15-213: Introduction to Computer Systems
Written Assignment #1

This written homework covers representation of floating point numbers, bitwise operations, the representation of integer numbers and deep learning (just kidding).

Directions

Complete the question(s) on the following pages with single paragraph answers. These questions are not meant to be particularly long! Once you are done, submit this assignment on Canvas. Below is an example question and answer.
Q: Please describe benefits of two's-complement signed integers versus other approaches.
A: Other representations of signed integers (ones-complement and sign-and-magnitude) have two representations of zero (+0 and −0), which makes testing for a zero result more difficult. Also, addition and subtraction of two's complement signed numbers are done exactly the same as addition and subtraction of unsigned numbers (with wraparound on overflow), which means a CPU can use the same hardware and machine instructions for both.

Grading

Assignments are graded via peer review:

1. Three other students will each provide short, constructive feedback on your assignment, and a score on a scale of 1 to 15. You will receive the maximum of the three scores.
2. You, in turn, will provide feedback and a score for three other students (not the same ones as in part 1). We will provide a rubric, a document describing what good answers to each question look like, to assist you. You receive five additional points for completing all of your peer reviews.

Due Date

This assignment is due on Wednesday, June 1 by 11:59pm Pittsburgh time (currently UTC−4). Remember to convert this time to the timezone you currently reside in.

Question #1

Your ECE friends over at Hamerschlag Hall are looking to implement hardware for a new computing system and have asked you for your help in choosing a specification for their 16-bit floating point value.

In addition to 64-bit (FP64) and 32-bit (FP32) specs, the IEEE 754 standard also specifies a 16-bit (FP16) floating point number. The 16 bits are divided as follows:

- 1 sign bit
- 5 EXP bits
- 10 FRAC bits

Google Brain, however, created their own Brain Floating Point Format (BFLOAT16) for use in their deep learning systems. The 16 bits are divided as follows:

- 1 sign bit
- 8 EXP bits
- 7 FRAC bits

a. Describe the tradeoffs between the FP16 and BFLOAT16 formats, i.e. for the ranges (largest and smallest positive values) and step size (distance between neighboring numbers). No need to calculate anything, just a qualitative explanation using the specs of each format.

b. List any problem(s) that there might be with converting certain numbers in FP16 to BFLOAT16 and vice versa.

c. Now think about how converting from FP16 to FP32 would work. What would you need to do to the EXP field and FRAC fields of the FP16 number?

d. Google Brain was formed in 2011 to leverage massive computing resources to perform deep learning research. Knowing that they need to do a ton of number conversions, why do you think they chose to create their own 16-bit floating point number that uses exactly 8 EXP bits? (Hint: How many EXP bits does FP32 have?)

a. The BFLOAT16 format has a larger range than the FP16 format because it has more EXP bits. Since FP16 has more frac bits, it can have a smaller step size in between the numbers compared to BFLOAT16.
b. FP16 -> BFLOAT16: There would have to be rounding for numbers that have too much precision in FP16. BFLOAT16 -> FP16: You couldn't represent numbers that are too far out of the range of FP16's range.
c. The EXP field would have to be scaled up since the Bias would increase, and then the FRAC field would have to have zeros appended. Something similar/close to this is appropriate.
d. So that they can convert quicker from 16-bit to 32-bit floats; no scaling of the EXP field would be necessary; because the 32-bit floats also have 8 EXP bits so converting is a lot quicker. Something to do with easier number conversions from 16 to 32 bits.

Question #2

In class and in Data Lab, you will learn and interact a lot with the bitwise Boolean arithmetic operators provided by C, & | ^ ~, as well as some useful tools for manipulating Boolean formulas, such as De Morgan's Laws.

a. Write a formula that calculates a & b using only the ~ and | operators. (Hint: this is one of De Morgan's Laws.)

a & b == ~(~a | ~b)

b. Write a formula that calculates a | b using only ~ and & operations. (Hint: this is the other one.)

a | b == ~(~a & ~b)

c. Write a formula that calculates a ^ b using only ~, &, and | operations. (Hint: this is *not* one of De Morgan's Laws.)

a ^ b == (~a & b) | (a & ~b) == (a | b) & (~a | ~b)
There are several other equivalent formulas.

d. As you can see from the previous three questions, it's not technically necessary to have all of the Boolean operators in your programming language; if you had only AND and NOT, you could construct all of the others from them. This is called "functional completeness." There are two special Boolean operators that are functionally complete all by themselves: if you have either NAND (a NAND b == ~(a & b)) or NOR (a NOR b == ~(a | b)) then you can construct all the other Boolean operators from them.

Write formulas that compute ~a, a | b, and a & b using only NOR. As shorthand, you can use a $ b to mean a NOR b.

~a == ~(a | a) == a $ a

a | b == ~(~(a | b)) == ~(a $ b) == (a $ b) $ (a $ b)

a & b == ~(~a) & ~(~b) == ~(~a | ~b) == ~a $ ~b == (a $ a) $ (b $ b)

Students do not need to show the derivation, only the formula at the far right for each.

Question #3

Before the twos-complement representation of signed binary numbers was invented, some computers used *sign-and-magnitude* representation. In this representation, the most significant bit is *only* the sign of the number (0 for positive, 1 for negative); it has no place value. The remaining bits are interpreted as an unsigned number. For instance, the four-bit number 1011 in sign-and-magnitude represents -3, and 0101 represents 5.

a. If you add the four-bit sign-and-magnitude numbers 0100 and 0101 using unsigned addition, what is the value of the result, interpreted as sign-and-magnitude?

1001 = -1

b. If you add the four-bit sign-and-magnitude numbers 0010 and 1001 using unsigned addition, what is the value of the result, interpreted as sign-and-magnitude?

1011 = -3

c. If you add the four-bit sign-and-magnitude numbers 1010 and 1001 using unsigned addition, what is the value of the result, interpreted as sign-and-magnitude?

0011 = 3

d. Describe an algorithm for adding sign-and-magnitude numbers which will make sums involving negative numbers come out correctly. For instance 0010 + 1001 should produce 0001 (1), and 1010 + 1001 should produce 1011 (-3). Don't worry about what happens on overflow in either direction.

1. If both numbers are positive, just do an unsigned addition.
2. If both numbers are negative, clear both numbers' sign bits, do an unsigned addition, then set the sign bit of the result.
3. The remaining case is that one number, **a**, is negative and the other, **b**, is positive:
    a. Clear **a**'s sign bit.
    b. If **a** > **b**, subtract **b** from **a** and set the sign bit of the result.
    c. Otherwise, subtract **a** from **b**.