Know your FPU

Michael Herf
April 5, 2000
back

Lots of people write floating-point code without knowing what can make it slow. I've been using x86 floating-point for a while, and this article should explain some of the basic tricks that I know.

First, there are two **major** bottlenecks to making code run fast on the P2/P3 chips.

Here are the top two:

1. Know how and why to turn on single precision.
2. Know how to do fast conversions to integer. Don't do it by calling your compiler's (int)x. (Yes, this means roll your own floor() and ceil() too.)

(2006) See Sree's updated notes about fp to int conversions. New: Linux/gcc-compatible code for these functions.

Precision control

First -- a tiny introduction on IEEE floating point formats. On most chips, IEEE floats come in three flavors: 32-bit, 64-bit, and 80-bit, called "single-", "double-" and "extended-" precision, respectively. The increase in precision from one to the next is better than it appears: 32-bit floats only give you 24 bits of mantissa, while 64-bit ones give you 53 bits. Extended gives you 64 bits. (Unfortunately, you can't use extended precision under Windows NT without special drivers. This is ostensibly for compatibility with other chips.) Other CPUs, like the PowerPC, have even larger formats, like a 128-bit format.

Second -- to dispel a myth. Typing "float" rather than "double" changes the memory representation of your data, but doesn't really change the way the chip uses it. In fact, it guarantees you *at least* floating point accuracy. Because optimized code keeps data in registers longer, it will sometimes be more precise than debug code, which flushes to the stack often.

Again: typing "float" does not change anything that goes on in the FPU internally. If you don't actively change the chip's precision, you're probably doing double-precision arithmetic while an operand is being processed.

The x86 FPU *does* have precision control, which will stop calculation after it's computed enough to achieve floating point accuracy. But, you can't change FPU precision from ANSI C.

What runs faster in single precision?

Speedwise, single precision affects exactly two calls: **divides** and **sqrts**. It won't make trancendentals any faster (sin, acos, log, etc., all run the same no matter what: 100+ cycles.) If your program does lots of single-precision divides and sqrts, then you need to know about this.

Single precision will at least double the speed of divides and sqrts. Divides take 17 cycles and sqrts take about 25 cycles. In double precision, they're at least twice that.

On x86, precision control is adjusted using the assembly call "fldcw". Microsoft has a nice wrapper called **_controlfp** that's easier to use. If you're using Linux, I recommend getting the Intel Instruction Set Reference and writing the inline assembly. (Send me code so I can post it!)

To set the FPU to single precision:

```
  _controlfp( _PC_24, MCW_PC );
```

To set it back to default (double) precision:

```
  _controlfp( _CW_DEFAULT, 0xfffff );
```

I use a C++ class that sets the precision while it's in scope and then drops back to the previous rounding and precision mode as it goes out -- very convenient, so you don't forget to reset the precision, and you can handle error cases properly.

Fast conversions: float to int

Lots of people have been talking about how bad Intel chips are at converting floating point to integer. (Intel, NVIDIA, and hordes of people on Usenet.)

I thought it would be fun to try to do the definitive, complete, version of this discussion. This will get a little winded at the end, but try to hang on -- good stuff ahead.

So, if you ever do this:

```
inline int Convert(float x)
{
  int i = (int) x;
  return i;
}
```

or you call floor() or ceil() or anything like that, you probably shouldn't.

The problem is that there is no dedicated x86 instruction to do an "ANSI C" conversion from floating point to integer. There is an instruction on the chip that does a conversion (**fistp**), but it respects the chip's current rounding mode (set with fldcw like the precision flags up above.) Of course, the default rounding mode does not clamp to zero.

To implement a "correct" conversion, compiler writers have had to switch the rounding mode, do the conversion with fistp, and switch back. Each switch requires a complete flush of the floating point state, and takes about 25 cycles.

This means that this function, even inlined, takes upwards of **80 (EIGHTY) cycles**!

Let me put it this way: this isn't something to scoff at and say, "Machines are getting faster." I've seen quite well-written code get **8 TIMES FASTER** after fixing conversion problems. Fix yours! And here's how...

Sree's Real2Int

There are quite a few ways to fix this. One of the best general conversion utilities I have is one that Sree Kotay wrote. It's only safe to use when your FPU is in *double-precision* mode, so you have to be careful about it if you're switching precision like above. It'll basically return '0' all the time if you're in single precision.

(I've recently discovered that this discussion bears a lot of resemblance to Chris Hecker's article on the topic, so be sure to read that if you can't figure out what I'm saying.)

