(Version: 2017/4/18)

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みよう。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　コンパイラが備える丸めモード変更命令

C コンパイラが C99 準拠ならば、fenv.h を include し、fesetround を使うことで丸めモードの変更が可能である。

```
#include <stdio.h>
#include <fenv.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()
{
```

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

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

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

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

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

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

コンパイル時には `-lm` など、適切なライブラリのリンクが必要な場合もある。これを使えば Intel 以外の CPU にも対応出来る。

なお、Visual C++については、Visual Studio 2012 以前は C99 に準拠しておらず `fenv.h` を持っていなかった。Visual Studio 2013 で C99 がサポートされるようになり、`fenv.h` を include すれば `fesetround` が使えるようになった。しかし、64bit モードにおいて上向き丸めと下向き丸めが逆になるという致命的なバグがあり、最新版 (2016 年 7 月現在) である Visual Studio 2013 update5 及び Visual Studio 2015 update3 においても直っていない。よって、Visual C++では `fesetround` は使い物にならない。

Visual C++では、丸めモード変更の方法として以前から `float.h` を include し `_controlfp` を使う方法を提供している。

```
#include <stdio.h>
#include <float.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()
{
        double x = 1.;
        double y = 10.;
        double z;

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

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

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

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

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

または、_controlfp_s を使う。これはセキュリティが強化されたバージョンらしいが詳細は未調査。

```
#include <stdio.h>
#include <float.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 int cw = 0;
```

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

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

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

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

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

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

これらの方法を使えば、CPU の違いや 32/64bit の違いを吸収してくれるので、大変楽である。しかし、必ず「レジスタ読み込み → 値の変更 → レジスタ書き込み」のセットになってしまうので、予め計算しておいた値を書き込むだけ、のような工夫を行うことが出来ない。また、inline 展開されないので、ライブラリ関数の呼び出しのオーバーヘッドが避けられない。

8 汎用かつ安全そうな書き方

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。

- `_MSC_VER` が定義されていれば Visual C++、そうでなければ Linux 系 (gcc 等)。
- (Visual C++の場合) `_WIN64` が定義されていれば 64bit、そうでなく `_WIN32` が定義されていれば 32bit、そうでなければ Intel CPU でないと判断。
- (Linux 系の場合) `__i386__`が定義されていれば 32bit、`__x86_64__`が定義されていれば 64bit、どちらでもなければ Intel CPU でないと判断。
- (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。

```
#if defined(_WIN32) || defined(_WIN64) // Windows
#if defined(_WIN64) // Windows 64bit

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

void roundnear()
{
        unsigned long int mode2;  // 32bit unsigned

        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        _mm_setcsr(mode2);
}

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

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

void roundup()
{
        unsigned long int mode2;  // 32bit unsigned

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

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

        mode2 = _mm_getcsr();
        mode2 |= 0x00006000;
        _mm_setcsr(mode2);
}

#elif defined(_WIN32) // Windows 32bit

#if _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

        _asm {fnstcw mode1}
        mode1 &= ~0x0c00;
        _asm {fldcw mode1}
#if _M_IX86_FP == 2
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        _mm_setcsr(mode2);
#endif
}

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

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

        _asm {fnstcw mode1}
        mode1 &= ~0x0c00;
        mode1 |= 0x0400;
        _asm {fldcw mode1}
#if _M_IX86_FP == 2
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        mode2 |= 0x00002000;
        _mm_setcsr(mode2);
#endif
}

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

        _asm {fnstcw mode1}
        mode1 &= ~0x0c00;
        mode1 |= 0x0800;
        _asm {fldcw mode1}
#if _M_IX86_FP == 2
        mode2 = _mm_getcsr();
        mode2 &= ~0x00006000;
        mode2 |= 0x00004000;
        _mm_setcsr(mode2);
#endif
}

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

        _asm {fnstcw mode1}
        mode1 |= 0x0c00;
        _asm {fldcw mode1}
#if _M_IX86_FP == 2
        mode2 = _mm_getcsr();
        mode2 |= 0x00006000;
        _mm_setcsr(mode2);
#endif
}

#else // Windows other CPU

#include <float.h>

void roundnear()
{
        unsigned int cw = 0;
        _controlfp_s(&cw, _RC_NEAR, _MCW_RC);
}

void rounddown()
{
        unsigned int cw = 0;
        _controlfp_s(&cw, _RC_DOWN, _MCW_RC);
}

void roundup()
{
        unsigned int cw = 0;
        _controlfp_s(&cw, _RC_UP, _MCW_RC);
}
```

