, xmm6
                mulps    xmm7, xmm5
                addps    xmm2, xmm7
                movaps   xmmword ptr [eax+30h], xmm2
        }
        StopRecordTime;
        __asm ret
}


__declspec(naked) void PIII_Mult00_6x6_6x6(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm {
                mov      ecx,  dword ptr [esp+8]      ; src2
                movlps   xmm3, qword ptr [ecx+72]
                mov      edx,  dword ptr [esp+4]      ; src1

                //       Loading first 4 columns (upper 4 rows) of src2.
                movaps   xmm0, xmmword ptr [ecx]
                movlps   xmm1, qword ptr [ecx+24]
                movhps   xmm1, qword ptr [ecx+32]
                movaps   xmm2, xmmword ptr [ecx+48]
                movhps   xmm3, qword ptr [ecx+80]

                //       Calculating first 4 elements in the first row of the destination matrix.
                movss    xmm4, dword ptr [edx]
                movss    xmm5, dword ptr [edx+4]
                mov      eax,  dword ptr [esp+0Ch]    ; dst
                shufps   xmm4, xmm4, 0
                movss    xmm6, dword ptr [edx+8]
                shufps   xmm5, xmm5, 0
                movss    xmm7, dword ptr [edx+12]
                mulps    xmm4, xmm0
                shufps   xmm6, xmm6, 0
                shufps   xmm7, xmm7, 0
                mulps    xmm5, xmm1
                mulps    xmm6, xmm2
                addps    xmm5, xmm4
                mulps    xmm7, xmm3
                addps    xmm6, xmm5
                addps    xmm7, xmm6

                movaps   xmmword ptr [eax], xmm7

                //       Calculating first 4 elements in the second row of the destination matrix.
                movss    xmm4, dword ptr [edx+24]
                shufps   xmm4, xmm4, 0
                mulps    xmm4, xmm0
                movss    xmm5, dword ptr [edx+28]
                shufps   xmm5, xmm5, 0
                mulps    xmm5, xmm1
                movss    xmm6, dword ptr [edx+32]
                shufps   xmm6, xmm6, 0
                movss    xmm7, dword ptr [edx+36]

                shufps   xmm7, xmm7, 0

                mulps    xmm6, xmm2
                mulps    xmm7, xmm3

                addps    xmm7, xmm6
                addps    xmm5, xmm4
                addps    xmm7, xmm5

                //       Calculating first 4 elements in the third row of the destination matrix.
                movss    xmm4, dword ptr [edx+48]
                movss    xmm5, dword ptr [edx+52]

                movlps   qword ptr [eax+24], xmm7 ; save 2nd
                movhps   qword ptr [eax+32], xmm7 ; row

                movss    xmm6, dword ptr [edx+56]
                movss    xmm7, dword ptr [edx+60]

                shufps   xmm4, xmm4, 0
                shufps   xmm5, xmm5, 0
                shufps   xmm6, xmm6, 0
                shufps   xmm7, xmm7, 0

                mulps    xmm4, xmm0
                mulps    xmm5, xmm1
                mulps    xmm6, xmm2
                mulps    xmm7, xmm3

                addps    xmm5, xmm4
                addps    xmm7, xmm6
                addps    xmm7, xmm5

                movaps   xmmword ptr [eax+48], xmm7

                //       Calculating first 4 elements in the fourth row of the destination matrix.
                movss    xmm4, dword ptr [edx+72]
                movss    xmm5, dword ptr [edx+76]
                movss    xmm6, dword ptr [edx+80]
                movss    xmm7, dword ptr [edx+84]

                shufps   xmm4, xmm4, 0
                shufps   xmm5, xmm5, 0
                shufps   xmm6, xmm6, 0
                shufps   xmm7, xmm7, 0

                mulps    xmm4, xmm0
                mulps    xmm5, xmm1
                mulps    xmm6, xmm2
                mulps    xmm7, xmm3

                addps    xmm4, xmm5
                addps    xmm6, xmm4
                addps    xmm7, xmm6

                movlps   qword ptr [eax+72], xmm7
                movhps   qword ptr [eax+80], xmm7

                //       Calculating first 4 elements in the fifth row of the destination matrix.
                movss    xmm4, dword ptr [edx+96]
                movss    xmm5, dword ptr [edx+100]
                movss    xmm6, dword ptr [edx+104]
                movss    xmm7, dword ptr [edx+108]

                shufps   xmm4, xmm4, 0
                shufps   xmm5, xmm5, 0
                shufps   xmm6, xmm6, 0
                shufps   xmm7, xmm7, 0

                mulps    xmm4, xmm0
                mulps    xmm5, xmm1
                mulps    xmm6, xmm2
                mulps    xmm7, xmm3

                addps    xmm5, xmm4
                addps    xmm7, xmm6
                addps    xmm7, xmm5

                movaps   xmmword ptr [eax+96], xmm7

                //       Calculating first 4 elements in the sixth row of the destination matrix.
                movss    xmm4, dword ptr [edx+120]
                movss    xmm5, dword ptr [edx+124]
                movss    xmm6, dword ptr [edx+128]
                movss    xmm7, dword ptr [edx+132]

                shufps   xmm4, xmm4, 0
                shufps   xmm5, xmm5, 0
                shufps   xmm6, xmm6, 0
                shufps   xmm7, xmm7, 0

                mulps    xmm4, xmm0
                mulps    xmm5, xmm1
                mulps    xmm6, xmm2
                mulps    xmm7, xmm3

                addps    xmm4, xmm5
                addps    xmm6, xmm4
                addps    xmm7, xmm6

                movhps   qword ptr [eax+128], xmm7
                movlps   qword ptr [eax+120], xmm7

                //       Loading first 4 columns (lower 2 rows) of src2.

                movlps   xmm0, qword ptr [ecx+96]
                movhps   xmm0, qword ptr [ecx+104]
                movlps   xmm1, qword ptr [ecx+120]
                movhps   xmm1, qword ptr [ecx+128]

                //       Calculating first 4 elements in the first row of the destination matrix.
                movss    xmm2, dword ptr [edx+16]
```

```
shufps    xmm2, xmm2, 0
movss     xmm4, dword ptr [edx+40]
movss     xmm3, dword ptr [edx+20]
movss     xmm5, dword ptr [edx+44]
movaps    xmm6, xmmword ptr [eax]
movlps    xmm7, qword ptr [eax+24]
shufps    xmm3, xmm3, 0
shufps    xmm5, xmm5, 0
movhps    xmm7, qword ptr [eax+32]

shufps    xmm4, xmm4, 0
mulps     xmm5, xmm1

mulps     xmm2, xmm0
mulps     xmm3, xmm1
mulps     xmm4, xmm0
addps     xmm6, xmm2

addps     xmm7, xmm4
addps     xmm7, xmm5
addps     xmm6, xmm3

movlps    qword ptr [eax+24], xmm7
movaps    xmmword ptr [eax], xmm6
movhps    qword ptr [eax+32], xmm7

//        Calculating first 4 elements in the third row of the destination matrix.
movss     xmm2, dword ptr [edx+64]
movss     xmm4, dword ptr [edx+88]
movss     xmm5, dword ptr [edx+92]
movss     xmm3, dword ptr [edx+68]
movaps    xmm6, xmmword ptr [eax+48]
movlps    xmm7, qword ptr [eax+72]
movhps    xmm7, qword ptr [eax+80]

shufps    xmm2, xmm2, 0
shufps    xmm4, xmm4, 0
shufps    xmm5, xmm5, 0
shufps    xmm3, xmm3, 0

mulps     xmm2, xmm0
mulps     xmm4, xmm0
mulps     xmm5, xmm1
mulps     xmm3, xmm1

addps     xmm6, xmm2
addps     xmm6, xmm3
addps     xmm7, xmm4
addps     xmm7, xmm5

movlps    qword ptr [eax+72], xmm7
movaps    xmmword ptr [eax+48], xmm6
movhps    qword ptr [eax+80], xmm7

//        Calculating first 4 elements in the fifth row of the destination matrix.
movss     xmm2, dword ptr [edx+112]
movss     xmm3, dword ptr [edx+116]
movaps    xmm6, xmmword ptr [eax+96]

shufps    xmm2, xmm2, 0
shufps    xmm3, xmm3, 0

mulps     xmm2, xmm0
mulps     xmm3, xmm1

addps     xmm6, xmm2
addps     xmm6, xmm3

movaps    xmmword ptr [eax+96], xmm6

//        Calculating first 4 elements in the sixth row of the destination matrix.
movss     xmm4, dword ptr [edx+136]
movss     xmm5, dword ptr [edx+140]
movhps    xmm7, qword ptr [eax+128]
movlps    xmm7, qword ptr [eax+120]

shufps    xmm4, xmm4, 0
shufps    xmm5, xmm5, 0

mulps     xmm4, xmm0
mulps     xmm5, xmm1
```

```
addps   xmm7, xmm4
addps   xmm7, xmm5

//      Calculating last 2 columns of the destination matrix.

movlps  xmm0, qword ptr [ecx+16]
movhps  xmm0, qword ptr [ecx+40]

movhps  qword ptr [eax+128], xmm7
movlps  qword ptr [eax+120], xmm7

movlps  xmm2, qword ptr [ecx+64]
movhps  xmm2, qword ptr [ecx+88]

movaps  xmm3, xmm2
shufps  xmm3, xmm3, 4Eh

movlps  xmm4, qword ptr [ecx+112]
movhps  xmm4, qword ptr [ecx+136]

movaps  xmm5, xmm4
shufps  xmm5, xmm5, 4Eh

movlps  xmm6, qword ptr [edx]
movhps  xmm6, qword ptr [edx+24]

movaps  xmm7, xmm6
shufps  xmm7, xmm7, 0F0h

mulps   xmm7, xmm0

shufps  xmm6, xmm6, 0A5h
movaps  xmm1, xmm0
shufps  xmm1, xmm1, 4Eh
mulps   xmm1, xmm6
addps   xmm7, xmm1

movlps  xmm6, qword ptr [edx+8]
movhps  xmm6, qword ptr [edx+32]

movaps  xmm1, xmm6
shufps  xmm1, xmm1, 0F0h
shufps  xmm6, xmm6, 0A5h

mulps   xmm1, xmm2
mulps   xmm6, xmm3
addps   xmm7, xmm1
addps   xmm7, xmm6

movhps  xmm6, qword ptr [edx+40]
movlps  xmm6, qword ptr [edx+16]

movaps  xmm1, xmm6
shufps  xmm1, xmm1, 0F0h
shufps  xmm6, xmm6, 0A5h

mulps   xmm1, xmm4
mulps   xmm6, xmm5
addps   xmm7, xmm1
addps   xmm7, xmm6

movlps  qword ptr [eax+16], xmm7
movhps  qword ptr [eax+40], xmm7

movlps  xmm6, qword ptr [edx+48]
movhps  xmm6, qword ptr [edx+72]

movaps  xmm7, xmm6
shufps  xmm7, xmm7, 0F0h

mulps   xmm7, xmm0

shufps  xmm6, xmm6, 0A5h
movaps  xmm1, xmm0
shufps  xmm1, xmm1, 4Eh
mulps   xmm1, xmm6
addps   xmm7, xmm1

movhps  xmm6, qword ptr [edx+80]
movlps  xmm6, qword ptr [edx+56]
```

```
                movaps    xmm1, xmm6
                shufps    xmm1, xmm1, 0F0h
                shufps    xmm6, xmm6, 0A5h

                mulps     xmm1, xmm2
                mulps     xmm6, xmm3
                addps     xmm7, xmm1
                addps     xmm7, xmm6