This function is basically an ANSI C compliant conversion -- it always chops, always does the right thing, for positive or negative inputs.

```
typedef double lreal;
typedef float  real;
typedef unsigned long uint32;
typedef long int32;

const lreal _double2fixmagic = 68719476736.0*1.5;     //2^36 * 1.5,  (52-_shiftamt=36) uses limited precisicion to floor
const int32 _shiftamt        = 16;                    //16.16 fixed point representation,

#if BigEndian_
        #define iexp_                           0
        #define iman_                           1
#else
        #define iexp_                           1
        #define iman_                           0
#endif //BigEndian_

// ================================================================================================
// Real2Int
// ================================================================================================
inline int32 Real2Int(lreal val)
{
#if DEFAULT_CONVERSION
        return val;
#else
        val             = val + _double2fixmagic;
        return ((int32*)&val)[iman_] >> _shiftamt;
#endif
}

// ================================================================================================
// Real2Int
// ================================================================================================
inline int32 Real2Int(real val)
{
#if DEFAULT_CONVERSION
        return val;
#else
        return Real2Int ((lreal)val);
#endif
}
```

fistp

In either precision mode, you can call fistp (the regular conversion) directly. However, this means you need to know the current FPU rounding mode. In most cases, the following function will round the conversion to the nearest integer, rather than clamping towards zero, so that's what I call it:

```
inline int32 Round(real32 a) {
        int32 retval;

        __asm fld a
        __asm fistp retval

        return retval;
}
```

Direct conversions

I originally learned this one (and this way of thinking about the FPU) from Dan Wallach when we were both at Microsoft in the summer of 1995. He showed me how to implement a fast lookup table by direct manipulation of the mantissa of a floating point number in a certain range.

First, you have to know a single precision float has three components:

```
  sign[1 bit]  exp[8 bits]  mantissa[23 bits]
```

The value of the number is computed as:
(-1)^sign * 1.mantissa * 2^(exp - 127)

So what does that mean? Well, it has a nice effect:

Between any power of two and the next (e.g., [2, 4)), the exponent is *constant*. For certain cases, you can just mask out the mantissa, and use that as a fixed-point number. Let's say you know your input is between 0 and 1, not including 1. You can do this:

```
int FloatTo23Bits(float x)
{
    float y = x + 1.f;
    return ((unsigned long&)y) & 0x7FFFFF;      // last 23 bits
}
```

The first line makes 'y' lie between 1 and 2, where the exponent is a constant 127. Reading the last 23 bits gives an exact 23 bit representation of the number between 0 and 1. Fast and easy.

Timing: fast conversions

Sree and I have benchmarked and spent a lot of time on this problem. I tend to use fistp a lot more than his conversion, because I write a lot of code that needs single precision. His tends to be faster, so I need to remember to use it more.

In specific cases, the direct conversions can be very useful, but they're about as fast as Sree's code, so generally there's not too much advantage. You can save a shift, maybe, and it does work in single precision mode.

Each of the above functions is about 6 cycles by itself. However, the **Real2Int** version is **much** better at pairing with floating point (it doesn't lock the FPU for 6 cycles like **fistp** does.) So in really well pipelined code, his function can be close to free. **fistp** will take about 6 cycles.

Hacking VC's Runtime for Fun and Debugging

Finally, here's a function that overrides Microsoft's **_ftol** (the root of all evil.) One of the comp.lang.x86.asm guys ripped this out of Microsoft's library, then Terje Mathison did the "OR" optimization, and then I hacked it into manually munging of the stack (note: no frame pointer?) like you see below. But don't do this at home -- it's evil too.

It also may have some bugs -- I'm not sure it works when there are floating point exceptions. Anyway, it's about 10-15 cycles faster than Microsoft's implementation.

Also, it will show up in a profile, which is a really great thing. You can put a breakpoint in it and see who's calling slow conversions, then go fix them. Overall, **I don't recommend shipping code with it**, but definitely use it for debugging.

Also, there's a "fast" mode that does simple rounding instead, but that doesn't get inlined, and it's generally a pain, since you can't get "ANSI correct" behavior when you want it.

```
#define ANSI_FTOL 1

extern "C" {
    __declspec(naked) void _ftol() {
        __asm    {
#if ANSI_FTOL
            fnstcw   WORD PTR [esp-2]
            mov      ax, WORD PTR [esp-2]

            OR AX,       0C00h

            mov      WORD PTR [esp-4], ax
            fldcw    WORD PTR [esp-4]
            fistp    QWORD PTR [esp-12]
            fldcw    WORD PTR [esp-2]
            mov      eax, DWORD PTR [esp-12]
            mov      edx, DWORD PTR [esp-8]
#else
            fistp    DWORD PTR [esp-12]
            mov          eax, DWORD PTR [esp-12]
            mov          ecx, DWORD PTR [esp-8]
#endif
            ret
        }
    }
}
```

I got some great code from Kevin Egan at Brown University. He reimplemented precision control and the inline conversion calls with AT&T-compatible syntax so they work on gcc.

