CS:APP Web Aside DATA:IA32-FP: Intel IA32 Floating-Point Arithmetic*

Randal E. Bryant
David R. O'Hallaron

June 5, 2012

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer's Perspective, Second Edition*, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2011. In this document, all references beginning with "CS:APP2e " are to this book. More information about the book is available at* `csapp.cs.cmu.edu`.

1 x87 Floating Point

In the CS:APP2e Chapter 3, we will begin an in-depth study of Intel IA32 processors, the processor found in many of today's personal computers. Here we highlight an idiosyncrasy of these machines that can seriously affect the behavior of programs operating on floating-point numbers when compiled with GCC.

Intel IA32 processors, like most other processors, have special memory elements called *registers* for holding floating-point values as they are being computed and used. Values held in registers can be read and written more quickly than those held in the main memory. The unusual feature of IA32 is that the floating-point registers use a special 80-bit *extended-precision* format to provide a greater range and precision than the normal 32-bit single-precision and 64-bit double-precision formats used for values held in memory. The extended-precision representation is similar to an IEEE floating-point format with a 15-bit exponent (i.e., $k = 15$) and a 63-bit fraction (i.e., $n = 63$). All single and double-precision numbers are converted to this format as they are loaded from memory into floating-point registers. The arithmetic is always performed in extended precision. Numbers are converted from extended precision to single or double-precision format as they are stored in memory.

2 Implications for Programs Executing on IA32 Machines

This extension to 80 bits for all register data and then contraction to a smaller format for memory data has some undesirable consequences for programmers. It means that storing a number from a register to memory and then retrieving it back into the register can cause it to change, due to rounding, underflow, or overflow. This storing and retrieving is not always visible to the C programmer, leading to some very peculiar results.

The following program snippet illustrates this property. It should be noted that the anomalous behavior we show here depends greatly on the particular compiler and command-line options used, and so the result shown below may not occur when compiled with other compilers, including other versions of GCC.

```
1  volatile int rcnt = 0;  /* Used to create side effects */
2
3  double recip(int denom) {
4      rcnt++;   /* Side effect to prevent optimization */
5      return 1.0/(double) denom;
6  }
7
8  int dequal(double x, double y) {
9      return x==y;
10 }
11
12 void test1(int denom) {
13     double r1, r2;
14     int t1, t2;
15
16     r1 = recip(denom);    /* Stored in memory              */
17     r2 = recip(denom);    /* Stored in register            */
18     t1 = r1 == r2;        /* Compares register to memory   */
19     t2 = dequal(r1,r2);   /* Compares memory to memory     */
20     printf("test1 t1: r1 %f %c= r2 %f\n", r1, t1 ? '=' : '!', r2);
21     printf("test1 t2: r1 %f %c= r2 %f\n", r1, t2 ? '=' : '!', r2);
22 }
```

Variables `r1` and `r2` are computed by the same function with the same argument. One would expect them to be identical. Furthermore, both variables `t1` and `t2` are computed by evaluating the expression `r1 == r2`, and so we would expect them both to equal `1`. When the complete program is compiled on an IA32 machine with command line option '`-O2`' and run with argument `10`, however, we get the following result:

```
test1 t1: r1 0.100000 != r2 0.100000
test1 t2: r1 0.100000 == r2 0.100000
```

The first test indicates the two reciprocals are different, while the second indicates they are the same! This is certainly not what we expect, nor what we want. Understanding all of the details of this example requires studying the machine-level floating-point code generated by GCC (see Web Aside ASM:X87), but the comments in the code provide a clue as to why this outcome occurs. The value computed by function

`recip` returns its result in a floating-point register. Whenever procedure `test1` calls a function, it must first store any value currently in a floating-point register to memory. In performing this store operation, the processor converts the extended-precision register values to double-precision memory values. Thus, before making the second call to `recip` (line 17), variable `r1` is converted and stored as a double-precision number. After the second call, variable `r2` has the extended-precision value returned by the function. As we have seen, no finite-precision floating-point format can exactly represent the value 0.1, and the 64-bit approximation `r1` is a rounded version of the 80-bit approximation `r2`. In computing `t1` (line 18), these two approximations are compared and are found to be different. Value `t2` is computed by calling the function `dequal` (line 19.) Arguments are passed to functions by storing them in memory. In storing `r2` to memory, it is converted to double precision, and hence the comparison within `dequal` (line 9) is made between identical, 64-bit approximations of 0.1.

This example demonstrates a deficiency of GCC on IA32 machines (the same result occurs on both Linux and Microsoft Windows implementations). The value associated with a variable changes due to operations that are not visible to the programmer, such as the saving and restoring of floating-point registers. Our experiments with the Microsoft Visual C++ compiler indicate that it does not have this problem. When compiled in 64-bit mode on a more recent Intel machine, the anomaly does not occur, because GCC makes use of a different floating-point capability on these machines, where all computation is done using 64-bit, double-precision numbers. One of the fundamental principles of optimizing compilers is that programs should produce the exact same results whether or not optimization is enabled. Unfortunately, GCC does not satisfy this requirement for floating-point code on IA32 machines.

3 Possible Remediations

There are several ways to overcome this problem, although none is ideal. The most reliable method we have found is to have GCC use extended-precision format in all of its computations by declaring all of the variables to be of type `long double`, as illustrated in the following reciprocal function

```
long double recip_l(int denom) {
    return 1.0/(long double) denom;
}
```

The declaration `long double` is supported by most recent C implementations, and the GCC implementations on Intel-compatible machines implement it using extended-precision format for memory data as well as for floating point register data. This allows us to take full advantage of the wider range and greater precision provided by the extended-precision format while avoiding the anomalies we have seen in our earlier examples. Unfortunately, this solution comes at a price. GCC uses 12 bytes in 32-bit mode and 16 bytes in 64-bit mode to store a long double, increasing memory consumption by 50–100%. (Although 10 bytes would suffice, it rounds this up to 12 or 16 to give a better memory performance. The same allocation is used on both Linux and Windows machines). Transferring these longer data between registers and memory takes more time, too. Still, this is the best option for programs that want to get the most accurate and predictable results.

Fortunately, newer versions of Intel processors provide direct support for single and double-precision arithmetic. As these processors become more prevalent, and as compilers generate code that uses the newer

floating-point instructions, fewer programmers will encounter the anomalous behavior we have demonstrated here