```
void roundchop()
{
        unsigned int cw = 0;
        _controlfp_s(&cw, _RC_CHOP, _MCW_RC);
}

#endif
#else // Linux, etc
#if defined(__x86_64__) // Linux 64bit

void roundnear()
{
        unsigned long int mode2;  // 32bit unsigned

        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));
}

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

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

void roundup()
{
        unsigned long int mode2;  // 32bit unsigned

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

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

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

#elif defined(__i386__) // Linux 32bit

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

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

#if defined(__SSE2_MATH__)
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));
#endif
}
```

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

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

#if defined(__SSE2_MATH__)
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        mode2 |= 0x00002000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));
#endif
}

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

        __asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
        mode1 &= ~0x0c00;
        mode1 |= 0x0800;
        __asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));
#if defined(__SSE2_MATH__)
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 &= ~0x00006000;
        mode2 |= 0x00004000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));
#endif
}

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

        __asm__ __volatile__ ("fnstcw␣%0" : "=m"(mode1));
        mode1 |= 0x0c00;
        __asm__ __volatile__ ("fldcw␣%0" : : "m"(mode1));
#if defined(__SSE2_MATH__)
        __asm__ __volatile__ ("stmxcsr␣%0" : "=m"(mode2));
        mode2 |= 0x00006000;
        __asm__ __volatile__ ("ldmxcsr␣%0" : : "m"(mode2));
#endif
}

#else // Linux other CPU

#include <fenv.h>

void roundnear()
{
        fesetround(FE_TONEAREST);
}

void rounddown()
{
        fesetround(FE_DOWNWARD);
}

void roundup()
{
        fesetround(FE_UPWARD);
```

```
}

void roundchop()
{
        fesetround(FE_TOWARDZERO);
}

#endif
#endif
```

9 最適化について

丸めの変更を伴う計算は、コンパイラが丸めモードの変更をあまり考慮しておらず、強い最適化を用いると思わぬ結果を招くことがある。次のプログラムを見てほしい。

```
#include <stdio.h>
#include "roundingmode-universal.c"

double div_down(double x, double y)
{
        double r;

        rounddown();
        r = x / y;
        roundnear();

        return r;
}

double div_up(double x, double y)
{
        double r;

        roundup();
        r = x / y;
        roundnear();

        return r;
}

int main()
{
        double x = 1.;
        double y = 10.;
        double z;

        z = div_down(x, y);
        printf("%.17g\n", z);

        z = div_up(x, y);
        printf("%.17g\n", z);
}
```

これを手元の gcc 4.4 で試してみると、オプション無しまたは-O0 のときは、

```
0.099999999999999992
0.10000000000000001
```

と正常に動作しているが、-O1, -O2, -O3 では、

```
0.10000000000000001
0.10000000000000001
```

のように異常な計算結果となってしまう。丸め変更の持つ副作用をコンパイラが理解していないので、

- 定数部分をコンパイル時に計算する
- 同一 (に見える) 計算を一つにまとめる
- 計算順序を変更する

などの最適化が行われ、正常に動作しなくなってしまう。

この対策は、次のような方法が考えられる。

```
#include <stdio.h>
#include "roundingmode-universal.c"

double div_down(double x, double y)
{
        #pragma STDC FENV_ACCESS ON
        double r;

        rounddown();
        r = x / y;
        roundnear();

        return r;
}

double div_up(double x, double y)
{
        #pragma STDC FENV_ACCESS ON
        double r;

        roundup();
        r = x / y;
        roundnear();

        return r;
}

int main()
{
        double x = 1.;
        double y = 10.;
        double z;

        z = div_down(x, y);
        printf("%.17g\n", z);

        z = div_up(x, y);
        printf("%.17g\n", z);
}
```

C99 で策定された FENV_ACCESS プラグマを用いる方法。丸めの変更が行われる可能性があるのでそれを考慮するようコンパイラに指示する。残念ながら今のところ gcc ではこの指定は無視されてしまう。gcc 4.4, gcc 4.8, gcc 5.3 で無視されてしまうことを確認した。

現状では、次のようにするしか無さそうである。

```
#include <stdio.h>
#include "roundingmode-universal.c"

double div_down(double x, double y)
{
        volatile double r, x1 = x, y1 = y;
```

```
        rounddown();
        r = x1 / y1;
        roundnear();

        return r;
}

double div_up(double x, double y)
{
        volatile double r, x1 = x, y1 = y;

        roundup();
        r = x1 / y1;
        roundnear();

        return r;
}

int main()
{
        double x = 1.;
        double y = 10.;
        double z;

        z = div_down(x, y);
        printf("%.17g\n", z);

        z = div_up(x, y);
        printf("%.17g\n", z);
}
```

