CIS 450 -- Computer Organization and Architecture – Spring 2018

Programming Lab Assignment 1: Data Representation Lab

Lab Dates: Jan. 26, Feb. 2, Due: Monday, Feb. 5, 2018, 11:59 pm

(50 points)

1. Problem Statement

This assignment deals with analyzing signed and unsigned integers and the IEEE 754 floating point representation of various values and expressions involving different types.

2. Logistics

The only file to be submitted via K-State OnLine will be your answers in a plain ASCII text file (lab1.txt) include a single blank line between the solutions for Part 1 and Part 2.

3. Instructions

To help with this lab, I've posted two programs, `puzzles.c` and `fdump.c` in a single gzipped tape archive (tar) file called `/pub/cis450/programs/Lab1.tgz`. Copy this file to your own home directory and extract the contents using the following commands. Note that commands you type are in **boldface**, with comments in *italics*.

```
cd ~              change directory (go) to home directory (~)
mkdir cis450      make a subdirectory for this class
cd cis450         go to the subdirectory ~/cis450
```

Then, you can copy and unzip the gzipped, tar file in your new class subdirectory using the commands:

```
cp /pub/cis450/programs/Lab1.tgz .
                  copy Lab1.tgz to the current working directory (.)
tar xvzf Lab1.tgz
                  uncompress (z) and extract (x) the files (f)
                  from the archive in verbose mode (v).
```

This will extract the files: `puzzles.c`, `fdump.c`, and `Makefile` into a new subdirectory called Lab1. The Makefile contains a list of commands that can be used to compile the source code (*.c files) into executables.

```
cd Lab1           go to the subdirectory ~/cis450/Lab1
less Makefile     view the contents of Makefile
```

You can use the command **`more`** or **`less`** as a pager to view the contents of a file one page at a time, in this case **`less`** is more ;-) (a bit of system programmer humor). Use the command **`make`** to cause the commands under "all:" in the Makefile to be executed. The first line of each section is the set of dependencies, followed by lines with commands; for example, to "make all" mean that you want to make fdump and make puzzles, and to make puzzles (which depends on puzzles.c) means that you want to invoke the compiler command **`gcc –o puzzles puzzles.c`**, compile the source code in `puzzles.c` and send the output (-o) to a file called `puzzles`. System developers don't like to type any more than necessary, so makefiles were invented to reduce the amount of typing required; e.g., we only have to type "make" instead of "gcc -o puzzles puzzles.c" and "gcc -o fdump fdump.c". Finally, use **`make clean`** to remove the executable files. Of course if you clean your directory, you will need to type "make" or "make all" to compile the code again. In Cygwin or Mingw, the files created are fdump.exe and puzzles.exe.

Part 1: Integer Puzzles

The program `puzzles.c` contains a set of integer puzzles to be solved. Solving a puzzle involves finding a set of inputs that serve as a **counter-example** for a particular logical expression; for example, uy > -1, is false for almost any input because the comparison is performed using unsigned comparison. If the user enters 5 for x and 6 for y, then uy = (unsigned) y = 6 as well, and we have an unsigned value on the left being compared with a signed value on the right, so the type of comparison in unsigned. Consequently, the right-hand side is treated like an unsigned value, and -1 is stored as all 1's which is the largest unsigned value. So, uy is not greater than 2^{32} -1, and the expression evaluates to "false" and you have one puzzle solved. To terminate input from standard input (the keyboard) you can use a **<ctrl>-d**; that is, hold down the control key and press d. Then, the program will report how many puzzles have been solved. For example, if we execute puzzles, and enter the input "-4 1" for x and y, respectively, then the following is output:

```
PUZZLES:
 1. (y>>4)<<4 <= y
 2. (x<<4)>>4 <= x
 3. dx * dy == (double) (x*y)
 4. ux == (unsigned) (float) ux
 5. uy == (unsigned) (double) uy
 6. x > y  implies  -x < -y
 7. y * y * y * y >= 0
 8. (int) (ux - uy) == (x-y)
 9. x >= 0  implies  -x <= 0
 10. x <= 0  implies  -x >= 0
 11. y != 0 implies y != -y
 12. ux >> 3 == ux/8
 13. x >> 3 == x/8
 14. dx * dx * dx * dx >= 0.0
 15. x>0 and y>0 implies x*x + y+y > 0
INT_MAX: 2147483647 7FFFFFFF
INT_MIN: -2147483648 80000000
UINT_MAX: -1 FFFFFFFF
Enter x and y: -8 1

DECIMAL:     x = -8, y = 1, ux = 4294967288, uy = 1
HEXIDECIMAL: x = FFFFFFF8, y = 00000001, ux = FFFFFFF8, uy = 00000001
4. ux != (unsigned) (float) ux for ux = 4294967288

Enter x and y: <ctrl>-d
Problems Solved: 4
Number Solved: 1
```

User input is in boldface above. To keep your life simple, save all of your solutions, with one pair of input values on each line, in a text file called **`lab1.txt`** (the given solution is already in the sample input file). Then, you can test your set of solutions by redirecting input to come from the file using:

`./puzzles < lab1.txt`

Part 2: Floating Point

The program `fdump.c` (when compiled and executed) dumps the bits and bytes of the floating point value entered as a command-line argument; e.g., **`gcc –o fdump fdump.c,`** then **`./fdump 1.0`** results in:

```
x = 1.0000000000E+00
3f 80 00 00
Sign Bit: 0
Exponent: 01111111
Mantissa: 00000000000000000000000
```

