Exploring Binary

When Floats Don't Behave Like Floats

By Rick Regan March 30th, 2010

These two programs — compiled with Microsoft Visual C++ and run on a 32-bit Intel Core Duo processor — demonstrate an anomaly that occurs when using single-precision floating point variables:

Program 1

```
#include "stdio.h"
int main (void)
{
 float f1 = 0.1f, f2 = 3.0f, f3;

 f3 = f1 * f2;
 if (f3 != f1 * f2)
   printf("Not equal\n");
}
```

Prints "**Not equal**".

Program 2

```
#include "stdio.h"
int main (void)
{
 float f1 = 0.7f, f2 = 10.0f, f3;
 int i1, i2;

 f3 = f1 * f2;
 i1 = (int)f3;
 i2 = (int)(f1 * f2);
 if (i1 != i2)
   printf("Not equal\n");
}
```

Prints "**Not equal**".

In each case, f3 and f1 * f2 differ. But why? I'll explain what's going on.

(This article was inspired by this question and several related questions on stackoverflow.com.)

Analyzing Program 1

Program 1 compares two floating point values for equality — something conventional wisdom says not to do. But why should that rule apply in this case? We're just checking that a variable holds the value we just gave it. How could it not?

The root of the "problem" is this: the compiler generates instructions that do floating-point calculations in extended precision. Whereas floats are 4 bytes long and have 24 significant bits of precision, extended precision values — which are stored in registers on the floating point stack — are 10 bytes long and have 64 bits of precision.

Computing f3

Using my function fp2bin, I printed the binary values of the three floating point variables in the computation:

- f1 = 0.000110011001100110011001101
- f2 = 11
- f3 = 0.0100110011001100110011001101

The value of f3 is only an approximation to f1 * f2. To understand why, let's calculate the true value of f1 * f2; that is, let's calculate it by hand, using binary multiplication:

```
  0.000110011001100110011001101
  x                           11
  -----------------------------
       110011001100110011001101
      110011001100110011001101
-------------------------------
  0.0100110011001100110011001100111
```

(You can also compute this with my binary calculator.)

So f1 * f2 = 0.0100110011001100110011001100111. It fits comfortably within extended precision. But to assign it to f3 — a float — it must be rounded. f1 * f2 has 26 significant bits, but a float holds only 24. Rounding it to the nearest 24 bit value makes it 0.0100110011001100110011001101.

The assembler code generated by the compiler confirms what we're seeing:

```
 f3 = f1 * f2;
0041130B  fld         dword ptr [f1]
0041130E  fmul        dword ptr [f2]
00411311  fstp        dword ptr [f3]
```

f1 * f2 is computed in extended precision — enough bits to hold its true value — but that extra precision is lost when stored in f3.

Comparing f3 and f1 * f2

The answer is in the assembler code, so let's get right to it:

```
 if (f3 != f1 * f2)
00411314  fld         dword ptr [f1]
00411317  fmul        dword ptr [f2]
0041131A  fld         dword ptr [f3]
0041131D  fucompp
```

Again, f1 * f2 is computed in extended precision, but this time **its true value is retained** — it is left on the stack. f3 is then loaded onto the stack (of course it still has only 24 bits of precision, even though it's been "promoted" to extended precision). The two values on top of the stack are then compared. Clearly, they differ.

Analyzing Program 2

Program 2 "fails" for the same reason as program 1, except that the "error" is magnified by the conversion of f3 and f1 * f2 to integers: the integer part of f3 is **7**, and the integer part of f1 * f2 is **6**.

Computing i1

The binary values of the three floating point variables in the computation are:

- f1 = 0.101100110011001100110011
- f2 = 1010
- f3 = 111

The value of f3 is an integer. To understand why, let’s calculate the true value of f1 * f2:

```
  0.101100110011001100110011
  x                     1010
  --------------------------
                           0
   101100110011001100110011
                         0
 101100110011001100110011
----------------------------
110.11111111111111111111111
```

So f1 * f2 = 110.11111111111111111111111. To assign it to f3 it must be rounded. f1 * f2 has 26 significant bits, and rounding it to the nearest 24 bit value makes it 111, or 7 decimal. Clearly, this means i1 will be 7 as well.

Here’s the assembler code:

```
 f3 = f1 * f2;
0041130B  fld         dword ptr [f1]
0041130E  fmul        dword ptr [f2]
00411311  fstp        dword ptr [f3]
 i1 = (int)f3;
00411314  fld         dword ptr [f3]
00411317  call        @ILT+155(__ftol2_sse) (4110A0h)
           ||
           ||
           \/
--- f:\dd\vctools\crt_bld\SELF_X86\crt\prebuild\tran\i386\ftol2.asm
00411600  cmp         dword ptr [___sse2_available (416554h)],0
00411607  je          _ftol2 (411636h)
00411609  push        ebp
0041160A  mov         ebp,esp
0041160C  sub         esp,8
0041160F  and         esp,0FFFFFFF8h
00411612  fstp        qword ptr [esp]
00411615  cvttsd2si   eax,mmword ptr [esp]
0041161A  leave
0041161B  ret
```

Computing i2

Here's the assembler code for computing i2:

```
 i2 = (int)(f1 * f2);
0041131F  fld         dword ptr [f1]
00411322  fmul        dword ptr [f2]
00411325  call        @ILT+155(__ftol2_sse) (4110A0h)
           ||
           ||
           \/
--- f:\dd\vctools\crt_bld\SELF_X86\crt\prebuild\tran\i386\ftol2.asm
00411600  cmp         dword ptr [___sse2_available (416554h)],0
00411607  je          _ftol2 (411636h)
00411609  push        ebp
0041160A  mov         ebp,esp
0041160C  sub         esp,8
0041160F  and         esp,0FFFFFFF8h
00411612  fstp        qword ptr [esp]
00411615  cvttsd2si   eax,mmword ptr [esp]
0041161A  leave
0041161B  ret
```

The interesting thing in the calculation of i2 is that f1 * f2 is stored in a *double* before being cast to an integer (see the highlighted fstp instruction). f1 * f2 in double precision is 110.11111111111111111111111; that is, its true 26 bit value. Casting this to an integer results in 110, or decimal 6. If the compiler had stored f1 * f2 in a float and used the cvtts**s**2si instruction instead, the answer would have been 7 — just like in the first calculation.

