

Study Questions for Patterson & Hennessey, Section 3.6

To paraphrase Matt Hurne from the Spring 2005 section of this course: “I had no idea that there were gaps in the floating point number line!”

It's insights like these that show you how assembly language ideas can be valuable even for those who will never program in assembly language themselves. We are talking about a number system that is not the one you learned about in mathematics classes. In particular, as we'll see in the first Fallacy in this section (p. 220), addition is not even associative for f.p. numbers

Floating point numbers use both signed magnitude (for the significand) and biased representation (for the exponent). The rationale is simplicity of hardware: The comparisons (like `c.lt.s $f3 $f4`) in MIPS) can be made by the same hardware logic as `slt $t3, $t1, $t2` uses. (Not by the same instruction, since the `slt` instruction executes in the CPU, but the `c.lt.s` instruction executes in the FPU (floating point processing unit) in coprocessor 1.

The exponent for IEEE (standard 754) float is 8 bits, and for double is 11 bits. Know what the biases are, or be able to recalculate them from first principles, given that they are half the maximum size of an unsigned number of the same size..

Overflow means that the exponent is either too positive or too negative to fit in the number of bits allowed. **Underflow** means that a result is closer to 0 than is allowed. In particular, whether there is hardware support for **denormalized** numbers makes a difference about how small a number needs to be in order to underflow. Both overflow and underflow are trapped by the hardware, and so must be handled by the operating system.

Know the table in Figure 3.15 about the IEEE 754 standard for how to represent $+\infty$, $-\infty$, and Nan.

Know how to convert from decimal to binary and binary to decimal, when fractions are involved.

Know (with the help of your green sheet and a table of syscalls) the f.p. commands in MIPS that were discussed in class:

<code>add.{s,d}, sub.{s,d}, mul.{s,d}, div.{s,d}</code>	for arithmetic;
<code>c.{eq, ne, gt, lt, ge, le}.{s,d}</code>	for comparison of floats and doubles;
<code>bc1t, bc1f</code>	for branching as a result of the comparison;
<code>lwcl, swcl, ldc1, sdc1</code>	for loading or storing floats or doubles.

Reading floats and doubles from the keyboard; writing floats and doubles to the screen.

```
.data
pi: .double 3.141592653589793238 # They're too precise, but the
e: .float 2.718281828459045 # assembler does the best that it can.
```

<code>s.s, s.d, l.s, l.d</code>	stores and loads to/from memory
<code>mfc1, mtc1, mtc1.d, mfc1.d</code>	moves between CPU and FPU

```
Example:  l.d    $f0, pi           # pseudo-op to put pi into $f0 and $f1
          addi   $sp, $sp, -8    # make room on stack
          sdc1  $f0, 0($sp)     # store $f0 and $f1 on the stack
```

See also the file `float.s` in directory `Q:\InstructorFiles\Chase_Gene\Assembly\`

You do **not** need to know about the differences between the round, guard, and sticky bits. You do need to know that it is possible to get as precise an answer as if you did the arithmetic in infinite precision, and then rounded to the size of the machine word at the very last moment, given 3 extra bits.