Exploring Binary

When Doubles Don’t Behave Like Doubles

By Rick Regan April 9th, 2010

In my article “When Floats Don’t Behave Like Floats” I explained how calculations involving single-precision floating-point variables may be done, under the covers, in double or extended precision. This leads to anomalies in expected results, which I demonstrated with two C programs — compiled with Microsoft Visual C++ and run on a 32-bit Intel Core Duo processor.

In this article, I’ll do a similar analysis for *double*-precision floating-point variables, showing how similar anomalies arise when extended precision calculations are done. I modified my two example programs to use doubles instead of floats. Interestingly, the doubles version of program 2 does not exhibit the anomaly. I’ll explain.

The Environment in Which the Anomalies Occur

To see the described anomalies, floating-point calculations must be done in precision greater than double-precision. For Intel processors, this means that x87 FPU instructions must be used (as opposed to SSE instructions), and the processor must be calculating in **extended-precision** mode.

Your compiler determines which instructions to generate, although you may influence it with compiler options. On my system, Visual C++ generates x87 FPU instructions by default. As for the precision mode, it has a default — which is double-precision on my system — that can be changed by a call to the *controlfp* function. I call this function in the programs below.

Program 1

```
#include "stdio.h"
#include "float.h"
int main (void)
{
 double d1 = 0.3333333333333333, d2 = 3.0, d3;
 unsigned int cur_cw;
 _controlfp_s(&cur_cw,_PC_64,MCW_PC);

 d3 = d1 * d2;
 if (d3 != d1 * d2)
   printf("Not equal\n");
}
```

Program 1 prints "**Not equal**", for the same reason its float counterpart does: greater precision is being used under the covers.

Computing d3

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

- d1 = 0.010101010101010101010101010101010101010101010101010101
- d2 = 11
- d3 = 1

The value of d3 is only an approximation to d1 * d2. Here's the true value of d1 * d2, computed by hand:

```
0.010101010101010101010101010101010101010101010101010101
x                                                     11
--------------------------------------------------------
   10101010101010101010101010101010101010101010101010101
  10101010101010101010101010101010101010101010101010101
--------------------------------------------------------
0.111111111111111111111111111111111111111111111111111111
```

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

The product is 54 bits, which can be represented exactly in extended precision, but only approximately in double-precision. Since bit 54 is a 1, the value is rounded up, to 53 bits. The result is d3 = 1.

Here's the relevant assembler code generated by the compiler:

```
 d3 = d1 * d2;
0041131F  fld         qword ptr [d1]
```

```
00411322  fmul        qword ptr [d2]
00411325  fstp        qword ptr [d3]
```

Comparing d3 and d1 * d2

The assembler code shows what is happening:

```
 if (d3 != d1 * d2)
00411328  fld         qword ptr [d1]
0041132B  fmul        qword ptr [d2]
0041132E  fcomp       qword ptr [d3]
```

d1 * d2 is computed in extended precision and left on the stack. It is then compared to d3, which still has 53 bits of precision even though it's "promoted" to extended precision. The two compared values will differ.

Program 2

```
#include "stdio.h"
#include "float.h"
int main (void)
{
 double d1 = 0.3333333333333333, d2 = 3.0, d3;
 long long i1, i2;
 unsigned int cur_cw;

 _controlfp_s(&cur_cw,_PC_64,MCW_PC);

 d3 = d1 * d2;
 i1 = (int)d3;
 i2 = (int)(d1 * d2);
 if (i1 != i2)
   printf("Not equal\n");
}
```

Program 2 does not print anything, which is surprising knowing that its float counterpart does. According to our analysis of program 1 — program 2 uses the same values for d1 and d2 — d1 * d2 rounds to 1 in double precision, but is slightly less than 1 in extended precision. Therefore, we would expect i1 = 1 and i2 = 0. Instead, both are 1.

The explanation is simple: the extended precision calculation of d1 * d2 is "spilled" into a double before the conversion to integer, so it will be equal to d3:

Computing i2

```
 i2 = (int)(d1 * d2);
004114A5  fld         qword ptr [d1]
004114A8  fmul        qword ptr [d2]
004114AB  call        @ILT+245(__ftol2_sse) (4110FAh)
           ||
           ||
           \/
--- f:\dd\vctools\crt_bld\SELF_X86\crt\prebuild\tran\i386\ftol2.asm
00411B40  cmp         dword ptr [___sse2_available (417598h)],0
00411B47  je          _ftol2 (411B76h)
00411B49  push        ebp
00411B4A  mov         ebp,esp
00411B4C  sub         esp,8
00411B4F  and         esp,0FFFFFFF8h
00411B52  fstp        qword ptr [esp]
00411B55  cvttsd2si   eax,mmword ptr [esp]
00411B5A  leave
00411B5B  ret
```

Related

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

How to Read a Binary Clock

An Hour of Code... A Lifelong...

Visualizing Consecutive Binary...

A Twelve Cent Binary Calendar

Properties of the Correction Loop in...

How to Install and Run GMP on Windows...

Decimal/ Binary Conversion...

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

2 comments

1. **Rick Regan**

   April 12, 2010 at 1:36 pm

   A reader tried these two programs on an Intel Core 2 Duo Mac and found they did not produce the anomaly. The disassembly shows why: SSE instructions (double-precision) are used (movsd, ucomisd, mulsd, cvttsd2si, etc.).

2. **Bruce Dawson**

   March 31, 2012 at 12:15 am

   You can create this anomaly without even changing the rounding precision by exploiting the extended exponent range that occurs in the x87 registers even when they are set to _PC_53 (double) precision. This code, by default on 32-bit VC++ x86 apps, prints "Not Equal\n"

   ```
   void DoublesAreNotDoublesTest()
   {
   double d1 = DBL_MAX, d2 = 3.0, d3;

   d3 = d1 * d2;
   if (d3 != d1 * d2)
   printf("Not equal\n");
   }
   ```

   Similar results could occur in the denormal range where a full 53-bits of precision is retained, when double precision would not have it.

Everything changes on VC++ for x64 or with /arch:SSE2 (especially with VS 11)). Otherwise the compiler is forced to assume that the CPU the code is run on might not have SSE instructions.

Comments are closed.