Lab 5: Floating point

Lab sessions Tue Oct 31 to Thu Nov 02

Lab written by Julie Zelenski and Chris Gregg

Learning goals

In this lab, you will:

1. experiment with float/double types and observe their operational behavior
2. explore the rounding inherent in floating point representation/arithmetic
3. dissect the internals of the floating point format

Find an open computer and somebody new to sit with. Introduce yourself and share your favorite vegetable.

Lab exercises

1. Get started.

Clone the repo by using the command below to create a lab5 directory containing the project files.

```
git clone /afs/ir/class/cs107/repos/lab5/shared lab5
```

Open the lab checkoff form (https://web.stanford.edu/class/cs107/cgi-bin/lab5).

2. Basic behavior of floating point types

Compile and run the `floats` program to answer the following questions about the basic behavior of floating point types:

- What are the values of `FLT_MAX` ? `FLT_MIN` ? `FLT_DIG` ? What are those values for type double?
- Does anything strike you as weird about `FLT_MIN` -vs- `INT_MIN` , which is `-2147483648` ?
- What happens on overflow to a value greater than max? What about divide by zero? How does divide by zero compare to those same operations for integer types? (run the `divbyzero` program to find out!)

3. Float bits

Now, let's go under the hood to look at the raw bits for a float. An IEEE 32-bit float is stored as:

```
N EEEEEEEE SSSSSSSSSSSSSSSSSSSSSSS
```

where N is the sign bit (1 if negative), E is the 8-bit exponent (with a bias of 127), and S is the 23-bit significand, with an implicit leading "1." for values. The sign bit is the bit position of most significance, the final bit of the significand is the least. The `floats` program prints the raw bits for various float values. Run the program and use its output to visualize the bit patterns and answer the following questions. There is also a neat online tool to interactively view float bit patterns (https://www.h-schmidt.net/FloatConverter/IEEE754.html).

- What is the bit pattern for `FLT_MAX` ? `FLT_MIN` ? the value 1.0? the value 1.75?
- How can you identify an exceptional value by its bit pattern?
- How can you identify a negative value by its bit pattern?
- How does a float's bit pattern change when multiplied by 2?

- Consider the float values that represent exact powers of 2. What do all their bit patterns have in common?

4. Practice converting normalized floats to decimal

As we discussed in class, converting floats to decimal is a mechanical process. Normalized floats have an exponent value that is neither all 0s nor all 1s. The steps to convert a normalized 32-bit float to decimal are as follows (using the example `0 01111110 01000000000000000000000`):

1. Determine the sign (`n`) from the most significant bit (0 is positive, 1 is negative).
2. Convert the 8-bit binary exponent to decimal, and subtract the bias (127 for 32-bit floats). E.g., `01111110` is 126 in decimal, so the actual exponent (`e`) is `126–127 == –1` (you can use this converter website (http://web.stanford.edu/class/cs107/float/convert.html) to ease the process).
3. Intrepret the final 23 bit significand (`s`) as a binary number 1.xxxx (e.g., `01000000000000000000000` is intrepreted as `1.01000000000000000000000`. Convert this value to decimal (you can use the same site as above). For this example, `s` = 1.25.
4. Apply the formula: Decimal Value = $(-1)^{n} \times s \times 2^{e}$. For our example: $(-1)^{0} \times 1.25 \times 2^{-1}$ = `0.`625.
5. You can check your answer at this website (https://www.h-schmidt.net/FloatConverter/IEEE754.html), where you can type in the binary value (e.g., `00111111001000000000000000000000`) into the "Binary Representation" field, and hit the `enter` key.

Fill in the table below on the lab checkoff form:

Binary Value	Sign	Un-biased exponent in decimal	Decimal value of 1.significand	Decimal Value
0 10000011 10110000000000000000000				
1 10001001 11001000010001000000000				

5. Converting decimal to IEEE floating point representation

To convert decimals to floats, you basically reverse the process above. We will only worry about creating normalized floats. Follow the steps below to convert 0.4 decimal to a 32-bit float:

1. Convert the decimal value to a binary representation. This is easiest with the converter from above (http://web.stanford.edu/class/cs107/float/convert.html). Most often, this will result in a non-terminating binary value. For our example, `0.4d` = `0.0110011001100110011001100110011001100110011001100110011001100110011001b...`.
2. Find the most significant 1 in the binary value, and use the following 23 bits after that 1. We may have to round, and the most common method is "round-to-nearest," which adds 1 to the least significant bit if the truncated bits would have contributed more than one-half. In our example, `10011001100110011001100` comprise the 23 bits after the first 1, and this leaves `11001100...` remaining, which is bigger than half (0.1 is 1/2). Therefore, we round up, and the significand is `10011001100110011001101`.
3. To determine the exponent, figure out how many places the binary value from step (1) needs to be shifted to the **left** to produce a binary value in the range `1.`xxxxxx. This number is the unbiased exponent. To bias it, add 127 to the number (for 32-bit floats), and convert to binary (you can use the same calculator as in step (1)). For our example, we need to left shift -2 places, so our exponent will be -2 + 127 = `125`, or 1111101 in binary
4. If the sign of the number is positive, the most significant bit will be 0, otherwise it will be 1.
5. Concatenate the bits to get the float representation: `0 1111101 10011001100110011001101`.