                movlps    xmm6, qword ptr [edx+64]
                movhps    xmm6, qword ptr [edx+88]

                movaps    xmm1, xmm6
                shufps    xmm1, xmm1, 0F0h
                shufps    xmm6, xmm6, 0A5h

                mulps     xmm1, xmm4
                mulps     xmm6, xmm5
                addps     xmm7, xmm1
                addps     xmm7, xmm6

                movlps    qword ptr [eax+64], xmm7
                movhps    qword ptr [eax+88], xmm7

                movlps    xmm6, qword ptr [edx+96]
                movhps    xmm6, qword ptr [edx+120]

                movaps    xmm7, xmm6
                shufps    xmm7, xmm7, 0F0h

                mulps     xmm7, xmm0

                shufps    xmm6, xmm6, 0A5h
                movaps    xmm1, xmm0
                shufps    xmm1, xmm1, 4Eh
                mulps     xmm1, xmm6
                addps     xmm7, xmm1

                movlps    xmm6, qword ptr [edx+104]
                movhps    xmm6, qword ptr [edx+128]

                movaps    xmm1, xmm6
                shufps    xmm1, xmm1, 0F0h
                shufps    xmm6, xmm6, 0A5h

                mulps     xmm1, xmm2
                mulps     xmm6, xmm3
                addps     xmm7, xmm1
                addps     xmm7, xmm6

                movlps    xmm6, qword ptr [edx+112]
                movhps    xmm6, qword ptr [edx+136]

                movaps    xmm1, xmm6
                shufps    xmm1, xmm1, 0F0h
                shufps    xmm6, xmm6, 0A5h

                mulps     xmm1, xmm4
                mulps     xmm6, xmm5
                addps     xmm7, xmm1
                addps     xmm7, xmm6

                movlps    qword ptr [eax+112], xmm7
                movhps    qword ptr [eax+136], xmm7
        }
        StopRecordTime;
        __asm ret
}


__declspec(naked) void PIII_Mult00_3x3_3x1(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm {
                mov       edx,  dword ptr [esp+4]       ; src1
                mov       ecx,  dword ptr [esp+8]       ; src2
                movss     xmm1, dword ptr [edx]
                mov       eax,  dword ptr [esp+0Ch]     ; dst
                movhps    xmm1, qword ptr [edx+4]
                movaps  xmm5, xmm1
                movss   xmm3, dword ptr [edx+12]
                movhps  xmm3, qword ptr [edx+24]
                movss   xmm4, dword ptr [ecx]
                shufps  xmm5, xmm3, 128
                movlps  xmm0, qword ptr [edx+16]
                shufps  xmm4, xmm4, 0
                movhps  xmm0, qword ptr [edx+28]
                shufps  xmm1, xmm0, 219
                movss   xmm2, dword ptr [ecx+4]
                movaps  xmm3, xmm1
                shufps  xmm1, xmm0, 129
                shufps  xmm2, xmm2, 0
                movss   xmm0, dword ptr [ecx+8]
                mulps   xmm4, xmm5
                mulps   xmm2, xmm1
                shufps  xmm0, xmm0, 0
                addps   xmm4, xmm2
                mulps   xmm0, xmm3
                addps   xmm4, xmm0
                movss   dword ptr [eax], xmm4
                movhps  qword ptr [eax+4], xmm4
        }
        StopRecordTime;
        __asm ret
}


__declspec(naked) void PIII_Mult10_3x3_3x1(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm {
                mov     ecx,  dword ptr [esp+8]      ; src2
                mov     edx,  dword ptr [esp+4]      ; src1
                mov     eax,  dword ptr [esp+0Ch]    ; dst

                movss   xmm0, dword ptr [ecx]

                movss   xmm5, dword ptr [edx]
                movhps  xmm5, qword ptr [edx+4]

                shufps  xmm0, xmm0, 0

                movss   xmm1, dword ptr [ecx+4]

                movss   xmm3, dword ptr [edx+12]
                movhps  xmm3, qword ptr [edx+16]

                shufps  xmm1, xmm1, 0

                mulps   xmm0, xmm5
                mulps   xmm1, xmm3

                movss   xmm2, dword ptr [ecx+8]
                shufps  xmm2, xmm2, 0

                movss   xmm4, dword ptr [edx+24]
                movhps  xmm4, qword ptr [edx+28]

                addps   xmm0, xmm1
                mulps   xmm2, xmm4

                addps   xmm0, xmm2

                movss   dword ptr [eax], xmm0
                movhps  qword ptr [eax+4], xmm0
        }
        StopRecordTime;
        __asm ret
}

__declspec(naked) void PIII_Mult00_4x4_4x1(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm {
                mov     ecx,  dword ptr [esp+ 8]      ; src2
                mov     edx,  dword ptr [esp+ 4]      ; src1

                movlps  xmm6, qword ptr [ecx   ]
                movlps  xmm0, qword ptr [edx   ]
                shufps  xmm6, xmm6, 0x44
                movhps  xmm0, qword ptr [edx+16]
                mulps   xmm0, xmm6
                movlps  xmm7, qword ptr [ecx+ 8]

                movlps  xmm2, qword ptr [edx+ 8]
                shufps  xmm7, xmm7, 0x44
                movhps  xmm2, qword ptr [edx+24]

                mulps   xmm2, xmm7
                movlps  xmm1, qword ptr [edx+32]
                movhps  xmm1, qword ptr [edx+48]

                mulps   xmm1, xmm6
                movlps  xmm3, qword ptr [edx+40]
                addps   xmm0, xmm2
                movhps  xmm3, qword ptr [edx+56]

                mov     eax,  dword ptr [esp+12]      ; dst

                mulps   xmm3, xmm7

                movaps  xmm4, xmm0
                addps   xmm1, xmm3

                shufps  xmm4, xmm1, 0x88
                shufps  xmm0, xmm1, 0xDD

                addps   xmm0, xmm4

                movaps  xmmword ptr [eax], xmm0
        }
        StopRecordTime;
        __asm ret
}

__declspec(naked)
void PIII_Mult00_6x6_6x1(float *src1, float *src2, float *dst)
{
        StartRecordTime;

        __asm {

                mov     ebx,  dword ptr [esp+ 4]      ; src1
                mov     ecx,  dword ptr [esp+ 8]      ; src2

                movlps  xmm7, qword ptr [ecx]
                movlps  xmm6, qword ptr [ecx+8]

                shufps  xmm7, xmm7, 0x44
                shufps  xmm6, xmm6, 0x44
                movlps  xmm0, qword ptr [ebx    ]
                movhps  xmm0, qword ptr [ebx+ 24]

                mulps   xmm0, xmm7
                movlps  xmm3, qword ptr [ebx+  8]
                movhps  xmm3, qword ptr [ebx+ 32]

                mulps   xmm3, xmm6
                movlps  xmm1, qword ptr [ebx+ 48]
                movhps  xmm1, qword ptr [ebx+ 72]

                mulps   xmm1, xmm7
                movlps  xmm2, qword ptr [ebx+ 96]
                movhps  xmm2, qword ptr [ebx+120]

                mulps   xmm2, xmm7
                movlps  xmm4, qword ptr [ebx+ 56]
                movhps  xmm4, qword ptr [ebx+ 80]

                movlps  xmm5, qword ptr [ebx+104]
                movhps  xmm5, qword ptr [ebx+128]

                mulps   xmm4, xmm6
                movlps  xmm7, qword ptr [ecx+16]
                addps   xmm0, xmm3
                shufps  xmm7, xmm7, 0x44
                mulps   xmm5, xmm6

                addps   xmm1, xmm4
                movlps  xmm3, qword ptr [ebx+ 16]
                movhps    xmm3, qword ptr [ebx+ 40]

                addps     xmm2, xmm5
                movlps    xmm4, qword ptr [ebx+ 64]
                movhps    xmm4, qword ptr [ebx+ 88]

                mulps     xmm3, xmm7
                movlps    xmm5, qword ptr [ebx+112]
                movhps    xmm5, qword ptr [ebx+136]

                addps     xmm0, xmm3
                mulps     xmm4, xmm7
                mulps     xmm5, xmm7

                addps     xmm1, xmm4
                addps     xmm2, xmm5

                movaps    xmm6, xmm0
                shufps    xmm0, xmm1, 0x88
                shufps    xmm6, xmm1, 0xDD

                movaps    xmm7, xmm2
                shufps    xmm7, xmm2, 0x88
                mov      eax,  dword ptr [esp+12]       ; dst
                shufps    xmm2, xmm2, 0xDD

                addps     xmm0, xmm6
                addps     xmm2, xmm7

                movaps    xmmword ptr [eax], xmm0
                movlps    qword ptr [eax+16], xmm2
        }

        StopRecordTime;

        __asm   ret

}

__declspec(naked) void PIII_Mult10_4x4_4x1(float *src1, float *src2, float *dst)
{
        StartRecordTime;
        __asm {
                mov       ecx,  dword ptr [esp+8]       ; src2
                mov       edx,  dword ptr [esp+4]       ; src1

                movss     xmm0, dword ptr [ecx]
                mov       eax,  dword ptr [esp+0Ch]     ; dst
                shufps    xmm0, xmm0, 0

                movss     xmm1, dword ptr [ecx+4]
                mulps     xmm0, xmmword ptr [edx]
                shufps    xmm1, xmm1, 0

                movss     xmm2, dword ptr [ecx+8]
                mulps     xmm1, xmmword ptr [edx+16]
                shufps    xmm2, xmm2, 0

                movss     xmm3, dword ptr [ecx+12]
                mulps     xmm2, xmmword ptr [edx+32]
                shufps    xmm3, xmm3, 0

                addps     xmm0, xmm1

                mulps     xmm3, xmmword ptr [edx+48]

                addps     xmm2, xmm3
                addps     xmm0, xmm2

                movaps    xmmword ptr [eax], xmm0
        }
        StopRecordTime;
        __asm ret
}


int Ra;
int Ca;
int Rb;
int Cb;
int StrideA; // Stride form one row of A to the next (in bytes)
int StrideB; // Stride form one row of B to the next (in bytes)
void MatrixMult(float *MatrixA, float *MatrixB, float *MatrixO)
{
        StartRecordTime;
        __asm {
                pushad
                Matrix_of_Results_Setup:
                mov     ecx, 0 ; Counter for rows in A - Ra
                Row_of_Results_Loop:
                mov     ebx, 0 ; Counter for columns in B - Cb
                Dot_Product_Setup:
                mov     eax, 0 ; Counter for single dot product - Ca or Rb
                mov     esi, MatrixA ; Load pointer to An0
                mov     edi, MatrixB ; Load pointer to B00
                lea     edi, [edi+ebx*4] ; Adjust pointer horizontally to correct batch of 24
                xorps   xmm2, xmm2 ; zero out accumulators for pass of 24 results
                xorps   xmm3, xmm3
                xorps   xmm4, xmm4
                xorps   xmm5, xmm5
                xorps   xmm6, xmm6
                xorps   xmm7, xmm7
                Dot_Product_Loop:
                mov     edx, [esi+eax*4]
                shl     edx, 1
                cmp     edx, 0
                je      Sparse_Entry_Escape
                movss   xmm0, [esi+eax*4]
                shufps  xmm0, xmm0, 0x0
                movaps  xmm1, [edi]
                mulps   xmm1, xmm0
                addps   xmm2, xmm1
                movaps  xmm1, [edi+16]
                mulps   xmm1, xmm0
                addps   xmm3, xmm1
                movaps  xmm1, [edi+32]
                mulps   xmm1, xmm0
                addps   xmm4, xmm1
                movaps  xmm1, [edi+48]
                mulps   xmm1, xmm0
                addps   xmm5, xmm1
                movaps  xmm1, [edi+64]
                mulps   xmm1, xmm0
                addps   xmm6, xmm1
                movaps  xmm1, [edi+80]
                mulps   xmm1, xmm0
                addps   xmm7, xmm1
                Sparse_Entry_Escape:
                add     edi, StrideB ; Move down a row in B
                inc     eax
                cmp     eax, Ca ; Can compare to Ca or Rb since they must be equal
                jl      Dot_Product_Loop
                 ; End_Dot_Product_Loop
                mov     eax, MatrixO ; Load pointer to On0
                lea     eax, [eax+ebx*4] ; Adjust pointer horizontally to correct batch of 24
                movaps  [eax], xmm2 ; store to Output
                movaps  [eax+16], xmm3
                movaps  [eax+32], xmm4
                movaps  [eax+48], xmm5
                movaps  [eax+64], xmm6
                movaps  [eax+80], xmm7
                add     ebx, 24 ; Move over to next batch of 24
                cmp     ebx, Cb ; Check to see if row is complete
                jl      Dot_Product_Setup
                 ; End_Row_of_Results_Loop
                mov     eax, MatrixA
                add     eax, StrideA
                mov     MatrixA, eax
                mov     eax, MatrixO
                add     eax, StrideB
                mov     MatrixO, eax
                inc     ecx
                cmp     ecx, Ra
                jl      Row_of_Results_Loop
                 ; End_Matrix_Matrix_Multiply_Loop
                popad
        }
        StopRecordTime;
}

