(Version: 2013/5/16)

Intel CPU の丸めモードの変え方まとめ

柏木 雅英 (kashi@waseda.jp)

1 はじめに

精度保証付き数値計算において、浮動小数点計算の丸めモードを変えることは大変重要である。しかし、近年の Intel CPU(または AMD などの互換 CPU) では、64bit 化や SIMD 化によって丸めモードの変え方が複雑になってきているので、まとめ資料を作ってみた。

コンパイラによっては丸めモードを変える命令が備わっていることもあるが、なるべく高速になるようできる限り Inline Assemler やそれに類する手法を用い、極力無駄を省くことを考えた。

環境としては、Windows で Visual C++、Linux 系で gcc を想定した。

2 FPU と SSE2

Intel の CPU は、double の浮動小数点計算について、数値演算コプロセッサ 8087 以来の伝統的な FPU (floating point number processing unit) と、SIMD (single instruction multiple data) 命令のための SSE2 (Streaming SIMD Extensions 2) の 2 つの演算器を持っている。SSE2 では 128bit のレジスタに double を 2 つ入れて、同時に 2 つの演算を行うことが出来る。

これらの演算器は、それぞれ丸めモードの変え方が違う。更に、 C 言語で書かれたプログラムがコンパイルされたとき (一般的には) どちらの演算器を使うようにコンパイルされるか分からない。これらのことが、丸めモードの変え方を複雑にしている。

3 32bit と 64bit

64bit モードは、AMD が Intel に先行して AMD64 として機能を追加した。従来の x86 CPU と互換性を保ったまま 64bit の命令セットを追加したものである。Intel は、元々Itanium (IA-64) という全く別のアーキテクチャの CPU で 64bit 化を推進するつもりだったが、結局 AMD に合わせて Intel64 というほぼ AMD64 と同じ互換性を保つ方式を採用することにした。

64bit モードを持つ CPU は、32bit 用、64bit 用のどちらの OS も実行することが出来、また 64bit 用の OS は 32bit 用のプログラムも動作するように工夫されている。64bit モードを持たない CPU では 64bit OS を実行することは出来ず、また 32bit OS では 64bit 用のプログラムを実行することは出来ない。

今は 32bit OS と 64bit OS が混在しており、どちらの binary も等しく出回っている状態である。丸めのモードを変えるという観点において注意すべきことを列挙しておく。

- 64bit モードを持つ CPU は、必ず SSE2 を持っている。
- 32bit のプログラムは、SSE2 を持たない CPU で実行されることを考慮して、一般的には FPU のみを使う。しかし、コンパイルオプションによっては、SSE2 を使わせることも出来る。

- 64bit のプログラムは、必ず SSE2 を持っているため、一般的には SSE2 のみを使い FPU は使わない。
- Visual C++で 64bit モードのプログラムをコンパイルする場合、Inline Assembler を使うことが出来ない。

4 丸めモードの制御レジスタ

FPU と SSE2 では丸めモードの制御レジスタが別個に存在する。

4.1 FPU Control Word

FPU の動作を制御するレジスタ。長さは 16bit であり、次のような内訳になる。

R	R	R	IC	RC(2)	PC(2)	R	R	PM	UM	OM	ZM	DM	IM

R: reserved

IC: infinity control

RC(2): rounding control (00: near, 01: down, 10: up, 11:chop)

PC(2): precision control (00: 24bit, 01: not used, 10: 53bit, 11:64bit)

PM: inexact precision mask

UM: underflow mask

OM: overflow mask

ZM: divide by 0 mask

DM: denormals mask

IM: invalid numbers mask

たくさんのフラグが詰まっているが、重要なのは RC(2) の 2bit で、

00: nearest mode。最も近い浮動小数点数に丸める。通常はこのモード。

01: down mode。$-\infty$ に向かう方向に丸める。いわゆる切り捨て。

10: up mode。$+\infty$ に向かう方向に丸める。いわゆる切り上げ。

11: chop mode。0 に向かう方向に丸める。絶対値の切り捨て。

という意味である。

4.2 MXCSR Control/Status Register

SSE2 の動作を制御するレジスタ。長さは 32bit であり、次のような内訳になる。

0(31-16)	FZ	RC(2)	PM	UM	OM	ZM	DM	IM	0	PE	UE	OE	ZE	DE	IE

0: 単に 0 になっている。

FZ: Flush to Zero mode。denormal 数を使わず 0 にする。

bit5-0: SIMD 計算中にエラーが起きたかどうかを示すフラグ。

RC(2) の 2bit の意味は FPU と同じ。

5 制御レジスタの読み書き

制御レジスタの読み書きを Inline Assembler の機能を使って行う。

5.1 FPU Control Word

```
unsigned short int mode1; // 16bit unsigned
```

のように 16bit のレジスタの値をコピーするための変数を用意する。
linux 系 OS の gcc の場合、

```
__asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
```