Discussion

In program 2, it would appear that single-precision is more accurate than extended precision. After all, forcing the value into a float gives the expected answer. This is just a happy coincidence. Two losses of precision — the conversion of 0.7 to floating-point and the rounding up of the product — have effectively canceled each other out.

Behavior Depends on the Processor and How It Is Used

This anomaly may not occur on your machine. It depends on your processor, and in particular, the instructions used and the mode that it's in. My example programs were compiled into Intel x87 FPU instructions, and ran with the x87 precision control field set to 53-bits. (I said above that my programs were using extended precision; technically they weren't, but double precision is sufficient to make them "fail". The true values of f1 * f2 are less than 53 bits.)

Intel processors can also do floating point using SSE (Streaming SIMD Extensions) instructions. I recompiled my programs using the Visual C++ compiler options /arch:SSE (single-precision floating-point) and /arch:SSE2 (double-precision floating-point). For the SSE option, there was no change; the compiler, at its discretion, still decided to generate x87 instructions. (See the comment below about the Mac's use of SSE instructions making the anomaly disappear.)

Recompiling with /arch:SSE2 I got different assembler code, but the same output; here's the assembler code for program 1:

```
f3 = f1 * f2;
00411313  cvtss2sd    xmm0,dword ptr [f1]
00411318  cvtss2sd    xmm1,dword ptr [f2]
0041131D  mulsd       xmm0,xmm1
00411321  cvtsd2ss    xmm0,xmm0
00411325  movss       dword ptr [f3],xmm0
if (f3 != f1 * f2)
0041132A  cvtss2sd    xmm0,dword ptr [f1]
0041132F  cvtss2sd    xmm1,dword ptr [f2]
00411334  mulsd       xmm0,xmm1
00411338  cvtss2sd    xmm1,dword ptr [f3]
0041133D  ucomisd     xmm1,xmm0
```

The SSE double-precision instructions are used. f3 — single precision — is compared to the double-precision intermediate result f1 * f2, resulting in a mismatch.

Please Try it Out

If you have access to a different compiler or processor, please try these programs out. Let me know what you find!

The Message

You can't count on floats being handled as single precision values; they can be processed in double or extended precision, as dictated by your compiler and CPU.

(I have written a companion article called "When Doubles Don't Behave Like Doubles".)

EB

Related

- Quick and Dirty Decimal to Floating-Point Conversion
- What Powers of Two Look Like Inside a Computer

Get articles by RSS (What Is RSS?)

Get articles by e-mail

Numbers in computers / Binary arithmetic, C, Code, Convert to binary, Decimals, Floating-point

5 comments

1. **Rick Regan**

   April 5, 2010 at 2:13 pm

A reader (thanks!) tried these two programs on an Intel Core 2 Duo Mac and found they ***did not*** produce the anomaly. The disassembly shows why: the single-precision SSE instructions are used!

Program 1

```
0x00001fb5 :    movss  -0x14(%ebp),%xmm0
0x00001fba :    mulss  -0x10(%ebp),%xmm0
0x00001fbf :    movss  %xmm0,-0xc(%ebp)
0x00001fc4 :    movss  -0x14(%ebp),%xmm0
0x00001fc9 :    mulss  -0x10(%ebp),%xmm0
0x00001fce :    ucomiss -0xc(%ebp),%xmm0
```

Program 2

```
0x00001fa5 :    movss  -0x1c(%ebp),%xmm0
0x00001faa :    mulss  -0x18(%ebp),%xmm0
0x00001faf :    movss  %xmm0,-0x14(%ebp)
0x00001fb4 :    movss  -0x14(%ebp),%xmm0
0x00001fb9 :    cvttss2si %xmm0,%eax
0x00001fbd :    mov    %eax,-0x10(%ebp)
0x00001fc0 :    movss  -0x1c(%ebp),%xmm0
0x00001fc5 :    mulss  -0x18(%ebp),%xmm0
0x00001fca :    cvttss2si %xmm0,%eax
```

(I added a new section to my article called “Behavior Depends on the Processor and How It Is Used”.)

---

2. **John**

November 12, 2011 at 3:30 am

Ran your code under Mac OS X and it did not happen to produce the same results as you. It appears that your compiler is malfunctional.

---

3. **Rick Regan**

November 12, 2011 at 8:48 am

@John,

If you want to look at it that way, mine and millions of other computers are "malfunctional" (x87 FPU instructions).

4. **Bruce Dawson**

   March 30, 2012 at 11:52 pm

   It is incorrect (responding to John's November 12th comment) to describe the compiler as malfunctional. The problem which Rick describes is a classic one, known since the early days of the IEEE format. The decision about whether to use higher precision intermediates is tricky. They improve some calculations significantly, but they lead to problems. There is no simple answer, and the experts know that. I discussed some aspects of it here:

   http://randomascii.wordpress.com/2012/03/21/intermediate-floating-point-precision/

   Your 'superior' Mac OS X compiler will produce worse results in some cases. That is simply the nature of the beast.

5. Pingback: Floating-point complexities | Random ASCII

Comments are closed.