void PII_Inverse_4x4(float* mat)
{
        float d, di;
        di      = mat[0];
        mat[0]  = d = 1.0f / di;
        mat[4]  *= -d;
        mat[8]  *= -d;
        mat[12] *= -d;
        mat[1]  *= d;
        mat[2]  *= d;
        mat[3]  *= d;
        mat[5]  += mat[4] * mat[1] * di;
        mat[6]  += mat[4] * mat[2] * di;
        mat[7]  += mat[4] * mat[3] * di;
        mat[9]  += mat[8] * mat[1] * di;
        mat[10] += mat[8] * mat[2] * di;
        mat[11] += mat[8] * mat[3] * di;
        mat[13] += mat[12] * mat[1] * di;
        mat[14] += mat[12] * mat[2] * di;
        mat[15] += mat[12] * mat[3] * di;
        di      = mat[5];
        mat[5]  = d = 1.0f / di;
        mat[1]  *= -d;
        mat[9]  *= -d;
        mat[13] *= -d;
        mat[4]  *= d;
        mat[6]  *= d;
        mat[7]  *= d;
        mat[0]  += mat[1] * mat[4] * di;
        mat[2]  += mat[1] * mat[6] * di;
        mat[3]  += mat[1] * mat[7] * di;
        mat[8]  += mat[9] * mat[4] * di;
        mat[10] += mat[9] * mat[6] * di;
        mat[11] += mat[9] * mat[7] * di;
        mat[12] += mat[13] * mat[4] * di;
        mat[14] += mat[13] * mat[6] * di;
        mat[15] += mat[13] * mat[7] * di;
        di      = mat[10];
        mat[10] = d = 1.0f / di;
        mat[2]  *= -d;
        mat[6]  *= -d;
        mat[14] *= -d;
        mat[8]  *= d;
        mat[9]  *= d;
        mat[11] *= d;
        mat[0]  += mat[2] * mat[8] * di;
        mat[1]  += mat[2] * mat[9] * di;
        mat[3]  += mat[2] * mat[11] * di;
        mat[4]  += mat[6] * mat[8] * di;
        mat[5]  += mat[6] * mat[9] * di;
        mat[7]  += mat[6] * mat[11] * di;
        mat[12] += mat[14] * mat[8] * di;
        mat[13] += mat[14] * mat[9] * di;
        mat[15] += mat[14] * mat[11] * di;
        di      = mat[15];
        mat[15] = d = 1.0f / di;
        mat[3]  *= -d;
        mat[7]  *= -d;
        mat[11] *= -d;
        mat[12] *= d;
        mat[13] *= d;
        mat[14] *= d;
        mat[0]  += mat[3] * mat[12] * di;
        mat[1]  += mat[3] * mat[13] * di;
        mat[2]  += mat[3] * mat[14] * di;
        mat[4]  += mat[7] * mat[12] * di;
        mat[5]  += mat[7] * mat[13] * di;
        mat[6]  += mat[7] * mat[14] * di;
        mat[8]  += mat[11] * mat[12] * di;
        mat[9]  += mat[11] * mat[13] * di;
        mat[10] += mat[11] * mat[14] * di;
}

void PIII_Inverse_4x4(float* src)
{
        __m128  minor0, minor1, minor2, minor3;
        __m128  row0,   row1,   row2,   row3;
        __m128  det,    tmp1;

        tmp1    = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)(src) ), (__m64*)(src+ 4));
        row1    = _mm_loadh_pi(_mm_loadl_pi(row1, (__m64*)(src+8)), (__m64*)(src+12));

        row0    = _mm_shuffle_ps(tmp1, row1, 0x88);
```

```
row1     = _mm_shuffle_ps(row1, tmp1, 0xDD);

tmp1     = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)(src+ 2)), (__m64*)(src+ 6));
row3     = _mm_loadh_pi(_mm_loadl_pi(row3, (__m64*)(src+10)), (__m64*)(src+14));

row2     = _mm_shuffle_ps(tmp1, row3, 0x88);
row3     = _mm_shuffle_ps(row3, tmp1, 0xDD);
//       -----------------------------------------------
tmp1     = _mm_mul_ps(row2, row3);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0xB1);

minor0   = _mm_mul_ps(row1, tmp1);
minor1   = _mm_mul_ps(row0, tmp1);

tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

minor0   = _mm_sub_ps(_mm_mul_ps(row1, tmp1), minor0);
minor1   = _mm_sub_ps(_mm_mul_ps(row0, tmp1), minor1);
minor1   = _mm_shuffle_ps(minor1, minor1, 0x4E);
//       -----------------------------------------------
tmp1     = _mm_mul_ps(row1, row2);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0xB1);

minor0   = _mm_add_ps(_mm_mul_ps(row3, tmp1), minor0);
minor3   = _mm_mul_ps(row0, tmp1);

tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

minor0   = _mm_sub_ps(minor0, _mm_mul_ps(row3, tmp1));
minor3   = _mm_sub_ps(_mm_mul_ps(row0, tmp1), minor3);
minor3   = _mm_shuffle_ps(minor3, minor3, 0x4E);
//       -----------------------------------------------
tmp1     = _mm_mul_ps(_mm_shuffle_ps(row1, row1, 0x4E), row3);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0xB1);
row2     = _mm_shuffle_ps(row2, row2, 0x4E);

minor0   = _mm_add_ps(_mm_mul_ps(row2, tmp1), minor0);
minor2   = _mm_mul_ps(row0, tmp1);

tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

minor0   = _mm_sub_ps(minor0, _mm_mul_ps(row2, tmp1));
minor2   = _mm_sub_ps(_mm_mul_ps(row0, tmp1), minor2);
minor2   = _mm_shuffle_ps(minor2, minor2, 0x4E);
//       -----------------------------------------------
tmp1     = _mm_mul_ps(row0, row1);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0xB1);

minor2   = _mm_add_ps(_mm_mul_ps(row3, tmp1), minor2);
minor3   = _mm_sub_ps(_mm_mul_ps(row2, tmp1), minor3);

tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

minor2   = _mm_sub_ps(_mm_mul_ps(row3, tmp1), minor2);
minor3   = _mm_sub_ps(minor3, _mm_mul_ps(row2, tmp1));
//       -----------------------------------------------
tmp1     = _mm_mul_ps(row0, row3);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0xB1);

minor1   = _mm_sub_ps(minor1, _mm_mul_ps(row2, tmp1));
minor2   = _mm_add_ps(_mm_mul_ps(row1, tmp1), minor2);

tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

minor1   = _mm_add_ps(_mm_mul_ps(row2, tmp1), minor1);
minor2   = _mm_sub_ps(minor2, _mm_mul_ps(row1, tmp1));
//       -----------------------------------------------
tmp1     = _mm_mul_ps(row0, row2);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0xB1);

minor1   = _mm_add_ps(_mm_mul_ps(row3, tmp1), minor1);
minor3   = _mm_sub_ps(minor3, _mm_mul_ps(row1, tmp1));

tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

minor1   = _mm_sub_ps(minor1, _mm_mul_ps(row3, tmp1));
minor3   = _mm_add_ps(_mm_mul_ps(row1, tmp1), minor3);
//       -----------------------------------------------
det      = _mm_mul_ps(row0, minor0);
det      = _mm_add_ps(_mm_shuffle_ps(det, det, 0x4E), det);
det      = _mm_add_ss(_mm_shuffle_ps(det, det, 0xB1), det);
tmp1     = _mm_rcp_ss(det);
```

```
        det      = _mm_sub_ss(_mm_add_ss(tmp1, tmp1), _mm_mul_ss(det, _mm_mul_ss(tmp1, tmp1)));
        det      = _mm_shuffle_ps(det, det, 0x00);

        minor0   = _mm_mul_ps(det, minor0);
        _mm_storel_pi((__m64*)(src), minor0);
        _mm_storeh_pi((__m64*)(src+2), minor0);

        minor1   = _mm_mul_ps(det, minor1);
        _mm_storel_pi((__m64*)(src+4), minor1);
        _mm_storeh_pi((__m64*)(src+6), minor1);

        minor2   = _mm_mul_ps(det, minor2);
        _mm_storel_pi((__m64*)(src+ 8), minor2);
        _mm_storeh_pi((__m64*)(src+10), minor2);

        minor3   = _mm_mul_ps(det, minor3);
        _mm_storel_pi((__m64*)(src+12), minor3);
        _mm_storeh_pi((__m64*)(src+14), minor3);
}


void PII_Inverse_6x6(float* mat)
{
        float d, di;
        di       = mat[0];
        mat[0]   = d = 1.0f / di;
        mat[6]   *= -d;
        mat[12]  *= -d;
        mat[18]  *= -d;
        mat[24]  *= -d;
        mat[30]  *= -d;
        mat[1]   *= d;
        mat[2]   *= d;
        mat[3]   *= d;
        mat[4]   *= d;
        mat[5]   *= d;
        mat[7]   += mat[6] * mat[1] * di;
        mat[8]   += mat[6] * mat[2] * di;
        mat[9]   += mat[6] * mat[3] * di;
        mat[10]  += mat[6] * mat[4] * di;
        mat[11]  += mat[6] * mat[5] * di;
        mat[13]  += mat[12] * mat[1] * di;
        mat[14]  += mat[12] * mat[2] * di;
        mat[15]  += mat[12] * mat[3] * di;
        mat[16]  += mat[12] * mat[4] * di;
        mat[17]  += mat[12] * mat[5] * di;
        mat[19]  += mat[18] * mat[1] * di;
        mat[20]  += mat[18] * mat[2] * di;
        mat[21]  += mat[18] * mat[3] * di;
        mat[22]  += mat[18] * mat[4] * di;
        mat[23]  += mat[18] * mat[5] * di;
        mat[25]  += mat[24] * mat[1] * di;
        mat[26]  += mat[24] * mat[2] * di;
        mat[27]  += mat[24] * mat[3] * di;
        mat[28]  += mat[24] * mat[4] * di;
        mat[29]  += mat[24] * mat[5] * di;
        mat[31]  += mat[30] * mat[1] * di;
        mat[32]  += mat[30] * mat[2] * di;
        mat[33]  += mat[30] * mat[3] * di;
        mat[34]  += mat[30] * mat[4] * di;
        mat[35]  += mat[30] * mat[5] * di;
        di       = mat[7];
        mat[7]   = d = 1.0f / di;
        mat[1]   *= -d;
        mat[13]  *= -d;
        mat[19]  *= -d;
        mat[25]  *= -d;
        mat[31]  *= -d;
        mat[6]   *= d;
        mat[8]   *= d;
        mat[9]   *= d;
        mat[10]  *= d;
        mat[11]  *= d;
        mat[0]   += mat[1] * mat[6] * di;
        mat[2]   += mat[1] * mat[8] * di;
        mat[3]   += mat[1] * mat[9] * di;
        mat[4]   += mat[1] * mat[10] * di;
        mat[5]   += mat[1] * mat[11] * di;
        mat[12]  += mat[13] * mat[6] * di;
        mat[14]  += mat[13] * mat[8] * di;
```

