CMPT 295 Assignment 5 (2%)

Submit your solutions by Friday, March 1, 2019 10am.
Remember, when appropriate, to justify your answers.

1. [7 marks] *Floating-Point Integers*

   Let S be the set of 32-bit IEEE floating-point values that are integers. (S contains both $+0$ and -0.)

   Answer the following questions about S. Express all your answers exactly, i.e., by using powers of 2.

   (a) [1 mark] How many elements are in S?

   (b) [1 mark] What's the largest *odd* value of S?

   (c) [1 mark] What are the largest two values of S?

   (d) [1 mark] How many consecutive integers are in S? You shouldn't count both $+0$ and -0 in your total.

   (e) [1 mark] The number 2^{32} is in S. What are its nearest neighbours? In other words, what two values in S are closest to 2^{32}?

   (f) [2 marks] Determine the number of values in S which are representable using 32-bit unsigned.

   (g) [1 BONUS mark] Determine the number of values in S which are representable using 64-bit 2's complement. An exact value is required for the bonus mark.

2. [2 marks] *Floating-Point Addition*

   Add the following pairs of 32-bit IEEE floating-point numbers. Show all your work.

   - `0x43938000` + `0xc2280000`.
   - `0x3f19999a` + `0x3eccccd`.

   Note: The rounding convention for numbers that are exactly halfway between two valid representations is to round to the nearest even number. For example, the decimal number 3.5 rounds to 4, but 4.5 also rounds to 4. In binary, the only even digit is 0, so you will round to 0, if necessary.

3. [11 marks] *Adding Positive Floating-Point Numbers*

   The function `sum_float()` adds an array `F[n]` of *positive* floating point numbers by the usual algorithm of summing to a running total. Because addition is not associative, the relative error generated due to rounding can vary depending on the order in which the numbers are added: in other words, some orders are better than others.

   (a) [2 marks] *C:* The base code in the care package adds together 24 floating-point numbers and displays the result in hex (`0x5060000f`). This result is too low due to the relative magnitudes of the numbers — because the first number (also `0x5060000f`) is about 2^{25} larger than any of the rest of them, the significance of the smaller numbers has been truncated and rounded down.

   More precise is to add the numbers from smallest to largest magnitude. Using the built-in `qsort()`, modify `main()` so that it sorts the array from smallest to largest. To read the docs on `qsort()`, try `man qsort`.

   (The corrected result should be `0x50600015`.)

   (b) [2 marks] *C:* The significance of the round-down error in part (a) occurred in the last 3 bits of the significand. It is possible to have the error occur the other direction, i.e., to end with a result that was too big in the last 3 bits instead of too small. Adjust the two values in your code so that such an error occurs.

(c) [7 marks] *x86-64:* It is possible to do even better by applying the heuristic of adding the two smallest. Though sorting will deprecate the need for a linear search, a better approach is to use a queue `Q` to hold the partial sums in the following algorithm:

```
Q <- empty
for i from 1 to n-1 do:
   x <- smallest of { head(F), head(Q) }
   dequeue(x)
   y <- smallest of { head(F), head(Q) }
   dequeue(y)
   enqueue(x+y) -> Q
return head(Q)
```

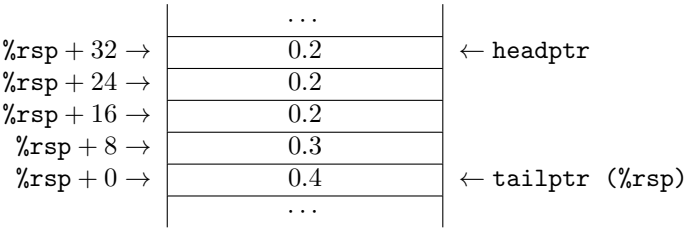
The elements of `Q` will be in increasing order, so you can always find the two smallest values or partial sums by simply comparing the heads of each. (The head of an empty queue is $+\infty$.)

Modify `sum_float()` so that it adopts this heuristic. Your code may assume the array comes in sorted order. There are no other functional requirements other than obeying the function call protocol: pay attention to what is scratch and what must be saved and restored!

(The sum of the second array should be `0x40f00000`.)

Hints:

- To implement a queue in assembly, it is recommended that you use the stack. Yes, a stack follows a last-in-first-out ordering, and for a queue you'll need a first-in-first-out ordering. That only means that you can't use both `push` and `pop` to get the job done. However, using `push` to implement `enqueue` would be a good design choice. Therefore `%rsp` should maintain the address to the tail of the queue.
- You will need to use a register to maintain the address of the head of the queue. What should it be initialized to? How should it be updated on `enqueue/dequeue`? How can you tell if the queue is empty?
- You will need a way to restore the stack to its original state after you are done with the queue.



You will submit:

(a) [5 marks] an electronic copy of your `sum_float.s` assembly source. This code will be tested for correctness with a selection of different inputs.

(b) [4 marks] an electronic copy of your `main.c` C source.

(c) [2 marks] a hard copy of your `sum_float.s` assembly source. Your source should be well documented, so that any other programmer could read your code and understand it. No stack diagram is required.