でレジスタの内容を `mode1` にコピーし、

```
__asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));
```

で `mode1` の内容をレジスタにコピー出来る。
windows で Visual C++の場合、

```
_asm {fnstcw mode1}
```

でレジスタの内容を `mode1` にコピーし、

```
_asm {fldcw mode1}
```

で `mode1` の内容をレジスタにコピー出来る。

5.2 MXCSR Control/Status Register

```
unsigned long int mode2; // 32bit unsigned
```

のように 32bit のレジスタの値をコピーするための変数を用意する。
linux 系 OS の gcc の場合、

```
__asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
```

でレジスタの内容を `mode2` にコピーし、

```
__asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));
```

で `mode2` の内容をレジスタにコピー出来る。
windows で Visual C++の場合、

```
_asm {stmxcsr mode2}
```

でレジスタの内容を mode2 にコピーし、

```
_asm {ldmxcsr mode2}
```

で mode2 の内容をレジスタにコピー出来る。

また、SSE2 の命令に関しては、Intrinsic 命令と呼ばれる機能があり、Inline Assembler を使わずに SSE2 の命令を埋め込むことが出来る。この機能は標準化されており、gcc, Visual C++のどちらでも同じ書き方が出来る。

```
mode2 = _mm_getcsr();
```

でレジスタの内容を mode2 にコピーし、

```
_mm_setcsr(mode2);
```

で mode2 の内容をレジスタにコピー出来る。なお、この機能を使うには、

```
#include <emmintrin.h>
```

の include が必要。

6 制御レジスタを変更し、丸めモードを変える

前節の方法を使って、丸めモードを変えて見よう。制御レジスタを読み出し、丸めモードに関係する 2bit を書き換えて、制御レジスタに書き込むことによって行う。

Linux の 32bit 環境で特別なコンパイルオプションを付けていなく、FPU のみで計算される場合を例として説明する。例えば丸めモードを down にするには、

```
__asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
mode1 &= ~0x0c00;
mode1 |= 0x0400;
__asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));
```

のようにする。最初にレジスタから読み出し、該当 2bit のみが 0 であるようなマスク (~は bit 反転) と and を取って該当 2bit を 0 にし、down モードの値と or を取って down モードにする。最後にレジスタに書き込む。実際に試してみる。

```
#include <stdio.h>

/*
 * Only for Little Endian
 */
void bit_view(void *p, int size)
{
        int i, j;
        unsigned char *p2;

        for (i=size-1; i>=0; i--) {
                p2 = (unsigned char *)p + i;
                for (j=7; j>=0; j--) {
                        if ((*p2 & (1 << j)) != 0) printf("1");
                        else printf("0");
                }
        }

        printf("\n");
}

int main()
{
        unsigned short int mode1; // 16bit unsigned
```

```
        double x = 1.;
        double y = 10.;
        double z;

        // default
        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // nearest
        __asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
        mode1 &= ~0x0c00;
        mode1 |= 0x0000;
        __asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // down
        __asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
        mode1 &= ~0x0c00;
        mode1 |= 0x0400;
        __asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // up
        __asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
        mode1 &= ~0x0c00;
        mode1 |= 0x0800;
        __asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // chop
        __asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
        mode1 &= ~0x0c00;
        mode1 |= 0x0c00;
        __asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));
}
```

これを実行すると、

```
0.10000000000000001
0011111110111001100110011001100110011001100110011001100110011010
0.10000000000000001
0011111110111001100110011001100110011001100110011001100110011010
0.099999999999999992
0011111110111001100110011001100110011001100110011001100110011001
0.10000000000000001
0011111110111001100110011001100110011001100110011001100110011010
0.099999999999999992
0011111110111001100110011001100110011001100110011001100110011001
```