```
mat[15]  += mat[13] * mat[9] * di;
mat[16]  += mat[13] * mat[10] * di;
mat[17]  += mat[13] * mat[11] * di;
mat[18]  += mat[19] * mat[6] * di;
mat[20]  += mat[19] * mat[8] * di;
mat[21]  += mat[19] * mat[9] * di;
mat[22]  += mat[19] * mat[10] * di;
mat[23]  += mat[19] * mat[11] * di;
mat[24]  += mat[25] * mat[6] * di;
mat[26]  += mat[25] * mat[8] * di;
mat[27]  += mat[25] * mat[9] * di;
mat[28]  += mat[25] * mat[10] * di;
mat[29]  += mat[25] * mat[11] * di;
mat[30]  += mat[31] * mat[6] * di;
mat[32]  += mat[31] * mat[8] * di;
mat[33]  += mat[31] * mat[9] * di;
mat[34]  += mat[31] * mat[10] * di;
mat[35]  += mat[31] * mat[11] * di;
di        = mat[14];
mat[14]  = d = 1.0f / di;
mat[2]   *= -d;
mat[8]   *= -d;
mat[20]  *= -d;
mat[26]  *= -d;
mat[32]  *= -d;
mat[12]  *= d;
mat[13]  *= d;
mat[15]  *= d;
mat[16]  *= d;
mat[17]  *= d;
mat[0]   += mat[2] * mat[12] * di;
mat[1]   += mat[2] * mat[13] * di;
mat[3]   += mat[2] * mat[15] * di;
mat[4]   += mat[2] * mat[16] * di;
mat[5]   += mat[2] * mat[17] * di;
mat[6]   += mat[8] * mat[12] * di;
mat[7]   += mat[8] * mat[13] * di;
mat[9]   += mat[8] * mat[15] * di;
mat[10]  += mat[8] * mat[16] * di;
mat[11]  += mat[8] * mat[17] * di;
mat[18]  += mat[20] * mat[12] * di;
mat[19]  += mat[20] * mat[13] * di;
mat[21]  += mat[20] * mat[15] * di;
mat[22]  += mat[20] * mat[16] * di;
mat[23]  += mat[20] * mat[17] * di;
mat[24]  += mat[26] * mat[12] * di;
mat[25]  += mat[26] * mat[13] * di;
mat[27]  += mat[26] * mat[15] * di;
mat[28]  += mat[26] * mat[16] * di;
mat[29]  += mat[26] * mat[17] * di;
mat[30]  += mat[32] * mat[12] * di;
mat[31]  += mat[32] * mat[13] * di;
mat[33]  += mat[32] * mat[15] * di;
mat[34]  += mat[32] * mat[16] * di;
mat[35]  += mat[32] * mat[17] * di;
di        = mat[21];
mat[21]  = d = 1.0f / di;
mat[3]   *= -d;
mat[9]   *= -d;
mat[15]  *= -d;
mat[27]  *= -d;
mat[33]  *= -d;
mat[18]  *= d;
mat[19]  *= d;
mat[20]  *= d;
mat[22]  *= d;
mat[23]  *= d;
mat[0]   += mat[3] * mat[18] * di;
mat[1]   += mat[3] * mat[19] * di;
mat[2]   += mat[3] * mat[20] * di;
mat[4]   += mat[3] * mat[22] * di;
mat[5]   += mat[3] * mat[23] * di;
mat[6]   += mat[9] * mat[18] * di;
mat[7]   += mat[9] * mat[19] * di;
mat[8]   += mat[9] * mat[20] * di;
mat[10]  += mat[9] * mat[22] * di;
mat[11]  += mat[9] * mat[23] * di;
mat[12]  += mat[15] * mat[18] * di;
mat[13]  += mat[15] * mat[19] * di;
mat[14]  += mat[15] * mat[20] * di;
mat[16]  += mat[15] * mat[22] * di;
```

```
mat[17]  += mat[15] * mat[23] * di;
mat[24]  += mat[27] * mat[18] * di;
mat[25]  += mat[27] * mat[19] * di;
mat[26]  += mat[27] * mat[20] * di;
mat[28]  += mat[27] * mat[22] * di;
mat[29]  += mat[27] * mat[23] * di;
mat[30]  += mat[33] * mat[18] * di;
mat[31]  += mat[33] * mat[19] * di;
mat[32]  += mat[33] * mat[20] * di;
mat[34]  += mat[33] * mat[22] * di;
mat[35]  += mat[33] * mat[23] * di;
di        = mat[28];
mat[28]  = d = 1.0f / di;
mat[4]   *= -d;
mat[10]  *= -d;
mat[16]  *= -d;
mat[22]  *= -d;
mat[34]  *= -d;
mat[24]  *= d;
mat[25]  *= d;
mat[26]  *= d;
mat[27]  *= d;
mat[29]  *= d;
mat[0]   += mat[4] * mat[24] * di;
mat[1]   += mat[4] * mat[25] * di;
mat[2]   += mat[4] * mat[26] * di;
mat[3]   += mat[4] * mat[27] * di;
mat[5]   += mat[4] * mat[29] * di;
mat[6]   += mat[10] * mat[24] * di;
mat[7]   += mat[10] * mat[25] * di;
mat[8]   += mat[10] * mat[26] * di;
mat[9]   += mat[10] * mat[27] * di;
mat[11]  += mat[10] * mat[29] * di;
mat[12]  += mat[16] * mat[24] * di;
mat[13]  += mat[16] * mat[25] * di;
mat[14]  += mat[16] * mat[26] * di;
mat[15]  += mat[16] * mat[27] * di;
mat[17]  += mat[16] * mat[29] * di;
mat[18]  += mat[22] * mat[24] * di;
mat[19]  += mat[22] * mat[25] * di;
mat[20]  += mat[22] * mat[26] * di;
mat[21]  += mat[22] * mat[27] * di;
mat[23]  += mat[22] * mat[29] * di;
mat[30]  += mat[34] * mat[24] * di;
mat[31]  += mat[34] * mat[25] * di;
mat[32]  += mat[34] * mat[26] * di;
mat[33]  += mat[34] * mat[27] * di;
mat[35]  += mat[34] * mat[29] * di;
di        = mat[35];
mat[35]  = d = 1.0f / di;
mat[5]   *= -d;
mat[11]  *= -d;
mat[17]  *= -d;
mat[23]  *= -d;
mat[29]  *= -d;
mat[30]  *= d;
mat[31]  *= d;
mat[32]  *= d;
mat[33]  *= d;
mat[34]  *= d;
mat[0]   += mat[5] * mat[30] * di;
mat[1]   += mat[5] * mat[31] * di;
mat[2]   += mat[5] * mat[32] * di;
mat[3]   += mat[5] * mat[33] * di;
mat[4]   += mat[5] * mat[34] * di;
mat[6]   += mat[11] * mat[30] * di;
mat[7]   += mat[11] * mat[31] * di;
mat[8]   += mat[11] * mat[32] * di;
mat[9]   += mat[11] * mat[33] * di;
mat[10]  += mat[11] * mat[34] * di;
mat[12]  += mat[17] * mat[30] * di;
mat[13]  += mat[17] * mat[31] * di;
mat[14]  += mat[17] * mat[32] * di;
mat[15]  += mat[17] * mat[33] * di;
mat[16]  += mat[17] * mat[34] * di;
mat[18]  += mat[23] * mat[30] * di;
mat[19]  += mat[23] * mat[31] * di;
mat[20]  += mat[23] * mat[32] * di;
mat[21]  += mat[23] * mat[33] * di;
mat[22]  += mat[23] * mat[34] * di;
mat[24]  += mat[29] * mat[30] * di;
```

```
        mat[25]  += mat[29] * mat[31] * di;
        mat[26]  += mat[29] * mat[32] * di;
        mat[27]  += mat[29] * mat[33] * di;
        mat[28]  += mat[29] * mat[34] * di;
}


void PIII_InverseG_6x6(float *src)
{

#define EPSILON                  1e-8
#define REAL_ZERO(x)             (fabs(x) < EPSILON ? 1:0)

        __m128  minor0, minor1, minor2, minor3;
        __m128  det, tmp1, tmp2, tmp3, mask, index;
        __m128  b[6];
        __m128  row[6];

        static const unsigned long          minus_hex        = 0x80000000;
        static const __m128                 minus    = _mm_set_ps1(*(float*)&minus_hex);
        static const __m128                 e        = _mm_set_ps(1.0f, 0.0f, 0.0f, 1.0f);
        static const __m128                 epsilon  = _mm_set_ss(EPSILON);

        float   max, f;

        int i, j, n1, n2, k, mask1, mask2, mask3;

        // Loading matrixes: 4x2 to row[0], row[1] and 4x4 to row[2]...row[5].

        tmp1    = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)(&src[12])), (__m64*)(&src[18]));
        tmp2    = _mm_loadh_pi(_mm_loadl_pi(tmp2, (__m64*)(&src[24])), (__m64*)(&src[30]));

        row[0]  = _mm_shuffle_ps(tmp1, tmp2, 0x88);
        row[1]  = _mm_shuffle_ps(tmp1, tmp2, 0xDD);

        tmp1    = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)(&src[14])), (__m64*)(&src[20]));
        tmp2    = _mm_loadh_pi(_mm_loadl_pi(tmp2, (__m64*)(&src[26])), (__m64*)(&src[32]));

        row[2]  = _mm_shuffle_ps(tmp1, tmp2, 0x88);
        row[3]  = _mm_shuffle_ps(tmp1, tmp2, 0xDD);

        tmp1    = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)(&src[16])), (__m64*)(&src[22]));
        tmp2    = _mm_loadh_pi(_mm_loadl_pi(tmp2, (__m64*)(&src[28])), (__m64*)(&src[34]));

        row[4]  = _mm_shuffle_ps(tmp1, tmp2, 0x88);
        row[5]  = _mm_shuffle_ps(tmp1, tmp2, 0xDD);

        // Finding the max(|src[0]|, |src[1]|, ..., |src[5]|).

        tmp1    = _mm_loadh_pi(_mm_load_ss(&src[2]), (__m64*)&src[0]);
        tmp2    = _mm_loadh_pi(_mm_load_ss(&src[3]), (__m64*)&src[4]);

        tmp1    = _mm_andnot_ps(minus, tmp1);
        tmp2    = _mm_andnot_ps(minus, tmp2);

        tmp3    = _mm_max_ps(tmp1, tmp2);
        tmp3    = _mm_max_ps(tmp3, _mm_shuffle_ps(tmp3, tmp3, _MM_SHUFFLE(3, 2, 3, 2)));
        tmp3    = _mm_max_ss(tmp3, _mm_shuffle_ps(tmp3, tmp3, _MM_SHUFFLE(1, 1, 1, 1)));
        tmp3    = _mm_shuffle_ps(tmp3, tmp3, _MM_SHUFFLE(0, 0, 0, 0));

        mask1   = _mm_movemask_ps(_mm_cmpeq_ps(tmp1, tmp3));
        mask1   |= _mm_movemask_ps(_mm_cmpeq_ps(tmp2, tmp3))<<4;

        mask2   = mask1 & 0x98;
        mask2   = mask2 - (mask2 << 1);
        n1      = ((unsigned int)mask2) >> 31;

        n1      |= ((mask1 & 0x11) != 0) << 1;

        mask2   = mask1 & 0xC0;
        mask2   = mask2 - (mask2 << 1);
        n1      |= (((unsigned int)mask2) >> 29) & 4;

        if(REAL_ZERO(src[n1]))
                return;

