CS 213, Fall 1998
Homework Assignment H3
Assigned: Oct. 1, Due: Wed., Oct. 7, 11:59PM

Kip Walker (`kwalker+ta@cs.cmu.edu`) is the lead person for this assignment.

The purpose of this assignment is to become familiar with floating point number representation.

Introduction

Similar to the way two's-complement defines an encoding for storing negative numbers in bit strings, the IEEE Floating Point standard describes encodings for storing non-integer values. Most modern processors support this standard and provide assembly instructions for manipulating floating point numbers of two sizes: single and double precision (32-bit and 64-bit respectively).

In this assignment, we will work with two smaller floating point formats based on the IEEE standard: little and tiny (8-bit and 6-bit respectively). These two formats follow the same encoding conventions as IEEE single and double precision numbers: `NaN`s, $\pm\infty$, normalized, and denormalized numbers are represented as described in the class lecture notes from Sept. 24.

There are 4 sets of problems, a total of 22 questions - each worth 1 point.

Logistics

You may work in a group of up to 2 people in solving the problems for this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the Web page `assigns.html` in the class WWW directory.

All files for this assignment are in the directory:

```
/afs/cs.cmu.edu/academic/class/15213-f98/H3
```

First create a (protected) directory to work in, and copy our template code using the command `tar -xvf /afs/cs.cmu.edu/academic/class/15213-f98/H3/H3.tar`. This will create the files `Makefile`, `h3.c`, `h3.h`, and `soln.c`. In this assignment you will modify `soln.c` as described later, and hand it in with the command `make handin NAME=yourname` where `yourname` is replaced with the Andrew Id of the first person on your team. Your solution will be copied to the handin directory. If you need to submit another version to fix a bug, use the command

```
make handin NAME=yourname VERSION=versionnumber
```

where `versionnumber` starts at 2 and counts up with each submission *after* the initial handin.

To check that you've formatted your answers correctly, run the command `make checker` and then run `./checker`, the resulting program. The output of this program shows you the way we'll interpret your answers when grading (but doesn't tell you whether or not your answer is correct). Please check your solution and fix any mistakes before turning it in.

Problems

Our scaled down versions of the IEEE Floating Point Format are defined as follows:

Little format

- There is a sign bit in the most significant bit.
- The next four bits are the exponent. The exponent bias is 7.
- The last three bits are the significand.

Tiny format

- There is a sign bit in the most significant bit.
- The next three bits are the exponent. The exponent bias is 3.
- The last two bits are the significand.

Otherwise, the rules are like those in the IEEE standard (normalized, denormalized, representation of 0, infinity, and NAN). Refer to the Sept. 24 class handout.

In the four problems below you will calculate the extreme values representable in Little and Tiny Formats, encode and decode Little Format numbers, and convert Little numbers to Tiny numbers. The file `soln.c` has four C arrays of strings with one empty string for each question. Fill in the empty strings with your answers, formatted like the examples in that file.

Problem 1 – Special Values

For *both* formats, give the following values:

- Largest positive finite number
- Positive normalized number closest to zero
- Largest positive denormalized number
- Positive denormalized number closest to zero

Problem 2 – Encoding "Little" numbers

Encode the following values as 8-bit Little floating point numbers: $\frac{3}{4}$, -$\frac{13}{16}$, 44, and -104

Problem 3 – Decoding "Little" numbers

Determine the values (real number, $\pm\infty$, or `NaN`) corresponding to the following Little-number bit patterns (MSB on left):

- 10110011
- 01111010
- 10010001
- 01001111
- 11000001

Problem 4 – Converting "Little" numbers to "Tiny" numbers

Convert the following 8-bit Littles to 6-bit Tinies. Overflow should yield $\pm\infty$, underflow should yield ± 0.0, and rounding should follow the "round-to-nearest-even" tie-breaking rule covered in class.

- 00010000
- 11101001
- 00110011
- 11001110
- 11000101