Name: \_\_\_\_\_\_\_Phung, Hannah \_\_\_\_\_\_\_\_\_ Date: Mar 8, 2022\_\_\_\_\_

(last name, first name)

NYU ID: N18096810

NetID: hmp3554

Section: 007

**Lab 1**

**Total in points** (100 points total): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Professor’s Comments:**

Github link:

**Report:**

1. Bits Manipulations:

bitXor (int x, int y):

Method: for this exclusive-or, I used the truth table to figure out the equivalent statement using only ~ and & before coding

| x | y | x&y | ~(x&y) | ~x | ~y | (~x)&(~y) | ~((~x)&(~y)) | (~(x&y))&(~((~x)&(~y))) | x^y |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

allOddBits (int x):

Since we are only allowed to use constants of 8 bits, I used 170, which is 1010 1010. All odd-numbered bits in the binary representation of this number are 1’s, and all the even-numbered bits are 0’s. Then, I used left-shift and 170 to get all odd-numbered bits 1’s and all even-numbered bits 0’s for all 32 bits. If (x & this number) == this number, x has all odd-numbered bits set to 1.

isAsciiDigit (int x): For x to be an Ascii digit, it has to be in the range of

Since the Ascii representation of digit 0 to 9 is always positive, we can first check if x is negative by using right-shift and logical negation.

Then, I used the same idea of less than or equal to (like the function below) to check for two conditions: x less than or equal to 0x39 and x greater than or equal to 0x30

conditional (int x, int y, int z):

x ? y : z means that if x is not equal to 0, then y, else then z. First I used two logical negation !! to turn x into boolean, then used ~ and +1 to manipulate this boolean. My goal was to have (1…111 & y) and (0…000 & z) if x is not zero, and (0…000 & y) and (1…111 & z) if z is zero

logicalNeg (int x):

If x is zero then 0 >> 31 (all 0 bits) | (~0 + 1) >> 31 (all 0 bits) would give us 0. This is because x and -x are the same when x is zero (0 = -0)

If it is positive then x >> 31 (all 0 bits) | (~x + 1) >> 31 (all 1 bits) would give -1 (all 1 bits)

In both cases, we can plus 1 to achieve our desired result (0+1 = 1 and -1 + 1 = 0)

Case for x is negative is similar to positive but reverse

1. Two’s Complement Arithmetic:

tmin ():

The minimum two’s complement integer would start with 1 bit in position 31, then follow by all 0’s in the remaining 31 bits from position 0 to 30

Since we are allowed to use small constant, we can use the operator left shift << and 1, which has 31 0's bits in position 1 to 31 except for the last one in position 0

isTmax (int x):

Initially, my method was finding a number which its sum and Tmax add up to 0 (all bits 0’s) or -1 (all bits are 1’s) so I can use the ~ operator and the logical negative ! operator. I thought only when x is Tmax then ~(x ^(x + 1)) = 0.

Challenge: however, there is the case of -1 (all 32 bits are 1’s) which also have the same property. Hence, I figured out a way to slightly manipulate the expression so that x + !(x + 1) = x only when x = Tmax but not when x = -1

negate (int x): I used the definition of two’s complement for this one. No real challenge.

isLessOrEqual (int x, int y):

There are two cases:

Case 1: x and y have the same sign (both greater than or equal to 0 or both less than 0)

Case 2: x and y have different signs (in order for this function to return 1, x has to be less than 0 and y has to be greater than or equal to 0.

Challenge: Case 1 was not difficult because I could negate x using the same idea as the above negate (x) function and sum this with y (if greater than or equal to 0 then true). However, I needed to figure out not only how to get the sign of x, but knowing if x and y have different signs. If they are different signs and x is negative then return 1.

howManyBits (x): It took me a while to figure out how to do this question without using loops, although I know that I would be doing a lot of shifting. In the end, I figured out a way of implementing the idea of binary search to locate the leftmost significant bit (the bit for the sign is added separately later so first I had to invert the number if it is negative). It took me several trials to get the shifting part work accurately for binary search (I got an off-by-one error in the beginning).

1. Floating-Point Operations

float\_twice (unsigned uf):

Challenge: making sure that I have covered all the cases. Also, a friend introduced me to the idea of masking (which I have used before but didn’t know how to utilize it properly), so I was able to separate the sign and exp for later questions. The key point was knowing the shifting (mainly 31 positions and 23 positions) to get the desired components of the floating point.

Case 1: Denormalized case where exp are all zeros

Case 2: When exp = 1111 1110 and multiplying uf by 2 would create an overflow/make this infinite/special number. Then this should return exp = 1111 1111 and all frac bits set as 0’s

Case 3: Normalized except for when exp is 1111 1110 (normalized excluding case 2)

Case 4: Special values (all frac bits are 0’s) and NaN (not all frac bits are 0’s) with exp = 1111 1111 would return the original argument.

float\_i2f (int x):

For this function, I used the idea of masking mentioned above to get the sign. Then, I used the definition of exp = E + bias = E + 127. In this case, I get the maximum E which is 31 + 127 = 158 to do the loop later and continuously divide the number by 2 (using left-shift << 1 for more convenience) until the number left of the decimal point is 1, then we stop.

For the case when x = 0, then floating point representation of x is also 0

For the case of negative x when the sign is 1, we would need to negate x before doing the conversion.

float\_f2i (unsigned uf):

For this problem, I also used the idea of masking mentioned above and shifting to get the sign. Same with finding exp but with >> 23 and 0xFF. I then used the fact that this is a 32-bit representation to define bias = 127, then used the definition of exp = E + bias to calculate E. To find the frac, I also used the idea of masking with 0x7FFFFF.

There are several cases of uf:

Case 1: NaN and special values. Special values (all frac bits are 0’s) and NaN (not all frac bits are 0’s) with exp = 1111 1111 would return 0x80000000u.

Case 2: E < 0 or exp < bias: Since , return 0

Case 3: Overflow/out of range case where E >= 31 would also return 0x80000000u

Case 4: Normalized number: shift left E - 23 if E > 22, shift right 23 - E if E <= 22.

Finally, we have to add back the 1 that the mantissa assume on the left of the decimal point

And if it is a negative number, we have to negate it.