となり、正しく丸めの向きが変わっていることが分かる。

Linux の 64bit の場合。

```
#include <stdio.h>

/*
 * Only for Little Endian
 */
void bit_view(void *p, int size)
{
        int i, j;
        unsigned char *p2;

        for (i=size-1; i>=0; i--) {
                p2 = (unsigned char *)p + i;
                for (j=7; j>=0; j--) {
                        if ((*p2 & (1 << j)) != 0) printf("1");
                        else printf("0");
                }
        }

        printf("\n");
}

int main()
{
        unsigned long int mode2;  // 32bit unsigned

        double x = 1.;
        double y = 10.;
        double z;

        // default
        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // nearest
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        mode2 |= 0x00000000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // down
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        mode2 |= 0x00002000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // up
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        mode2 |= 0x00004000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // chop
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
```

```
        mode2 |= 0x00006000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));
}
```

実行結果は 32bit の場合と同じ。

なお、本当に最速を目指すなら、「制御レジスタを読み出して丸めモードに関係する 2bit を書き換えた状態」を記憶しておき、丸めモードの変更は制御レジスタへの書き込みのみにするのが一番速い。ただし、これは制御レジスタの他の bit をプログラム実行中に全く書き換えないことが前提となる。

Windows の 32bit の場合。Inline Assemler の書き方が違うだけ。

```
#include <stdio.h>

/*
 * Only for Little Endian
 */
void bit_view(void *p, int size)
{
        int i, j;
        unsigned char *p2;

        for (i=size-1; i>=0; i--) {
                p2 = (unsigned char *)p + i;
                for (j=7; j>=0; j--) {
                        if ((*p2 & (1 << j)) != 0) printf("1");
                        else printf("0");
                }
        }

        printf("\n");
}

int main()
{
        unsigned short int mode1; // 16bit unsigned

        double x = 1.;
        double y = 10.;
        double z;

        // default
        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // nearest
        _asm {fnstcw mode1}
        mode1 &= ~0x0c00;
        mode1 |= 0x0000;
        _asm {fldcw mode1}

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // down
        _asm {fnstcw mode1}
        mode1 &= ~0x0c00;
        mode1 |= 0x0400;
        _asm {fldcw mode1}
```

```
        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // up
        _asm {fnstcw mode1}
        mode1 &= ~0x0c00;
        mode1 |= 0x0800;
        _asm {fldcw mode1}

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // chop
        _asm {fldcw mode1}
        mode1 &= ~0x0c00;
        mode1 |= 0x0c00;
        _asm {fldcw mode1}

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));
}
```

Windows の 64bit の場合。Inline Assemler が使えないので、Intrinsic を使う。

```
#include <stdio.h>
#include <emmintrin.h> // for _mm_getcsr, _mm_setcsr

/*
 * Only for Little Endian
 */
void bit_view(void *p, int size)
{
        int i, j;
        unsigned char *p2;

        for (i=size-1; i>=0; i--) {
                p2 = (unsigned char *)p + i;
                for (j=7; j>=0; j--) {
                        if ((*p2 & (1 << j)) != 0) printf("1");
                        else printf("0");
                }
        }

        printf("\n");
}

int main()
{
        unsigned long int mode2;  // 32bit unsigned

        double x = 1.;
        double y = 10.;
        double z;

        // default
        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // nearest
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        mode2 |= 0x00000000;
        _mm_setcsr(mode2);
```

```
        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // down
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        mode2 |= 0x00002000;
        _mm_setcsr(mode2);

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // up
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        mode2 |= 0x00004000;
        _mm_setcsr(mode2);

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));

        // chop
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        mode2 |= 0x00006000;
        _mm_setcsr(mode2);

        z = x / y;
        printf("%.17g\n", z);
        bit_view(&z, sizeof(z));
}
```

7 安全な書き方

同一のソースファイルで、異なる OS、異なるコンパイラ、32/64bit の別、コンパイラオプションの違いになるべく対応すること考える。ここで、32bit の場合は、コンパイルオプションによっては FPU と SSE2 の両方が混在して使われる可能性があるので、そのときは両方の丸めモードを変えることにする。