丸めが変わってほしい演算の入出力に使われる変数をいったん volatile 変数にコピーしてから丸めモードを変更し、演算を行う方法。速度の低下は避けられないが、現状ではこれが最も確実と思われる。

10 おまけ: コンパイルオプション

10.1 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。

10.2 Visual C++

64bit OS で、VS2013(2015) x64 Native Tools 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、VS2013(2015) x86 Native Tools 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。

11 異常現象

ここでは、過去に著者が遭遇した丸めに関する様々な異常現象を紹介する。

11.1 Visual C++の fesetround の丸めの向きが逆

Visual Studio 2012 以前は C99 に準拠しておらず fenv.h を持っていなかったが、Visual Studio 2013 で C99 がサポートされるようになり、fenv.h を include すれば fesetround が使えるようになった。しかし、64bit モードにおいて上向き丸めと下向き丸めが逆になるという致命的なバグがあり、最新版 (2016 年 4 月現在) である Visual Studio 2013 update5 及び Visual Studio 2015 update3 においても直っていない。(2017 年 4 月 18 日追記: Visual Studio 2017 で直ったとの情報あり。未確認。)

次のプログラムで、丸めモードがきちんと変わっているか調査できる。

```
#include <iostream>
#include <cfenv>
#include <cmath>

int check_rounding()
{
        volatile double x, y, z;

        x = 1;
        y = pow(2., -55);

        z = x + y;

        if (z > 1) return 2; // up

        z = x - y;

        if (z == 1) return 0; // nearest

        z = - x + y;

        if (z == -x) return 1; // down

        return 3; // chop
}


int main()
{
        std::cout << check_rounding() << "\n";

        fesetround(FE_TONEAREST);
        std::cout << check_rounding() << "\n";

        fesetround(FE_DOWNWARD);
        std::cout << check_rounding() << "\n";

        fesetround(FE_UPWARD);
        std::cout << check_rounding() << "\n";

        fesetround(FE_TOWARDZERO);
        std::cout << check_rounding() << "\n";
}
```

このプログラムを動かすと、IEEE754 に準拠しているなら

```
0
0
1
2
3
```

と表示されるはずであるが、Visual C++の 64bit モードでコンパイルすると

```
0
0
2
1
3
```

となってしまう。すなわち、最近点丸めとゼロ方向丸めは正常に動作するが、上向き丸めと下向き丸めが入れ替わっている。32bit モードでは問題は起きない。

11.2 Visual C++の sqrt の丸めの向きが変更されない

まだ書いてない。

32bit の Visual C++の sqrt はどうやら FPU を呼ばず独自実装らしく、丸めの向きの変更が効かない話。

また、cygwin や msys2 が 32bit の Visual C++で作られているらしく、コンパイルオプションによっては sqrt の丸めの向きの変更が効かなくなるという話。matlab2007b でも同様の現象が確認できた話。

11.3 double rounding に起因する現象

まず、次のプログラムを見て欲しい。

```
#include <stdio.h>

main()
{
        volatile double a = 5000000000000001.;
        volatile double b = 0.499755859375;
        volatile double c;

        printf("%.80g\n", a);
        printf("%.80g\n", b);

        c = a + b;

        printf("%.80g\n", c);
}
```

これを、Linux(64bit) で普通にコンパイルした場合と、-m32 を付けて 32bit でコンパイルした場合を比べてみる。

```
$ cc doubleround.c
$ ./a.out
5000000000000001
0.499755859375
5000000000000001
$ cc -m32 doubleround.c
$ ./a.out
5000000000000001
0.499755859375
5000000000000002
$
```

このように結果が異なってしまう。5000000000000001 と 5000000000000002 は隣同士の double で表現可能な数であり、後者は最近点に丸められていないという点で IEEE754 規格に反している。

これは、Intel の CPU の持つ FPU が IEEE754 の double よりも精度が高く (全長 80bit, 仮数部 64bit)、FPU で (規格よりも高い) 精度で計算し、メモリに格納するときに double に丸める、とい

う動きをするために起きている。「より高い精度で計算をしているのだから問題はないだろう」と安易にこういう実装にしたと思われるが、ときにこの実装は問題を引き起こす。“double rounding” とは、高精度での丸めと低精度での丸めの 2 段階の丸めを指している。

先の例を詳細に検討してみよう。

$$5000000000000001(10) = 1000111000011011110010011011111100000100000000000000001(2)$$