6. If you check your answer, you might find that we have lost some information! See the next section for the reason why.

6. Limits of a finite representation (nearest representable floats)

As a finite representation trying to model values drawn from an infinite range, floating point is inherently a compromise. Although some values are exactly representable, many others will have to approximated. Let's drill down a bit to understand what precision you can expect. We will see that it is the number of bits dedicated to the significand that dictates the precision of what can be stored/computed.

- *Rounding during assignment from a constant*. Many constants cannot be assigned to a float without loss of precision. Floating point representation is based on binary decimal. If a given constant does not terminate when expressed as a binary decimal, it will have to be approximated. Consider the constant `0.4` we looked at in Part 5. This is `4/10`, or, in binary, `100/1010`. Apply division to that binary fraction and you'll get a repeating binary decimal $0.0\overline{1100}$. There is no terminating sequence of powers-of-two that sums exactly to `0.4`, so no hope of exactly representing this value! Let's follow what happens during these assignment:

```
    float f = 0.4;  // not exact,  f rounded to nearest representable float
```

In Part 5, we were only able to use 23 bits of our significand (well, 24 because of the implicit leading 1), and we had to round. This left some information out of the representation.

Some constants are rounded because the binary decimal expansion (though terminating) requires more significand bits than are available in the data type. Consider these assignments:

```
        // has terminating binary, but cannot be exactly represented as float
        float f_inexact = 900000000.25;

        // exact, 9.25 can be exactly represented as float
        float f_exact = 9.25;
```

Convert the values to their floating point values, using the steps in part 5. You should find out that the first is not representable in 32-bit floating point format, but the second is.

7. Floating point arithmetic

An important lesson that comes from studying floats is learning why/how to arrange calculations to minimize rounding and loss of precision. Read over the code in the `arith.c` program. Execute it and observe the result of the `sum_doubles` function to see another surprising outcome. A sequence of doubles is summed forward and again backwards. The same values are included in the sum, but the order of operations is different. The result is not the same, and not just epsilon different either! Has the law of associativity been repealed? Explain what is going on. The relative error for floating point multiplication and division is small, but addition and subtraction have some gotchas to watch out for. For example, both addition and subtraction cannot maintain precision when applied to operands of very different magnitude. Two operands very close in magnitude can also cause a problem for subtraction or addition of values with opposite sign. If two floats have the same exponent and agree in the leading 20 bits of the significand, then only few bits of precision will remain after subtracting the two, and leaving just those few digits that were least significant too---double bummer! This event is called *catastrophic cancellation* (the catastrophe is so-called because you lose a large amount of significant digits all at once, unlike roundoff error which occurs more gradually in the bits of low significance). A subtraction of nearly equal values ideally should be reformulated to avoid this cancellation.

8. Test BlueBook

If you brought your laptop, you can test the BlueBook program that we will be using for the midterm exam. Download this program (http://cs107.stanford.edu/BlueBook_MidtermPre.zip), unzip it completely, and then run the BlueBook.jar program. If you do not have a Java runtime, you may need to download it. If you are on a Mac and you get a message that says, "App can't be opened because it is from an unidentified developer," right-click on the exam and select "Open With -> Jar Launcher."

Once you have opened up BlueBook, you will see a login screen. Put your Name and Stanford SUNet ID, and then click the honor code checkbox. Finally, scroll down and enter the password, which is `examexam`. At this point, you should be able to see the practice problems. When you are satisfied that BlueBook is working, go ahead and click the "Finish Exam" button, and submit your practice, with your SUNet ID and password (i.e., the password you log onto Myth with). You should wait until you see an "Exam Submitted" message to ensure that you have properly submitted.

Fun and interesting further reading on floats:

- The classic article What every computer scientist should know about floating-point arithmetic (http://download.oracle.com/docs/cd/E19957-01/806-3568/ncg_goldberg.html). (how's that for a title that makes you feel compelled to read it?)
- A little bit of history on the 1994 Pentium floating-point bug (http://horstmann.com/unblog/2011-07-29/Capture-ch02_jfe2.png)that led to a half-billion dollar chip recall. Thanks to Cay Horstmann for this excerpt.
- An excellent blog series on floating point intricacies (http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/) written by Bruce Dawson.
- Can you get rich by capitalizing on floating-point roundoff? More on the great salami slicing scam (http://en.wikipedia.org/wiki/Salami_slicing).

Check off with TA

Before you leave, be sure to submit your checkoff sheet and have lab TA come by and approve it so you will be properly credited for lab. If you don't finish all the exercises in the lab period, we encourage you to work through any remainder on your own. Double-check your progress with self check (/class/cs107/selfcheck.html#lab5).