It's a lot of code, so I'm posting it at the end here. :)

```
#include
#include

#ifdef USE_SSE
    #include
#endif // USE_SSE



/*
SAMPLE RUN

~/mod -> g++-3.2 fpu.cpp
~/mod -> ./a.out

ANSI slow, default FPU, float 1.80000 int 1
fast, default FPU, float 1.80000 int 2
ANSI slow, modified FPU, float 1.80000 int 1
fast, modified FPU, float 1.80000 int 1

ANSI slow, default FPU, float 1.80000 int 1
fast, default FPU, float 1.80000 int 2
ANSI slow, modified FPU, float 1.80000 int 1
fast, modified FPU, float 1.80000 int 1

ANSI slow, default FPU, float 1.10000 int 1
fast, default FPU, float 1.10000 int 1
ANSI slow, modified FPU, float 1.10000 int 1
fast, modified FPU, float 1.10000 int 1

ANSI slow, default FPU, float -1.80000 int -1
fast, default FPU, float -1.80000 int -2
ANSI slow, modified FPU, float -1.80000 int -1
fast, modified FPU, float -1.80000 int -1

ANSI slow, default FPU, float -1.10000 int -1
```

```
fast, default FPU, float -1.10000 int -1
ANSI slow, modified FPU, float -1.10000 int -1
fast, modified FPU, float -1.10000 int -1
*/


/**
 * bits to set the floating point control word register
 *
 * Sections 4.9, 8.1.4, 10.2.2 and 11.5 in
 * IA-32 Intel Architecture Software Developer's Manual
 *   Volume 1: Basic Architecture
 *
 * http://www.intel.com/design/pentium4/manuals/245471.htm
 *
 * http://www.geisswerks.com/ryan/FAQS/fpu.html
 *
 * windows has _controlfp() but it takes different parameters
 *
 * 0 : IM invalid operation mask
 * 1 : DM denormalized operand mask
 * 2 : ZM divide by zero mask
 * 3 : OM overflow mask
 * 4 : UM underflow mask
 * 5 : PM precision, inexact mask
 * 6,7 : reserved
 * 8,9 : PC precision control
 * 10,11 : RC rounding control
 *
 * precision control:
 * 00 : single precision
 * 01 : reserved
 * 10 : double precision
 * 11 : extended precision
 *
 * rounding control:
 * 00 = Round to nearest whole number. (default)
 * 01 = Round down, toward -infinity.
 * 10 = Round up, toward +infinity.
 * 11 = Round toward zero (truncate).
 */
#define __FPU_CW_EXCEPTION_MASK__   (0x003f)
#define __FPU_CW_INVALID__          (0x0001)
#define __FPU_CW_DENORMAL__         (0x0002)
#define __FPU_CW_ZERODIVIDE__       (0x0004)
#define __FPU_CW_OVERFLOW__         (0x0008)
#define __FPU_CW_UNDERFLOW__        (0x0010)
#define __FPU_CW_INEXACT__          (0x0020)

#define __FPU_CW_PREC_MASK__        (0x0300)
#define __FPU_CW_PREC_SINGLE__      (0x0000)
#define __FPU_CW_PREC_DOUBLE__      (0x0200)
#define __FPU_CW_PREC_EXTENDED__    (0x0300)

#define __FPU_CW_ROUND_MASK__       (0x0c00)
#define __FPU_CW_ROUND_NEAR__       (0x0000)
#define __FPU_CW_ROUND_DOWN__       (0x0400)
#define __FPU_CW_ROUND_UP__         (0x0800)
#define __FPU_CW_ROUND_CHOP__       (0x0c00)

#define __FPU_CW_MASK_ALL__         (0x1f3f)


#define __SSE_CW_FLUSHZERO__        (0x8000)

#define __SSE_CW_ROUND_MASK__       (0x6000)
#define __SSE_CW_ROUND_NEAR__       (0x0000)
#define __SSE_CW_ROUND_DOWN__       (0x2000)
#define __SSE_CW_ROUND_UP__         (0x4000)
#define __SSE_CW_ROUND_CHOP__       (0x6000)

#define __SSE_CW_EXCEPTION_MASK__   (0x1f80)
#define __SSE_CW_PRECISION__        (0x1000)
#define __SSE_CW_UNDERFLOW__        (0x0800)
#define __SSE_CW_OVERFLOW__         (0x0400)
#define __SSE_CW_DIVIDEZERO__       (0x0200)
#define __SSE_CW_DENORMAL__         (0x0100)
#define __SSE_CW_INVALID__          (0x0080)
// not on all SSE machines
// #define __SSE_CW_DENORMALZERO__     (0x0040)

#define __SSE_CW_MASK_ALL__         (0xffc0)



#define __MOD_FPU_CW_DEFAULT__ \
    (__FPU_CW_EXCEPTION_MASK__ + __FPU_CW_PREC_DOUBLE__ + __FPU_CW_ROUND_CHOP__)

#define __MOD_SSE_CW_DEFAULT__ \
    (__SSE_CW_EXCEPTION_MASK__ + __SSE_CW_ROUND_CHOP__ + __SSE_CW_FLUSHZERO__)
```

```
#ifdef USE_SSE
inline unsigned int getSSEStateX86(void);
inline void setSSEModDefault(unsigned int control);