であり、

$$0.499755859375(10) = 0.011111111111(2)$$

である。当然これを加えると、正確な値は

$$1000111000011011110010011011111100000100000000000000001.011111111111(2)$$

となる。分かりやすいように

- 53 桁目と 54 桁目の間
- 64 桁目と 65 桁目の間

に | を入れて表示すると、

$$1000111000011011110010011011111100000100000000000000001.|01111111111|1(2)$$

となる。これを一気に 53 桁に丸めるなら、54 桁目は 0 なので切り捨てられて

$$1000111000011011110010011011111100000100000000000000001(2)$$

となるが、まず 64 桁に丸めて次に 53 桁に丸めようとすると、まず 65 桁目を見て 1 なので繰り上がって

$$1000111000011011110010011011111100000100000000000000001.|10000000000(2)$$

となり、次に 64 桁目を見ると 1 なので繰り上がって、

$$1000111000011011110010011011111100000100000000000000010(2)$$

となってしまう。ちょうど、1.49 をいきなり小数点以下 1 桁目で四捨五入すると 1 になるが、小数点以下 2 桁目で四捨五入してから小数点以下 1 桁目で四捨五入すると、$1.49 \to 1.5 \to 2$ のようになってしまうのと同じ現象である。

Intel CPU において double rounding が起きるのは FPU の精度が高すぎるせいであるので、一般的に FPU を使わない 64bit モードでは発生しない。また、FPU を使っていても、FPU の計算精度を 53bit に制限することが出来 (FPU Control Word の “PC” を “10” にすればよい)、その場合はこの現象は発生しない。“PC” のデフォルト値は、Visual C++では “10”、Linux では “11” のようである。

また、次のプログラムを考える。

```
#include <stdio.h>
#include <fenv.h>

main()
{
```

```
        volatile double x = -1;
        volatile double y = 10;
        volatile double z;

        fesetround(FE_UPWARD);
        z = -(x / y);
        printf("%.20g\n", z);
}
```

実行結果は 0.1 より小さい値となるはずであるが、Linux の 32bit で実行すると 0.1 より大きい値になってしまう。

```
$ cc -m64 doubleround2.c -lm
$ ./a.out
0.09999999999999991674
$ cc -m32 doubleround2.c -lm
$ ./a.out
0.10000000000000000556
```

これも double rounding による異常である。この現象の原因は分かりにくいが、次の通りである。通常は、

(1) `x / y` が上向き丸めで計算されて −0.1 より大きい (絶対値は 0.1 より小さい) 数になる。

(2) 符号反転は無誤差で行われるので、結果は 0.1 より小さい数になる。

という動作をするはずが、

(1) `x / y` が 64bit 精度で上向き丸めで計算されて −0.1 より「わずかに」大きい (絶対値は 0.1 よりわずかに小さい)64bit 数になる。

(2) 符号反転は無誤差で行われるので、結果は 0.1 よりわずかに小さい 64bit 数になる。

(3) 0.1 よりわずかに小さい 64bit 数を 53bit に上向き丸めで丸めると、0.1 より大きい数になる。

というプロセスで、異常な結果が得られてしまう。

gcc のように 64bit 精度数を明示的に `long double` として扱える処理系を用いて、途中経過を明示的に表示させてみるとよく分かる。

```
#include <stdio.h>
#include <fenv.h>

main()
{
        volatile long double x = -1;
        volatile long double y = 10;
        volatile long double z1;
        volatile double z;

        fesetround(FE_UPWARD);
        z1 = -(x / y);
        printf("%.20Lg\n", z1);
        z = z1;
        printf("%.20g\n", z);
}
```

実行すると、

```
$ cc -m64 doubleround2a.c -lm
$ ./a.out
0.09999999999999999995
0.10000000000000000556
```

```
$ cc -m32 doubleround2a.c -lm
$ ./a.out
0.09999999999999999995
0.10000000000000000556
```

となる。

11.4 Intel コンパイラと非正規化数

Intel コンパイラ (icc) では、SSE 命令を使ったとき非正規化数を強制的に 0 にしてしまう動作がデフォルトになっており、注意が必要である。これは丸めモードを無視して行われるので、例えば丸めモード上向きの状態で計算結果が $2^{-1074} \leq x < 2^{-1022}$ になったときそれが 0 にされてしまうということは、丸めモードの指定に従っていないことになる。

それを抑制するには、オプション `/Qftz-`(Windows)、`-ftz-`(Linux, Mac) を付ける必要がある。

12 独り言

速度を重視するのでなければ、基本は `fesetround`、windows で Visual C++の場合のみ `_controlfp` を使うのが現状ではお勧めできる。