        // The first Gauss iteration.

        tmp1    = row[n1];
        row[n1] = row[0];
        row[0]  = tmp1;
```

```
tmp2     = _mm_load_ss(&src[n1]);

src[n1]  = src[0];

f        = src[n1+6];
src[n1+6]        = src[6];
src[6]   = f;

tmp1     = _mm_rcp_ss(tmp2);
tmp2     = _mm_sub_ss(_mm_add_ss(tmp1, tmp1), _mm_mul_ss(tmp2, _mm_mul_ss(tmp1, tmp1)));

_mm_store_ss(&src[0], tmp2);

tmp2     = _mm_shuffle_ps(tmp2, tmp2, 0x00);
row[0]   = _mm_mul_ps(row[0], tmp2);

tmp1     = _mm_load_ss(&src[1]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[1]   = _mm_sub_ps(row[1], _mm_mul_ps(row[0], tmp1));

tmp1     = _mm_load_ss(&src[2]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[2]   = _mm_sub_ps(row[2], _mm_mul_ps(row[0], tmp1));

tmp1     = _mm_load_ss(&src[3]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[3]   = _mm_sub_ps(row[3], _mm_mul_ps(row[0], tmp1));

tmp1     = _mm_load_ss(&src[4]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[4]   = _mm_sub_ps(row[4], _mm_mul_ps(row[0], tmp1));

tmp1     = _mm_load_ss(&src[5]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[5]   = _mm_sub_ps(row[5], _mm_mul_ps(row[0], tmp1));

tmp3     = _mm_load_ss(&src[6]);
tmp3     = _mm_mul_ss(tmp3, tmp2);
_mm_store_ss(&src[6], tmp3);

tmp1     = _mm_load_ss(&src[1]);
tmp2     = _mm_load_ss(&src[7]);
tmp2     = _mm_sub_ss(tmp2, _mm_mul_ss(tmp1, tmp3));
_mm_store_ss(&src[7], tmp2);

tmp3     = _mm_shuffle_ps(tmp3, tmp3, 0x00);
tmp1     = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)&src[2]), (__m64*)&src[ 4]);
tmp2     = _mm_loadh_pi(_mm_loadl_pi(tmp2, (__m64*)&src[8]), (__m64*)&src[10]);

tmp2     = _mm_sub_ps(tmp2, _mm_mul_ps(tmp1, tmp3));

_mm_storel_pi((__m64*)&src[ 8], tmp2);
_mm_storeh_pi((__m64*)&src[10], tmp2);

// Finding the max(src[7], src[8], ..., src[11]).

tmp1     = _mm_loadh_pi(_mm_load_ss(&src[7]), (__m64*)&src[10]);
tmp2     = _mm_loadl_pi(tmp2, (__m64*)&src[8]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, _MM_SHUFFLE(0,3,2,2));

tmp1     = _mm_andnot_ps(minus, tmp1);
tmp2     = _mm_andnot_ps(minus, tmp2);

tmp3     = _mm_max_ps(tmp1, tmp2);
tmp3     = _mm_max_ps(tmp3, _mm_shuffle_ps(tmp1, tmp1, _MM_SHUFFLE(0,0,3,2)));
tmp3     = _mm_max_ss(tmp3, _mm_shuffle_ps(tmp3, tmp3, _MM_SHUFFLE(1,1,1,1)));
tmp3     = _mm_shuffle_ps(tmp3, tmp3, _MM_SHUFFLE(0,0,0,0));

mask1    = _mm_movemask_ps(_mm_cmpeq_ps(tmp2, tmp3));
mask2    = _mm_movemask_ps(_mm_cmpeq_ps(tmp1, tmp3));

n2       = ((mask1 & 3) | (mask2 & 7)) + 7;

if(REAL_ZERO(src[n2]))
         return;

// The second Gauss iteration.

tmp2     = _mm_load_ss(&src[n2]);
src[n2]  = src[7];
```

```
        n2       -= 6;

        tmp1     = row[n2];
        row[n2]  = row[1];
        row[1]   = tmp1;

        f        = src[n2];
        src[n2]  = src[1];
        src[1]   = f;

        //if(n2==n1) n2 = 0;
        n2       *= (n1!=n2);

        tmp1     = _mm_rcp_ss(tmp2);
        tmp2     = _mm_sub_ss(_mm_add_ss(tmp1, tmp1), _mm_mul_ss(tmp2, _mm_mul_ss(tmp1, tmp1)));

        _mm_store_ss(&src[7], tmp2);

        tmp2     = _mm_shuffle_ps(tmp2, tmp2, 0x00);
        row[1]   = _mm_mul_ps(row[1], tmp2);

        tmp1     = _mm_load_ss(&src[6]);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
        row[0]   = _mm_sub_ps(row[0], _mm_mul_ps(row[1], tmp1));

        tmp1     = _mm_load_ss(&src[8]);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
        row[2]   = _mm_sub_ps(row[2], _mm_mul_ps(row[1], tmp1));

        tmp1     = _mm_load_ss(&src[9]);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
        row[3]   = _mm_sub_ps(row[3], _mm_mul_ps(row[1], tmp1));

        tmp1     = _mm_load_ss(&src[10]);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
        row[4]   = _mm_sub_ps(row[4], _mm_mul_ps(row[1], tmp1));

        tmp1     = _mm_load_ss(&src[11]);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
        row[5]   = _mm_sub_ps(row[5], _mm_mul_ps(row[1], tmp1));

        row[0]   = _mm_xor_ps(row[0], minus);
        row[1]   = _mm_xor_ps(row[1], minus);

        // Inverting the matrix 4x4 by the Kramers method.

        row[3]   = _mm_shuffle_ps(row[3], row[3], 0x4E);
        row[5]   = _mm_shuffle_ps(row[5], row[5], 0x4E);

        tmp2     = _mm_mul_ps(row[4], row[5]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);

        minor0   = _mm_mul_ps(row[3], tmp1);
        minor1   = _mm_mul_ps(row[2], tmp1);

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor0   = _mm_sub_ps(_mm_mul_ps(row[3], tmp1), minor0);
        minor1   = _mm_sub_ps(_mm_mul_ps(row[2], tmp1), minor1);
        minor1   = _mm_shuffle_ps(minor1, minor1, 0x4E);
//      -----------------------------------------------
        tmp2     = _mm_mul_ps(row[3], row[4]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);

        minor0   = _mm_add_ps(_mm_mul_ps(row[5], tmp1), minor0);
        minor3   = _mm_mul_ps(row[2], tmp1);

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor0   = _mm_sub_ps(minor0, _mm_mul_ps(row[5], tmp1));
        minor3   = _mm_sub_ps(_mm_mul_ps(row[2], tmp1), minor3);
        minor3   = _mm_shuffle_ps(minor3, minor3, 0x4E);
//      -----------------------------------------------
        tmp2     = _mm_mul_ps(_mm_shuffle_ps(row[3], row[3], 0x4E), row[5]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);
        row[4]   = _mm_shuffle_ps(row[4], row[4], 0x4E);

        minor0   = _mm_add_ps(_mm_mul_ps(row[4], tmp1), minor0);
        minor2   = _mm_mul_ps(row[2], tmp1);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor0   = _mm_sub_ps(minor0, _mm_mul_ps(row[4], tmp1));
        minor2   = _mm_sub_ps(_mm_mul_ps(row[2], tmp1), minor2);
        minor2   = _mm_shuffle_ps(minor2, minor2, 0x4E);
//      -----------------------------------------------
        tmp2     = _mm_mul_ps(row[2], row[3]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);

        minor2   = _mm_add_ps(_mm_mul_ps(row[5], tmp1), minor2);
        minor3   = _mm_sub_ps(_mm_mul_ps(row[4], tmp1), minor3);

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor2   = _mm_sub_ps(_mm_mul_ps(row[5], tmp1), minor2);
        minor3   = _mm_sub_ps(minor3, _mm_mul_ps(row[4], tmp1));
//      -----------------------------------------------
        tmp2     = _mm_mul_ps(row[2], row[5]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);

        minor1   = _mm_sub_ps(minor1, _mm_mul_ps(row[4], tmp1));
        minor2   = _mm_add_ps(_mm_mul_ps(row[3], tmp1), minor2);

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor1   = _mm_add_ps(_mm_mul_ps(row[4], tmp1), minor1);
        minor2   = _mm_sub_ps(minor2, _mm_mul_ps(row[3], tmp1));
//      -----------------------------------------------
        tmp2     = _mm_mul_ps(row[2], row[4]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);

        minor1   = _mm_add_ps(_mm_mul_ps(row[5], tmp1), minor1);
        minor3   = _mm_sub_ps(minor3, _mm_mul_ps(row[3], tmp1));

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor1   = _mm_sub_ps(minor1, _mm_mul_ps(row[5], tmp1));
        minor3   = _mm_add_ps(_mm_mul_ps(row[3], tmp1), minor3);
//      -----------------------------------------------
        det      = _mm_mul_ps(row[2], minor0);
        det      = _mm_add_ps(_mm_shuffle_ps(det, det, 0x4E), det);
        det      = _mm_add_ss(_mm_shuffle_ps(det, det, 0xB1), det);

        if(_mm_movemask_ps(_mm_cmplt_ss(_mm_andnot_ps(minus, det), epsilon)) & 1)
                 return;

        tmp1     = _mm_rcp_ss(det);
        det      = _mm_sub_ss(_mm_add_ss(tmp1, tmp1), _mm_mul_ss(det, _mm_mul_ss(tmp1, tmp1)));
        det      = _mm_shuffle_ps(det, det, 0x00);

        row[2]   = _mm_mul_ps(det, minor0);
        row[3]   = _mm_mul_ps(det, minor1);

        ////////////////////////////////////////

        b[0]     = _mm_unpacklo_ps(row[0], row[1]);
        b[2]     = _mm_unpackhi_ps(row[0], row[1]);
        row[4]   = _mm_mul_ps(det, minor2);
        b[1]     = _mm_shuffle_ps(b[0], b[2], 0x4E);
        row[5]   = _mm_mul_ps(det, minor3);
        b[3]     = _mm_shuffle_ps(b[2], b[0], 0x4E);

        tmp1     = _mm_shuffle_ps(row[2], row[3], 0x50);
        tmp2     = _mm_mul_ps(b[0], tmp1);

        tmp1     = _mm_shuffle_ps(row[2], row[3], 0xA5);
        tmp2     = _mm_add_ps(tmp2, _mm_mul_ps(b[1], tmp1));

        tmp1     = _mm_shuffle_ps(row[2], row[3], 0xFA);
        tmp2     = _mm_add_ps(tmp2, _mm_mul_ps(b[2], tmp1));

        tmp1     = _mm_shuffle_ps(row[2], row[3], 0x0F);
        row[0]   = _mm_add_ps(tmp2, _mm_mul_ps(b[3], tmp1));

        tmp1     = _mm_shuffle_ps(row[4], row[5], 0x50);
        tmp2     = _mm_mul_ps(b[0], tmp1);

        tmp1     = _mm_shuffle_ps(row[4], row[5], 0xA5);
        tmp2     = _mm_add_ps(tmp2, _mm_mul_ps(b[1], tmp1));

        tmp1     = _mm_shuffle_ps(row[4], row[5], 0xFA);
        tmp2     = _mm_add_ps(tmp2, _mm_mul_ps(b[2], tmp1));
```

