Homework 1

Integer and Floating Point Number Representations

Integer

Problem 1

Suppose you have a 4 GHz x64 core and you can execute four integer operations (additions or subtractions) every cycle.

1. How long (how many seconds) will the following loop run?

```
uint32_t i;   /* 32 bit unsigned integer */
uint64_t s = 0;   /* 64 bit unsigned integer */
uint64_t z = s-1; /* 64 bit unsigned integer */
for (i = 1 ; i != 0; i++) {
      s += i;
      z -= i;
}
```

2. How many times does s overflow?
3. How many times does z overflow?

Problem 2

How can you compute the following using only shifts, adds, and subtracts? Here, `x` is a `uint32_t`.

```
16 * x
13 * x
39 * x
x / 16
x / 5  (Hard! Outside the scope of class, but included as a challenge)
```

Problem 3

Some instruction sets, including x86, provide an integer representation in addition to two's complement. This representation is called Binary Coded Decimal (BCD). In BCD, a decimal digit (0,1,2,3,4,5,6,7,8,9) is encoded into a group of 4 bits using 0000 through 1001. How many unique numbers can be represented in a 32 bit BCD quantity? Why might one use BCD to represent prices like $10.99 instead of using fixed point binary or floating point?

Problem 4

Many instruction sets have instructions where when you multiply two k bit numbers the result is stored as two k bit numbers. Why? Similarly, many have instructions where if you divide two k bit numbers, the result is stored as two k bit numbers. Why? By "k bit numbers" think of unsigned integers, although the same reasoning holds for signed integers.

Problem 5

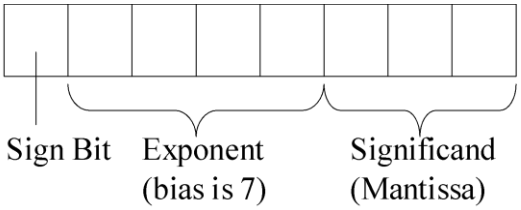
Early computers used Baudot Codes to encode characters. These allowed 5 bits per character, so 32 symbols could be encoded. The UK used one encoding, but another was used for Europe (incl. France, Germany, etc). Why would five bits be the number chosen? And why might countries other than the UK have wanted a different encoding?

New computers use UTF-8, 16, 32 character encodings – the number corresponds to the bitwidth used to encode characters. Why are characters up to 32 bits wide useful?

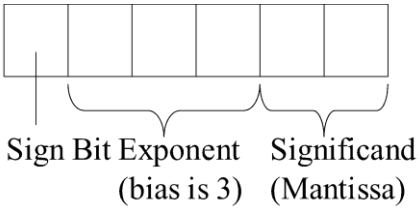
Floating Point

Consider the following two small floating point formats based on the IEEE standard:

- Little Format



- Tiny Format



Except for the sizes of these formats, the rules are those of the IEEE standard.

Problem 1

For both formats, determine the following values (in decimal)

1. Largest negative finite number (by absolute value)
2. Negative normalized number closest to zero
3. Largest positive denormalized number
4. Positive denormalized number closest to zero

Problem 2

Encode the following values in the 8 bit Little Format or report that they cannot be so encoded: 7/4, -1/16, 144, –35, NaN, and infinity. Show each in binary and hexadecimal, if possible.

Problem 3

Determine the values corresponding to the following Little Format bit patterns. The leftmost bit is the most significant

1. 01010010
2. 00000111
3. 01001010
4. 10010000
5. 00111010
6. 11111001

Problem 4

Convert the following 8 bit Little Format numbers into 6 bit Tiny Format numbers. Overflow should yield +/- infinity, underflow should yield +/- 0.0, and rounding should follow the "round-to-nearest-even" tie-breaking rule. Show the bit pattern and its hex representation.

1. 01111001
2. 11111000
3. 01001100
4. 00110011
5. 11001010
6. 01111000
7. 01111111

Problem 5

The changing demands of scientific computation, the growing importance of machine learning computation, and issues with the IEEE floating point standard you are learning have resulted in a range of new alternatives that current vying for attention.[1] For machine learning purposes (specifically neural network "deep learning" computations), it is generally believed that 16 bit floating point is sufficient.

- A 16 bit version of the IEEE standard exists and is widely implemented. A float16 has 5 exponent bits and 10 mantissa bits.
- Brain Floating Point is another important standard (also widely implemented in recent Intel and ARM chips, and originating in Google's TPU chips). A bfloat16 has 8 exponent bits and 7 mantissa bits. It is essentially a 32 bit IEEE floating point number with 16 bits chopped off the end of the mantissa.

What are the comparative advantages and disadvantages of these formats in terms of the real numbers they can represent?

Some proposed alternatives to IEEE floating point encode the split between exponent bits and mantissa bits directly in the number itself. For example, for an n bit number, we might introduce a log2(n) bit field that encodes the position of the last exponent bit. For 16 bits, this field would be 4 bits wide, leaving just 12 bits for the sign, exponent and mantissa. For 32 bits, the field would be 5 bits wide, and so on. What are the advantages and disadvantages of such a scheme? What happens as the bit width of the number increases?

Problem 6

The floating point (and integer, fixed point, bcd, etc) numbers and their operators (+/-/*/…) are generally implemented in hardware for speed. However, software implementations also exist, such as GNU MPFR. While these are much slower, they allow a programmer to use a bit width that the hardware does not support by emulating it using software. For example, the programmer could choose to use 1024 bit floating point numbers or 2048 bit integers. Why might a programmer do this?

(Thought experiment – nothing to hand in) Could you write a program that implements a rational number system? In a rational number system, every value is represented as a fraction (a ratio of two integers).

Problem 7

One proposed representation is to make 8-bit numbers with a 1-bit sign ("s") and a 7-bit significand ("val"), representing $(-1)^s * 2^{\alpha * val}$ where α is chosen on a per-program basis so that the representation is more broadly useful.

[1] Other important examples: unums and posits

1. What kinds of operations might be easier to carry out with this representation, from a computing perspective? What kinds might be harder?

2. What makes it difficult to encode zero in this representation?

3. If you wanted a very similar representation that could encode zero, how could you create such a representation?

Problem 8

1. In the textbook (Section 2.1.4) you are introduced to C-style strings and the `show_bytes` function (Section 2.1.3, Fig 2.4). **What would be printed as a result of the code snippet below?**

```
const char *s = "tonight's the night.";
show_bytes( (byte_pointer) s, strlen(s) + 1  );
```

2. Alternative string representations store relevant metadata associated with raw pointer to the string data, to make up for the short-comings of C-style strings. Think of comparing two strings for equality. How expensive is this function in terms of the length of the shorter string being compared? What are **two** examples of metadata you could add to make it possible to do the comparison in one step, independent of the lengths of the input strings?

```
typedef struct mystring {
    METADATA1;
    METADATA2;
    const char *s;
} mystring_t;

mystring_t a, b;
if (are_same_string(a,b)) { printf("are the same\n"); }
```

3. Simple C-style strings may be slower to compare, but they do offer relevant benefits. Please list and explain at least **three benefits** from using C-style strings versus an alternative string representation (like the one that you created in problem 8.2).