inline void modifySSEStateX86(unsigned int control, unsigned int mask);
#endif // USE_SSE


inline void setRoundingMode(unsigned int round);

inline unsigned int getFPUStateX86(void);
inline void setFPUStateX86(unsigned int control);

inline void setFPUModDefault(void);
inline void assertFPUModDefault(void);

inline void modifyFPUStateX86(const unsigned int control, const unsigned int mask);

inline int FastFtol(const float a);

// assume for now that we are running on an x86
// #ifdef __i386__

#ifdef USE_SSE

inline
unsigned int
getSSEStateX86
    (void)
{
    return _mm_getcsr();
}

inline
void
setSSEStateX86
    (unsigned int control)
{
    _mm_setcsr(control);
}


inline
void
modifySSEStateX86
    (unsigned int control, unsigned int mask)
{
    unsigned int oldControl = getFPUStateX86();
    unsigned int newControl = ((oldControl & (~mask)) | (control & mask));
    setFPUStateX86(newControl);
}


inline
void
setSSEModDefault
    (void)
{
    modifySSEStateX86(__MOD_SSE_CW_DEFAULT__, __SSE_CW_MASK_ALL__);
}
#endif // USE_SSE


inline
void
setRoundingMode
    (Uint32 round)
{
    ASSERT(round < 4);
    Uint32 mask = 0x3;

    Uint32 fpuControl = getFPUStateX86();
    fpuControl &= ~(mask << 10);
    fpuControl |= round << 10;
    setFPUStateX86(fpuControl);

#ifdef USE_SSE
    Uint32 sseControl = getSSEStateX86();
    sseControl &= ~(mask << 13);
    sseControl |= round << 13;
    setSSEStateX86(sseControl);
#endif // USE_SSE
}


inline
unsigned int
getFPUStateX86
    (void)
{
    unsigned int control = 0;
#if defined(_MSVC)
    __asm fnstcw control;
#elif defined(__GNUG__)
    __asm__ __volatile__ ("fnstcw %0" : "=m" (control));
#endif
    return control;
}



/* set status */
inline
void
setFPUStateX86
    (unsigned int control)
{
#if defined(_MSVC)
    __asm fldcw control;
#elif defined(__GNUG__)
    __asm__ __volatile__ ("fldcw %0" : : "m" (control));
#endif
}


inline
void
modifyFPUStateX86
    (const unsigned int control, const unsigned int mask)
{
    unsigned int oldControl = getFPUStateX86();
    unsigned int newControl = ((oldControl & (~mask)) | (control & mask));
    setFPUStateX86(newControl);
}


inline
void
setFPUModDefault
    (void)
{
    modifyFPUStateX86(__MOD_FPU_CW_DEFAULT__, __FPU_CW_MASK_ALL__);
    assertFPUModDefault();
}


inline
void
assertFPUModDefault
    (void)
{
    assert((getFPUStateX86() & (__FPU_CW_MASK_ALL__)) ==
            __MOD_FPU_CW_DEFAULT__);
}

// taken from http://www.stereopsis.com/FPU.html
// this assumes the CPU is in double precision mode
inline
int
FastFtol(const float a)
{
    static int    b;

#if defined(_MSVC)
    __asm fld a
    __asm fistp b
#elif defined(__GNUG__)
    // use AT&T inline assembly style, document that
    // we use memory as output (=m) and input (m)
    __asm__ __volatile__ (
        "flds %1        \n\t"
        "fistpl %0      \n\t"
        : "=m" (b)
        : "m" (a));
#endif
    return b;
}




void test()
{
    float testFloats[] = { 1.8, 1.8, 1.1, -1.8, -1.1 };
    int testInt;
    unsigned int oldControl = getFPUStateX86();

    for (int i = 0; i < 5; i++) {
        float curTest = testFloats[i];

        setFPUStateX86(oldControl);
        testInt = (int) curTest;
        printf("ANSI slow, default FPU, float %.5f int %d\n",
                curTest, testInt);
        testInt = FastFtol(curTest);
        printf("fast, default FPU, float %.5f int %d\n",
                curTest, testInt);

        setFPUModDefault();

        testInt = (int) curTest;
        printf("ANSI slow, modified FPU, float %.5f int %d\n",
                curTest, testInt);
        testInt = FastFtol(curTest);
        printf("fast, modified FPU, float %.5f int %d\n\n",
                curTest, testInt);
    }


}


int
main
    (int argc, char** argv)
{
    test();
    return 0;
}
```