```
tmp1    = _mm_shuffle_ps(row[4], row[5], 0x0F);
row[1]  = _mm_add_ps(tmp2, _mm_mul_ps(b[3], tmp1));

b[2]    = _mm_shuffle_ps(row[0], row[0], 0x44);
b[3]    = _mm_shuffle_ps(row[0], row[0], 0xEE);
b[4]    = _mm_shuffle_ps(row[1], row[1], 0x44);
b[5]    = _mm_shuffle_ps(row[1], row[1], 0xEE);

// Calculating row number n2

tmp1    = _mm_load_ss(&src[8]);
tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

b  [1]  = _mm_sub_ps(_mm_shuffle_ps(e, e, 0x4E), _mm_mul_ps(b[2], tmp1));
row[1]  = _mm_xor_ps(_mm_mul_ps(row[2], tmp1), minus);

tmp1    = _mm_load_ss(&src[9]);
tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

b  [1]  = _mm_sub_ps(b  [1], _mm_mul_ps(b  [3], tmp1));
row[1]  = _mm_sub_ps(row[1], _mm_mul_ps(row[3], tmp1));

tmp1    = _mm_load_ss(&src[10]);
tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

b  [1]  = _mm_sub_ps(b  [1], _mm_mul_ps(b  [4], tmp1));
row[1]  = _mm_sub_ps(row[1], _mm_mul_ps(row[4], tmp1));

tmp1    = _mm_load_ss(&src[11]);
tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

b  [1]  = _mm_sub_ps(b  [1], _mm_mul_ps(b  [5], tmp1));
row[1]  = _mm_sub_ps(row[1], _mm_mul_ps(row[5], tmp1));

tmp1    = _mm_load_ss(&src[6]);
tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

b  [1]  = _mm_sub_ps(b[1], _mm_mul_ps(e, tmp1));

tmp2    = _mm_load_ss(&src[7]);
tmp2    = _mm_shuffle_ps(tmp2, tmp2, 0x00);

b  [1]  = _mm_mul_ps(b  [1], tmp2);
row[1]  = _mm_mul_ps(row[1], tmp2);

// Calculating row number n1

tmp1    = _mm_load_ss(&src[1]);
tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

b  [0]  = _mm_sub_ps(e, _mm_mul_ps(b[1], tmp1));
row[0]  = _mm_xor_ps(_mm_mul_ps(row[1], tmp1), minus);

tmp1    = _mm_load_ss(&src[2]);
tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

b  [0]  = _mm_sub_ps(b  [0], _mm_mul_ps(b  [2], tmp1));
row[0]  = _mm_sub_ps(row[0], _mm_mul_ps(row[2], tmp1));

tmp1    = _mm_load_ss(&src[3]);
tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

b  [0]  = _mm_sub_ps(b  [0], _mm_mul_ps(b  [3], tmp1));
row[0]  = _mm_sub_ps(row[0], _mm_mul_ps(row[3], tmp1));

tmp1    = _mm_load_ss(&src[4]);
tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

b  [0]  = _mm_sub_ps(b  [0], _mm_mul_ps(b  [4], tmp1));
row[0]  = _mm_sub_ps(row[0], _mm_mul_ps(row[4], tmp1));

tmp1    = _mm_load_ss(&src[5]);
tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

b  [0]  = _mm_sub_ps(b  [0], _mm_mul_ps(b  [5], tmp1));
row[0]  = _mm_sub_ps(row[0], _mm_mul_ps(row[5], tmp1));

tmp2    = _mm_load_ss(&src[0]);
tmp2    = _mm_shuffle_ps(tmp2, tmp2, 0x00);
```

```
        b  [0]   = _mm_mul_ps(b  [0], tmp2);
        row[0]   = _mm_mul_ps(row[0], tmp2);

        n2       = (n2==0)*(n1-n2)+n2;

        tmp1     = row[ 1];       row[ 1]  = row[n2];       row[n2]  = tmp1;
        tmp2     = b  [ 1];       b  [ 1]  = b  [n2];       b  [n2]  = tmp2;

        tmp1     = row[ 0];       row[ 0]  = row[n1];       row[n1]  = tmp1;
        tmp2     = b  [ 0];       b  [ 0]  = b  [n1];       b  [n1]  = tmp2;

        _mm_storel_pi((__m64*)&src[ 0], b  [0]);
        _mm_storel_pi((__m64*)&src[ 2], row[0]);
        _mm_storeh_pi((__m64*)&src[ 4], row[0]);

        _mm_storel_pi((__m64*)&src[ 6], b  [1]);
        _mm_storel_pi((__m64*)&src[ 8], row[1]);
        _mm_storeh_pi((__m64*)&src[10], row[1]);

        _mm_storel_pi((__m64*)&src[12], b  [2]);
        _mm_storel_pi((__m64*)&src[14], row[2]);
        _mm_storeh_pi((__m64*)&src[16], row[2]);

        _mm_storel_pi((__m64*)&src[18], b  [3]);
        _mm_storel_pi((__m64*)&src[20], row[3]);
        _mm_storeh_pi((__m64*)&src[22], row[3]);

        _mm_storel_pi((__m64*)&src[24], b  [4]);
        _mm_storel_pi((__m64*)&src[26], row[4]);
        _mm_storeh_pi((__m64*)&src[28], row[4]);

        _mm_storel_pi((__m64*)&src[30], b  [5]);
        _mm_storel_pi((__m64*)&src[32], row[5]);
        _mm_storeh_pi((__m64*)&src[34], row[5]);

#undef  EPSILON
#undef  REAL_ZERO

}  // PIII_InverseG_6x6


void PIII_InverseS_6x6(float *src)
{

#define EPSILON                  1e-8
#define REAL_ZERO(x)     (fabs(x) < EPSILON ? 1:0)

        __m128   minor0, minor1, minor2, minor3;
        __m128   det, tmp1, tmp2;
        __m128   b0, b1, b2, b3;
        __m128   row[6];

        static const unsigned long minus_hex       = 0x80000000;
        static const __m128       minus            = _mm_set_ps1(*(float*)&minus_hex);
        static const __m128       zero             = _mm_setzero_ps();
        static const __m128       e                = _mm_set_ps(1.0f,0.0f,0.0f,1.0f);
        static const __m128       epsilon          = _mm_set_ss(EPSILON);
        static const __m128       epsilon1         = _mm_set_ss(-EPSILON);

        // Loading matrixes: 4x2 to row[0], row[1] and 4x4 to row[2]...row[5].

        tmp1     = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)(&src[12])), (__m64*)(&src[18]));
        tmp2     = _mm_loadh_pi(_mm_loadl_pi(tmp2, (__m64*)(&src[24])), (__m64*)(&src[30]));

        row[0]   = _mm_shuffle_ps(tmp1, tmp2, 0x88);
        row[1]   = _mm_shuffle_ps(tmp1, tmp2, 0xDD);

        tmp1     = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)(&src[14])), (__m64*)(&src[20]));
        tmp2     = _mm_loadh_pi(_mm_loadl_pi(tmp2, (__m64*)(&src[26])), (__m64*)(&src[32]));

        row[2]   = _mm_shuffle_ps(tmp1, tmp2, 0x88);
        row[3]   = _mm_shuffle_ps(tmp1, tmp2, 0xDD);

        tmp1     = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)(&src[16])), (__m64*)(&src[22]));
        tmp2     = _mm_loadh_pi(_mm_loadl_pi(tmp2, (__m64*)(&src[28])), (__m64*)(&src[34]));

        row[4]   = _mm_shuffle_ps(tmp1, tmp2, 0x88);
        row[5]   = _mm_shuffle_ps(tmp1, tmp2, 0xDD);

        // ----------------
```

```
tmp2     = _mm_load_ss(&src[0]);
tmp1     = _mm_rcp_ss(tmp2);
tmp2     = _mm_sub_ss(_mm_add_ss(tmp1, tmp1), _mm_mul_ss(tmp2, _mm_mul_ss(tmp1, tmp1)));

_mm_store_ss(&src[0], tmp2);

tmp2     = _mm_shuffle_ps(tmp2, tmp2, 0x00);
row[0]   = _mm_mul_ps(row[0], tmp2);

tmp1     = _mm_load_ss(&src[1]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[1]   = _mm_sub_ps(row[1], _mm_mul_ps(row[0], tmp1));

tmp1     = _mm_load_ss(&src[2]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[2]   = _mm_sub_ps(row[2], _mm_mul_ps(row[0], tmp1));

tmp1     = _mm_load_ss(&src[3]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[3]   = _mm_sub_ps(row[3], _mm_mul_ps(row[0], tmp1));

tmp1     = _mm_load_ss(&src[4]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[4]   = _mm_sub_ps(row[4], _mm_mul_ps(row[0], tmp1));

tmp1     = _mm_load_ss(&src[5]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[5]   = _mm_sub_ps(row[5], _mm_mul_ps(row[0], tmp1));

b0       = _mm_load_ss(&src[6]);
b0       = _mm_mul_ss(b0, tmp2);
_mm_store_ss(&src[6], b0);

tmp1     = _mm_load_ss(&src[1]);
tmp2     = _mm_load_ss(&src[7]);
tmp2     = _mm_sub_ss(tmp2, _mm_mul_ss(tmp1, b0));
_mm_store_ss(&src[7], tmp2);

b0       = _mm_shuffle_ps(b0, b0, 0x00);
tmp1     = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)&src[2]), (__m64*)&src[ 4]);
tmp2     = _mm_loadh_pi(_mm_loadl_pi(tmp2, (__m64*)&src[8]), (__m64*)&src[10]);

tmp2     = _mm_sub_ps(tmp2, _mm_mul_ps(tmp1, b0));

_mm_storel_pi((__m64*)&src[ 8], tmp2);
_mm_storeh_pi((__m64*)&src[10], tmp2);

// ----------------

tmp2     = _mm_load_ss(&src[7]);
tmp1     = _mm_rcp_ss(tmp2);
tmp2     = _mm_sub_ss(_mm_add_ss(tmp1, tmp1), _mm_mul_ss(tmp2, _mm_mul_ss(tmp1, tmp1)));

_mm_store_ss(&src[7], tmp2);

tmp2     = _mm_shuffle_ps(tmp2, tmp2, 0x00);
row[1]   = _mm_mul_ps(row[1], tmp2);

row[0]   = _mm_sub_ps(row[0], _mm_mul_ps(row[1], b0));

tmp1     = _mm_load_ss(&src[8]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[2]   = _mm_sub_ps(row[2], _mm_mul_ps(row[1], tmp1));

tmp1     = _mm_load_ss(&src[9]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[3]   = _mm_sub_ps(row[3], _mm_mul_ps(row[1], tmp1));

tmp1     = _mm_load_ss(&src[10]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[4]   = _mm_sub_ps(row[4], _mm_mul_ps(row[1], tmp1));

tmp1     = _mm_load_ss(&src[11]);
tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);
row[5]   = _mm_sub_ps(row[5], _mm_mul_ps(row[1], tmp1));

row[0]   = _mm_xor_ps(row[0], minus);
row[1]   = _mm_xor_ps(row[1], minus);

row[3]   = _mm_shuffle_ps(row[3], row[3], 0x4E);
row[5]   = _mm_shuffle_ps(row[5], row[5], 0x4E);
```