Recall that the `float` data type is stored using 32 bits, the most significant bit is used for the sign, followed by 8 bits for the exponent and 23 bits for the significand (mantissa). In this example, since 1.0 is stored as 1.00000000000000000 base 2, the significand is all zeros (remember that the leading 1 is implied), and the exponent is biased by 127, so 1.0 x 2^0, an exponent of 0 is stored as 127, which is 01111111 base 2. Other allowable inputs are: NAN = not a number, INFINITY, and fractions which are input by entering the numerator and denominator as separate arguments; e.g., 3/256 is entered using: **`./fdump 3 256,`** resulting in:

```
x = 1.1718750000E-02
3c 40 00 00
Sign Bit: 0
Exponent: 01111000
Mantissa: 10000000000000000000000
```

Note that 3/256 = 3.0 x 2^{-8} (base 10), 3 (base 10) = 11 (base 2), so 3/256 = 1.1 (base 2) x 2^{-7}. Then, -7 + 127 = 120 = 01111000 (base 2), and 1.10000000000000000000 (base 2) is stored as 10000000000000000 (the fractional part of the number is stored).

Part 2 Problems

1. Around 250 B.C., the Greek mathematician Archimedes proved that 223/71 < π < 22/7. With access to the standard math library header file <math.h> located at /usr/include/math.h, he would have been able to determine that the single-precision floating point approximation of π has a hexadecimal representation of 0x40490FDB.

   Example: How are the bits representing 223/71 stored:

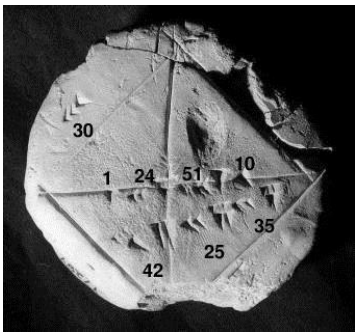
   Solution : **./fdump 223 71**
   x = 3.1408450603E+00
   40 49 03 9b
   Sign Bit: 0
   Exponent: 10000000
   Mantissa: 10010010000001110011011

   So, the value stored is: 11.00100100000011100110 11

   a. What is the fractional binary number stored for 22/7? _____

   b. What is the fractional binary number denoted by the hex representation of 0x40490FDB?
   Hint: the fdump program also accepts input in hexadecimal form, e.g., **fdump 0x40490FDB**

2. Floating Point Problems:



   a. The Babylonian clay tablet YBC 7289 (c. 1800-1600 BC) shown to the right, gives an approximation to the square root of 2 in four sexagesimal figures: 1 + 24/60 + 51/602 + 10/603 = 30547/21600.

      What is the binary value stored for 30547/21600?

      A more precise representation is stored in the header file <math.h> as M_SQRT2. The output can be displayed using **fdump M_SQRT2**.

   b. Another early close approximation is given in ancient Indian mathematical texts, the Sulbasutras (c. 800–200 BC) as follows: *Increase the length [of the side] by its third and this third by its own fourth less the thirty-fourth part of that fourth.*[2] That is, 1 + 1/3 + 1/(3*4) – 1/(3*4*34) = 577/408. What is the binary value stored?

   c. How do the two approximations in a. and b. compare with the actual value? Are they too small or too large?

   d. How is the number 7.75 stored internally in base 2? Can this number be represented exactly as a float or double?

   e. In decimal form, what is the largest odd integer that can be represented exactly as a **float**? What is the largest odd integer that can be represented exactly as a **double**?

   f. In decimal form, what is the largest even integer that can be represented exactly as a **float**? What is the largest even integer that can be represented exactly as a **double**?

   g. In decimal form, what is the largest positive normalized value that can be represented as a **double**?

   h. In decimal form, what is the largest positive denormalized value that can be represented as a **double**?

   i. In decimal form, what is the largest negative denormalized value (closest to zero) that can be represented as a **double**?

      Hint: recall that the largest denormalized value has it exponent represented as all zeroes, E for denormalized values is 1-*Bias* instead of 0-*Bias* which it would be for normalized values. This is to allow for a smooth transition between denormalized and normalized values.

LAB CONTEST: The student who solves the most puzzles with the fewest inputs and the student who solves the most puzzles with a single input (which should be listed as your first input in lab1.txt, note that -4 1 is not the best ;-) will win the prizes for this Lab – prizes are usually candy bars, etc.

Notes: Remember to include **all** of your inputs in lab1.txt, just put your best single puzzle solution on the first line. You should have several input pairs for Part 1, one on each line in the input file. Just leave a blank line at the end of the input pairs from Part 1 before adding your solutions to Part 2; e.g., sample input is shown in lab1.txt, below I added a few more lines:

```
-8 1
          ← remember to include the blank line between parts
1a. The fractional binary number stored for 22/7 is 11.001001001001001001001.
1b. ..
```

To edit the input file, you can use a number of different text file editors in Linux including: nano, pico, vi, etc. or refer to the files in your home folder (U: drive). Use **putty** to remote into a CS Linux machine; e.g., putty cislinux.cs.ksu.edu, and use **FileZilla** to move files between Unix and Windows, or just access your home directory.