どのような状況でコンパイルされているかを区別するマクロは、次のものを用いた。

- `_WIN32` または`_WIN64` が定義されていれば Visual C++、そうでなければ Linux 系。
- (Visual C++の場合) `_WIN64` が定義されていれば 64bit、そうでなければ 32bit。
- (Linux 系の場合) `__x86_64__`が定義されていれば 64bit、そうでなければ
- (Visual C++の場合) `_M_IX86_FP` が 2 であれば SSE2 が有効にされている。
- (Linux 系の場合) `__SSE2_MATH__`が定義されていれば SSE2 が有効にされている。

```
#include <stdio.h>
#if defined(_WIN64) || _M_IX86_FP == 2
#include <emmintrin.h> // for _mm_getcsr, _mm_setcsr
#endif

void roundnear()
```

```
{
        unsigned short int mode1; // 16bit unsigned
        unsigned long int mode2;  // 32bit unsigned

#if defined(_WIN32) || defined(_WIN64) // Windows
  #if !defined(_WIN64)
        _asm {fnstcw mode1}
        mode1 &= ~0x0c00;
        mode1 |= 0x0000;
        _asm {fldcw mode1}
  #endif
  #if defined(_WIN64) || _M_IX86_FP == 2
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        mode2 |= 0x00000000;
        _mm_setcsr(mode2);
  #endif
#else // Linux, etc
  #if !defined(__x86_64__)
        __asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
        mode1 &= ~0x0c00;
        mode1 |= 0x0000;
        __asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));
  #endif
  #if defined(__x86_64__) || defined(__SSE2_MATH__)
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        mode2 |= 0x00000000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));
  #endif
#endif
}

void rounddown()
{
        unsigned short int mode1; // 16bit unsigned
        unsigned long int mode2;  // 32bit unsigned

#if defined(_WIN32) || defined(_WIN64) // Windows
  #if !defined(_WIN64)
        _asm {fnstcw mode1}
        mode1 &= ~0x0c00;
        mode1 |= 0x0400;
        _asm {fldcw mode1}
  #endif
  #if defined(_WIN64) || _M_IX86_FP == 2
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        mode2 |= 0x00002000;
        _mm_setcsr(mode2);
  #endif
#else // Linux, etc
  #if !defined(__x86_64__)
        __asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
        mode1 &= ~0x0c00;
        mode1 |= 0x0400;
        __asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));
  #endif
  #if defined(__x86_64__) || defined(__SSE2_MATH__)
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        mode2 |= 0x00002000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));
  #endif
#endif
}
```

```
void roundup()
{
        unsigned short int mode1; // 16bit unsigned
        unsigned long int mode2;  // 32bit unsigned

#if defined(_WIN32) || defined(_WIN64) // Windows
  #if !defined(_WIN64)
        _asm {fnstcw mode1}
        mode1 &= ~0x0c00;
        mode1 |= 0x0800;
        _asm {fldcw mode1}
  #endif
  #if defined(_WIN64) || _M_IX86_FP == 2
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        mode2 |= 0x00004000;
        _mm_setcsr(mode2);
  #endif
#else // Linux, etc
  #if !defined(__x86_64__)
        __asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
        mode1 &= ~0x0c00;
        mode1 |= 0x0800;
        __asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));
  #endif
  #if defined(__x86_64__) || defined(__SSE2_MATH__)
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        mode2 |= 0x00004000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));
  #endif
#endif
}

void roundchop()
{
        unsigned short int mode1; // 16bit unsigned
        unsigned long int mode2;  // 32bit unsigned

#if defined(_WIN32) || defined(_WIN64) // Windows
  #if !defined(_WIN64)
        _asm {fnstcw mode1}
        mode1 &= ~0x0c00;
        mode1 |= 0x0c00;
        _asm {fldcw mode1}
  #endif
  #if defined(_WIN64) || _M_IX86_FP == 2
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        mode2 |= 0x00006000;
        _mm_setcsr(mode2);
  #endif
#else // Linux, etc
  #if !defined(__x86_64__)
        __asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
        mode1 &= ~0x0c00;
        mode1 |= 0x0c00;
        __asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));
  #endif
  #if defined(__x86_64__) || defined(__SSE2_MATH__)
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        mode2 |= 0x00006000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));
  #endif
#endif
}
```

8 おまけ: コンパイルオプション

8.1 gcc

-m32 で 32bit アプリケーション、-m64 で 64bit のバイナリにコンパイル出来る。

32bit の場合、-msse2 で SSE2 が有効に。これに加えて-mfpmath=sse を付けると、浮動小数点計算に SSE2 が使われる。-mfpmath=387 で FPU、\mfpmath=sse,387 で両方が使われる。

8.2 Visual C++

64bit OS で、Visual Studio 2005(2008) x64 Win64 コマンドプロンプトを開き、そこで cl を起動すれば 64bit アプリを作れる。32bit を作るには、通常の Visual Studio 2005(2008) コマンドプロンプトで。

32bit の場合、/arch:SSE2 で SSE2 が使われるようになるが、実際に使われるかどうかはコンパイラの判断による。つまり、FPU と SSE2 の混在になる。