```
        // Inverting the matrix 4x4 by the Kramers method.

        tmp2     = _mm_mul_ps(row[4], row[5]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);

        minor0   = _mm_mul_ps(row[3], tmp1);
        minor1   = _mm_mul_ps(row[2], tmp1);

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor0   = _mm_sub_ps(_mm_mul_ps(row[3], tmp1), minor0);
        minor1   = _mm_sub_ps(_mm_mul_ps(row[2], tmp1), minor1);
        minor1   = _mm_shuffle_ps(minor1, minor1, 0x4E);
//      -----------------------------------------------
        tmp2     = _mm_mul_ps(row[3], row[4]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);

        minor0   = _mm_add_ps(_mm_mul_ps(row[5], tmp1), minor0);
        minor3   = _mm_mul_ps(row[2], tmp1);

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor0   = _mm_sub_ps(minor0, _mm_mul_ps(row[5], tmp1));
        minor3   = _mm_sub_ps(_mm_mul_ps(row[2], tmp1), minor3);
        minor3   = _mm_shuffle_ps(minor3, minor3, 0x4E);
//      -----------------------------------------------
        tmp2     = _mm_mul_ps(_mm_shuffle_ps(row[3], row[3], 0x4E), row[5]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);
        row[4]   = _mm_shuffle_ps(row[4], row[4], 0x4E);

        minor0   = _mm_add_ps(_mm_mul_ps(row[4], tmp1), minor0);
        minor2   = _mm_mul_ps(row[2], tmp1);

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor0   = _mm_sub_ps(minor0, _mm_mul_ps(row[4], tmp1));
        minor2   = _mm_sub_ps(_mm_mul_ps(row[2], tmp1), minor2);
        minor2   = _mm_shuffle_ps(minor2, minor2, 0x4E);
//      -----------------------------------------------
        tmp2     = _mm_mul_ps(row[2], row[3]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);

        minor2   = _mm_add_ps(_mm_mul_ps(row[5], tmp1), minor2);
        minor3   = _mm_sub_ps(_mm_mul_ps(row[4], tmp1), minor3);

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor2   = _mm_sub_ps(_mm_mul_ps(row[5], tmp1), minor2);
        minor3   = _mm_sub_ps(minor3, _mm_mul_ps(row[4], tmp1));
//      -----------------------------------------------
        tmp2     = _mm_mul_ps(row[2], row[5]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);

        minor1   = _mm_sub_ps(minor1, _mm_mul_ps(row[4], tmp1));
        minor2   = _mm_add_ps(_mm_mul_ps(row[3], tmp1), minor2);

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor1   = _mm_add_ps(_mm_mul_ps(row[4], tmp1), minor1);
        minor2   = _mm_sub_ps(minor2, _mm_mul_ps(row[3], tmp1));
//      -----------------------------------------------
        tmp2     = _mm_mul_ps(row[2], row[4]);
        tmp1     = _mm_shuffle_ps(tmp2, tmp2, 0xB1);

        minor1   = _mm_add_ps(_mm_mul_ps(row[5], tmp1), minor1);
        minor3   = _mm_sub_ps(minor3, _mm_mul_ps(row[3], tmp1));

        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

        minor1   = _mm_sub_ps(minor1, _mm_mul_ps(row[5], tmp1));
        minor3   = _mm_add_ps(_mm_mul_ps(row[3], tmp1), minor3);
//      -----------------------------------------------
        det      = _mm_mul_ps(row[2], minor0);
        det      = _mm_add_ps(_mm_shuffle_ps(det, det, 0x4E), det);
        det      = _mm_add_ss(_mm_shuffle_ps(det, det, 0xB1), det);

        if(_mm_movemask_ps(_mm_and_ps(_mm_cmplt_ss(det, epsilon), _mm_cmpgt_ss(det, epsilon1))) & 1)
                return;

        tmp1     = _mm_rcp_ss(det);
        det      = _mm_sub_ss(_mm_add_ss(tmp1, tmp1), _mm_mul_ss(det, _mm_mul_ss(tmp1, tmp1)));
        det      = _mm_shuffle_ps(det, det, 0x00);

        row[2]   = _mm_mul_ps(det, minor0);
        row[3]   = _mm_mul_ps(det, minor1);
        row[4]   = _mm_mul_ps(det, minor2);
        row[5]   = _mm_mul_ps(det, minor3);

        b0       = _mm_unpacklo_ps(row[0], row[1]);
        b2       = _mm_unpackhi_ps(row[0], row[1]);
        b1       = _mm_shuffle_ps(b0, b2, 0x4E);
        b3       = _mm_shuffle_ps(b2, b0, 0x4E);

        tmp1     = _mm_shuffle_ps(row[2], row[3], 0x50);
        tmp2     = _mm_mul_ps(b0, tmp1);

        tmp1     = _mm_shuffle_ps(row[2], row[3], 0xA5);
        tmp2     = _mm_add_ps(tmp2, _mm_mul_ps(b1, tmp1));

        tmp1     = _mm_shuffle_ps(row[2], row[3], 0xFA);
        tmp2     = _mm_add_ps(tmp2, _mm_mul_ps(b2, tmp1));

        tmp1     = _mm_shuffle_ps(row[2], row[3], 0x0F);
        row[0]   = _mm_add_ps(tmp2, _mm_mul_ps(b3, tmp1));

        tmp1     = _mm_shuffle_ps(row[4], row[5], 0x50);
        tmp2     = _mm_mul_ps(b0, tmp1);

        tmp1     = _mm_shuffle_ps(row[4], row[5], 0xA5);
        tmp2     = _mm_add_ps(tmp2, _mm_mul_ps(b1, tmp1));

        tmp1     = _mm_shuffle_ps(row[4], row[5], 0xFA);
        tmp2     = _mm_add_ps(tmp2, _mm_mul_ps(b2, tmp1));

        tmp1     = _mm_shuffle_ps(row[4], row[5], 0x0F);
        row[1]   = _mm_add_ps(tmp2, _mm_mul_ps(b3, tmp1));

        // Calculating row number 1

        b0       = e;
        tmp1     = _mm_load_ss(&src[8]);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);

        b0       = _mm_sub_ps(b0, _mm_mul_ps(_mm_shuffle_ps(row[0], row[0], 0x4E), tmp1));
        b1       = _mm_xor_ps(_mm_mul_ps(row[2], tmp1), minus);

        tmp1     = _mm_load_ss(&src[9]);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);

        b0       = _mm_sub_ps(b0, _mm_mul_ps(row[0], tmp1));
        b1       = _mm_sub_ps(b1, _mm_mul_ps(row[3], tmp1));

        tmp1     = _mm_load_ss(&src[10]);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);

        b0       = _mm_sub_ps(b0, _mm_mul_ps(_mm_shuffle_ps(row[1], row[1], 0x4E), tmp1));
        b1       = _mm_sub_ps(b1, _mm_mul_ps(row[4], tmp1));

        tmp1     = _mm_load_ss(&src[11]);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);

        b0       = _mm_sub_ps(b0, _mm_mul_ps(row[1], tmp1));
        b1       = _mm_sub_ps(b1, _mm_mul_ps(row[5], tmp1));

        tmp1     = _mm_load_ss(&src[6]);
        tmp1     = _mm_shuffle_ps(tmp1, tmp1, 0x00);

        b0       = _mm_sub_ps(b0, _mm_mul_ps(_mm_shuffle_ps(e, e, 0x4E), tmp1));

        tmp2     = _mm_load_ss(&src[7]);
        tmp2     = _mm_shuffle_ps(tmp2, tmp2, 0x00);

        b0       = _mm_mul_ps(b0, tmp2);
        b1       = _mm_mul_ps(b1, tmp2);

        _mm_storeh_pi((__m64*)&src[ 6], b0);
        _mm_storel_pi((__m64*)&src[ 8], b1);
        _mm_storeh_pi((__m64*)&src[10], b1);

        // Calculating row number 0

        tmp1     = _mm_load_ss(&src[1]);
```

```
        tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

        b0      = _mm_sub_ps(e, _mm_mul_ps(_mm_shuffle_ps(b0, b0, 0x4E), tmp1));
        b1      = _mm_xor_ps(_mm_mul_ps(b1, tmp1), minus);

        tmp1    = _mm_load_ss(&src[2]);
        tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

        b0      = _mm_sub_ps(b0, _mm_mul_ps(row[0], tmp1));
        b1      = _mm_sub_ps(b1, _mm_mul_ps(row[2], tmp1));

        tmp1    = _mm_load_ss(&src[3]);
        tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

        b0      = _mm_sub_ps(b0, _mm_mul_ps(_mm_shuffle_ps(row[0], row[0], 0x4E), tmp1));
        b1      = _mm_sub_ps(b1, _mm_mul_ps(row[3], tmp1));

        tmp1    = _mm_load_ss(&src[4]);
        tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

        b0      = _mm_sub_ps(b0, _mm_mul_ps(row[1], tmp1));
        b1      = _mm_sub_ps(b1, _mm_mul_ps(row[4], tmp1));

        tmp1    = _mm_load_ss(&src[5]);
        tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x00);

        b0      = _mm_sub_ps(b0, _mm_mul_ps(_mm_shuffle_ps(row[1], row[1], 0x4E), tmp1));
        b1      = _mm_sub_ps(b1, _mm_mul_ps(row[5], tmp1));

        tmp2    = _mm_load_ss(&src[0]);
        tmp2    = _mm_shuffle_ps(tmp2, tmp2, 0x00);

        b0      = _mm_mul_ps(b0, tmp2);
        b1      = _mm_mul_ps(b1, tmp2);

        _mm_storel_pi((__m64*)&src[0], b0);
        _mm_storel_pi((__m64*)&src[2], b1);
        _mm_storeh_pi((__m64*)&src[4], b1);

        _mm_storel_pi((__m64*)&src[12], row[0]);
        _mm_storel_pi((__m64*)&src[14], row[2]);
        _mm_storeh_pi((__m64*)&src[16], row[2]);

        _mm_storeh_pi((__m64*)&src[18], row[0]);
        _mm_storel_pi((__m64*)&src[20], row[3]);
        _mm_storeh_pi((__m64*)&src[22], row[3]);

        _mm_storel_pi((__m64*)&src[24], row[1]);
        _mm_storel_pi((__m64*)&src[26], row[4]);
        _mm_storeh_pi((__m64*)&src[28], row[4]);

        _mm_storeh_pi((__m64*)&src[30], row[1]);
        _mm_storel_pi((__m64*)&src[32], row[5]);
        _mm_storeh_pi((__m64*)&src[34], row[5]);

} // PIII_InverseS_6x6


char* testname;
_MM_ALIGN16 float m31[] = {       11,12,13,
                                  21,22,23,
                                  31,32,33};
_MM_ALIGN16 float m32[] = {       1,2,3,
                                  0,1,2,
                                  -1,0,1};
_MM_ALIGN16 float m33[] = {-2,34,70, -2,64,130, -2,94,190};

_MM_ALIGN16 float v4[] = {0, 1, 2, 3};

_MM_ALIGN16 float m41[] = {       11,12,13,14,
                                  21,22,23,24,
                                  31,32,33,34,
                                  41,42,43,44};
_MM_ALIGN16 float m42[] = {       1,2,3,4,
                                  0,1,2,3,
                                  -1,0,1,2,
                                  0,1,2,3};
_MM_ALIGN16 float m43[] = {-2,48,98,148,-2,88,178,268,-2,128,258,388,-2,168,338,508};
_MM_ALIGN16 float m44[16];
_MM_ALIGN16 float m45[16];
_MM_ALIGN16 float m46[16];
```

```
_MM_ALIGN16 float m61[] = {       11,12,13,14,15,16,
                                  21,22,23,24,25,26,
                                  31,32,33,34,35,36,
                                  41,42,43,44,45,46,
                                  51,52,53,54,55,56,
                                  61,62,63,64,65,66};

_MM_ALIGN16 float m62[] = {       1,2,3,4,5,6,
                                  0,1,2,3,4,5,
                                  -1,0,1,2,3,4,
                                  0,1,2,3,4,5,
                                  1,2,3,4,5,6,
                                  2,3,4,5,6,7};
_MM_ALIGN16 float m63[36];
_MM_ALIGN16 float m64[36];
_MM_ALIGN16 float m65[36];
_MM_ALIGN16 float m66[36];

void testbase() {
        SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL);
        StartRecordTime;
        StopRecordTime;
        ticks[0] = end - start;
        StartRecordTime;
        StopRecordTime;
        ticks[1] = end - start;
        StartRecordTime;
        StopRecordTime;
        ticks[2] = end - start;
        StartRecordTime;
        StopRecordTime;
        ticks[3] = end - start;
        StartRecordTime;
        StopRecordTime;
        ticks[4] = end - start;
        StartRecordTime;
        StopRecordTime;
        ticks[5] = end - start;
        StartRecordTime;
        StopRecordTime;
        ticks[6] = end - start;
        StartRecordTime;
        StopRecordTime;
        ticks[7] = end - start;
        StartRecordTime;
        StopRecordTime;
        ticks[8] = end - start;
        StartRecordTime;
        StopRecordTime;
        ticks[9] = end - start;
        SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_NORMAL);
        // report("%i %i %i %i %i %i %i %i %i %i ",
        //              (int)ticks[0], (int)ticks[1], (int)ticks[2], (int)ticks[3], (int)ticks[4],
        //              (int)ticks[5], (int)ticks[6], (int)ticks[7], (int)ticks[8], (int)ticks[9]);
        base = Duration(10);
        report("Duration for %s:\t%i", testname, base);
}

void test_3x3_3x1_PII() {
        START_MEASUREMENTS;
        PII_Mult00_3x3_3x1(m31, m32, v4);
        END_MEASUREMENTS;
}

void test_3x3_3x1_PIII() {
        START_MEASUREMENTS;
        PIII_Mult00_3x3_3x1(m31, m32, v4);
        END_MEASUREMENTS;
}

void test_3x3T_3x1_PIII() {
        START_MEASUREMENTS;
        PIII_Mult10_3x3_3x1(m31, m32, v4);
        END_MEASUREMENTS;
}

void test_4x4_4x1_PII() {
        START_MEASUREMENTS;
        PII_Mult00_4x4_4x1(m41, m42, v4);
        END_MEASUREMENTS;
```

```
}

void test_4x4_4x1_PIII() {
        START_MEASUREMENTS;
        PIII_Mult00_4x4_4x1(m41, m42, v4);
        END_MEASUREMENTS;
}

void test_4x4T_4x1_PIII() {
        START_MEASUREMENTS;
        PIII_Mult10_4x4_4x1(m41, m42, v4);
        END_MEASUREMENTS;
}

void test_3x3_3x3_PII() {
        START_MEASUREMENTS;
        PII_Mult_3x3_3x3(m31, m32, m33);
        END_MEASUREMENTS;
}

void test_3x3_3x3_PIII() {
        START_MEASUREMENTS;
        PIII_Mult00_3x3_3x3(m31, m32, m33);
        END_MEASUREMENTS;
}

void test_4x4_4x4_PII() {
        START_MEASUREMENTS;
        PII_Mult00_4x4_4x4(m41, m42, m43);
        END_MEASUREMENTS;
}

void test_4x4_4x4_PIII() {
        START_MEASUREMENTS;
        PIII_Mult00_4x4_4x4(m41, m42, m43);
        END_MEASUREMENTS;
}

void test_6x6_6x1_PII() {
        START_MEASUREMENTS;
        PII_Mult00_6x6_6x1(m61, m62, m63);
        END_MEASUREMENTS;
}

void test_6x6_6x1_PIII() {
        START_MEASUREMENTS;
        PIII_Mult00_6x6_6x1(m61, m62, m63);
        END_MEASUREMENTS;
}

void test_6x6_6x6_PII() {
        START_MEASUREMENTS;
        PII_Mult00_6x6_6x6(m61, m62, m63);
        END_MEASUREMENTS;
}

void test_6x6_6x6_PIII() {
        START_MEASUREMENTS;
        PIII_Mult00_6x6_6x6(m61, m62, m63);
        END_MEASUREMENTS;
}

void test_Inverse_4x4_PII() {
        START_MEASUREMENTS;
        StartRecordTime;
        PII_Inverse_4x4(m44);
        StopRecordTime();
        END_MEASUREMENTS;
}

void test_Inverse_4x4_PIII() {
        START_MEASUREMENTS;
        StartRecordTime;
        PIII_Inverse_4x4(m45);
        StopRecordTime();
        END_MEASUREMENTS;
}

void test_Inverse_6x6_PII() {
        START_MEASUREMENTS;
        StartRecordTime;
        PII_Inverse_6x6(m64);
        StopRecordTime();
        END_MEASUREMENTS;
}

void test_InverseG_6x6_PIII() {
        START_MEASUREMENTS;
        StartRecordTime;
        PIII_InverseG_6x6(m65);
        StopRecordTime();
        END_MEASUREMENTS;
}

void test_InverseS_6x6_PIII() {
        START_MEASUREMENTS;
        StartRecordTime;
        PIII_InverseS_6x6(m66);
        StopRecordTime();
        END_MEASUREMENTS;
}

void testMult_4x4_PIII() {
        Ra = 4;
        Ca = 4;
        Rb = 4;
        Cb = 4;
        StrideA = (((Ca+3)>>2)<<4);
        StrideB = (((Cb+3)>>2)<<4);
        START_MEASUREMENTS;
        MatrixMult(m41, m42, m43);
        END_MEASUREMENTS;
}


void main(int argc, char* argv[])
{
        // We are looking for the best value among SAMPLES to
        // eliminate cache delays and effects of cpuid variable timing.

        testname = "rdtsc base";
        testbase();

        testname = "3x3 * 3x1 (PII)";
        test_3x3_3x1_PII();

        testname = "Transpose(3x3) * 3x1 (PIII)";
        test_3x3T_3x1_PIII();

        testname = "3x3 * 3x1 (PIII)";
        test_3x3_3x1_PIII();

        testname = "4x4 * 4x1 (PII)";
        test_4x4_4x1_PII();

        testname = "Transpose(4x4) * 4x1 (PIII)";
        test_4x4T_4x1_PIII();

        testname = "4x4 * 4x1 (PIII)";
        test_4x4_4x1_PIII();

        testname = "3x3 * 3x3 (PII)";
        test_3x3_3x3_PII();

        testname = "3x3 * 3x3 (PIII)";
        test_3x3_3x3_PIII();

        testname = "4x4 * 4x4 (PII)";
        test_4x4_4x4_PII();

        testname = "4x4 * 4x4 (PIII)";
        test_4x4_4x4_PIII();

        testname = "6x6 * 6x1 (PII)";
        test_6x6_6x1_PII();

        testname = "6x6 * 6x1 (PIII)";
        test_6x6_6x1_PIII();

        testname = "6x6 * 6x6 (PII)";
        test_6x6_6x6_PII();
        testname = "6x6 * 6x6 (PIII)";
        test_6x6_6x6_PIII();

        testname = "4x4 * 4x4 (general case, PIII)";
        testMult_4x4_PIII();

        int i;

        for(i = 0 ; i <16; i++)
                m44[i] = m45[i] = (float)rand() / RAND_MAX;

        for(i = 0 ; i < 36; i++)
                m64[i] = m65[i] = m66[i] = (float)rand() / RAND_MAX;

        testname = "Inverse 4x4 special (PII)";
        test_Inverse_4x4_PII();

        testname = "Inverse 4x4 (PIII)";
        test_Inverse_4x4_PIII();

        testname = "Inverse 6x6 special (PII)";
        test_Inverse_6x6_PII();

        testname = "Inverse 6x6 generic (PIII)";
        test_InverseG_6x6_PIII();

        testname = "Inverse 6x6 special (PIII)";
        test_InverseS_6x6_PIII();

// Test inverse.
//#define       zero(x)         ( fabs(x) < 1e-4 ? 1:0 )
//
//              for( i=0; i<16; i++ )
//                      if( !zero( m44[i] - m45[i] ) )
//                              break;
//              if( i<16 )
//                      report("Test PIII_Invert_4x4 fail.");
//              else
//                      report("Test PIII_Invert_4x4 passed.");
//
//              for( i=0; i<36; i++ )
//                      if( !zero( m64[i] - m65[i] ) )
//                              break;
//              if( i<36 )
//                      report("Test PIII_InvertG_6x6 fail.");
//              else
//                      report("Test PIII_InvertG_6x6 passed.");
//
//              for( i=0; i<36; i++ )
//                      if( !zero( m64[i] - m66[i] ) )
//                              break;
//              if( i<36 )
//                      report("Test PIII_InvertS_6x6 fail.");
//              else
//                      report("Test PIII_InvertS_6x6 passed.");

// Verify multiplications:

// 74, 134, 194       for 3x3 * 3x1
// 130, 230, 330, 430 for 4x4 * 4x1
//    report("%4.1f %4.1f %4.1f %4.1f", v4[0], v4[1], v4[2], v4[3]);

//    report("%4.1f %4.1f %4.1f", m33[0], m33[1], m33[2]); // -2,34,70
//    report("%4.1f %4.1f %4.1f", m33[3], m33[4], m33[5]); // -2,64,130
//    report("%4.1f %4.1f %4.1f", m33[6], m33[7], m33[8]); // -2,94,190

//    report("%4.1f %4.1f %4.1f %4.1f", m43[0], m43[1], m43[2], m43[3]);  // -2,48,98,148
//    report("%4.1f %4.1f %4.1f %4.1f", m43[4], m43[5], m43[6], m43[7]);  // -2,88,178,268
//    report("%4.1f %4.1f %4.1f %4.1f", m43[8], m43[9], m43[10],m43[11]); // -2,128,258,388
//    report("%4.1f %4.1f %4.1f %4.1f", m43[12],m43[13],m43[14],m43[15]); // -2,168,338,508

//    report("%4.1f %4.1f %4.1f %4.1f %4.1f %4.1f",
//        m63[0], m63[1], m63[2] ,m63[3] ,m63[4] ,m63[5]); // 45,126,207,288,369,450
//    report("%4.1f %4.1f %4.1f %4.1f %4.1f %4.1f",
//        m63[6], m63[7], m63[8] ,m63[9] ,m63[10],m63[11]); // 75,216,357,498,639,780
//    report("%4.1f %4.1f %4.1f %4.1f %4.1f %4.1f",
//        m63[12],m63[13],m63[14],m63[15],m63[16],m63[17]); // 105,306,507,708,909,1110
//    report("%4.1f %4.1f %4.1f %4.1f %4.1f %4.1f",
//        m63[18],m63[19],m63[20],m63[21],m63[22],m63[23]); // 135,396,657,918,1179,1440
//    report("%4.1f %4.1f %4.1f %4.1f %4.1f %4.1f",
//        m63[24],m63[25],m63[26],m63[27],m63[28],m63[29]); // 165,486,807,1128,1449,1770
//    report("%4.1f %4.1f %4.1f %4.1f %4.1f %4.1f",
//	m63[30],m63[31],m63[32],m63[33],m63[34],m63[35]); // 195,576,957,1338,1719,2100
}
```