Name \_\_Zhao, Rena\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_03/09/22\_\_\_\_\_\_\_\_\_\_

NetID: \_\_ryz215\_\_\_\_\_\_ Section: \_\_007\_\_\_\_\_\_\_\_\_

**Shape

Description automatically generated with medium confidence**

I have completed this assignment independently:

**Lab 1**

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

**Professor’s Comments:**

GitHub Link: <https://github.com/ryzhao123/cso-sp22/blob/main/bits.c>

1. bitXor
   1. We can expand x ^ y into (x & ~y) | (~x & y) using the definition of exclusive-or, which is that the expression is true IFF x and y are different. To eliminate the “or” operator, we can negate the expression twice, the first time expanding using DeMorgan’s law, and the second time without expanding. We now get ~(~(x & ~y) & ~(~x & y)).
2. tmin
   1. The minimum Two’s Complement integer is represented in binary with the leftmost bit equaling to 1 and all the bits to the right are 0s. We can express this with a left shift of 31 places.
3. isTmax
   1. We can pretend that x equals TMax. Using overflow properties, we can see that TMax ^ (TMax + 1) is 1111…, which means that ~(x ^ (x + 1)) is 0. However, we get this result for x equals TMax and x equals -1, so to remove the possibility of the latter also having the same result as TMax, we use the fact that -1 + 1 = 0. This gives us a second criteria of !(x + 1), where if x = -1, this criteria results in 1. Putting the two parts together, we get (~(x ^ (x + 1)) + !(x + 1)), which becomes 0 + 0 if x = TMax, and the negation of it all returns 1 IFF x = TMax.
4. allOddBits
   1. Create a 32-bit vector of alternating 1s and 0s (101010…) to test inputs of x against. This is equal to 0xAAAAAAAA; however, the decimal representation is not allowed due to this problem’s restrictions. (Line 174) To follow the restrictions, use the decimal representation of 0xAA (10101010 in binary), which is 170, and left shift it 8 places and add 170 again to get 16 spaces of alternating 1s and 0s. Keep doing this with a left shift of 16 and 24. This now gives us 0xAAAAAAAA to compare with. (Line 175) To verify there are 1s in all the odd bits (doesn’t matter what’s in the even bits), we use the & operator on x and compare. If so, it will return 0xAAAAAAAA. Further using the ^ operator on this result will return 0 IFF the (x & compare) part is being compared to itself, and otherwise if not. Then, the logical NOT operator ! will make the 0 into a 1, and turn any other result into a 0.
5. negate
   1. Using the definition of Two’s Complement, to get the two’s complement negative notation of an int, first write out the number in binary, then invert the digits, then add 1 to the result. We take the bitwise NOT operator ~ to invert all the bits in x, then add 1 to the result.
6. isAsciiDigit
   1. We must verify that (0x39 – x >= 0) and (x – 0x30 >= 0). This can be represented with Two’s Complement negative notation, which would turn the previous statements into (0x39 + (~x + 1) >= 0) and (x + (~0x30 + 1) >= 0). Now, to determine if both statements are true, we can use arithmetic right shift rules with negative numbers. If the left sides of the condition statements are positive or 0, a right shift of 31 would result in 0x0. If the left sides of the condition statements are negative, a right shift of 31 would result in 0xFFFFFFFF since the sign bit 1 is preserved. Therefore, if both conditions are met, they result in 0x0. The negation of each statement with the & operator will return 1 if x is valid.
7. conditional
   1. Since x is any int, using !x can turn it into only either 0 or 1 for convenience in this problem. Do the opposite of if we were using x: if !x is 0 (x is true), return y, and if !x is 1 (x is false), return z. Using a two’s complement statement where condition = ~(!x) + 1, if !x is 0, we get 0, and if !x is 1, we get 0xFFFFFFFF. This makes it so ~condition & y will return y and condition & z will return 0 if !x is 0 (therefore x is true). If !x is 1 (therefore x is false), then ~condition & y will return 0 and condition & z will return z. Therefore, (~condition & y) | (condition & z) is the representation of the conditional.
8. isLessOrEqual
   1. The goal is to return 1 if (y – x) >= 0. We first find the signs of x and y using right shifts so that the sign bit is preserved: if x and/or y is non-negative, it will return 0, and if x and/or y is negative, it will return 0xFFFFFFFF. We do the same to find the sign bit of the difference y – x. The possible combinations of valid results are if x and y have different signs and x is negative (y is positive), or if the difference of y – x is positive and x and y have the same sign.
9. logicalNeg
   1. We need to get that if x = 0, return 1 and otherwise 0. Using the two’s complement definition, (~x + 1) = x IFF x = 0, and for other numbers, the sign bits are different. With the expression ((x | (~x + 1)) >> 31), we get that if x = 0, the result is 0, and if x is not equal to 0, the result is 0xFFFFFFFF. The | operator ensures that it will return all 1s when used on a number other than 0 and its two’s complement, and the right shift fills up all spaces with 1s since the sign bit is guaranteed to be 1. The result of the | operator when both variables are 0 is 0. Then, adding 1 gives the desired result: if x = 0, then return 1 and otherwise 0 (0xFFFFFFFF + 1 = -1 + 1 = 0).
10. howManyBits
    1. For positive numbers, we need to find the first instance of 1, and for negative numbers, we need to find the first instance of 0. For convenience in this problem, we only need to find the first instance of 1 by flipping the bits of negative numbers with x = (sign & ~x) | (~sign & x). If x is negative, the first part will be significant, and if x is positive, the second part will be significant. Now we almost perform a binary search on the bits. We look for if the most significant 16 bits have at least one 1. If so, bits16 = 16 since a left shift of 4 is basically 24, and if not, bits16 = 0. The double negation !! ensures that the result of (x >> 16) is either 0 or 1. If bits16 = 16, right shift x by 16 bits so we can search within the leftmost and most significant 16 bits. If bits16 = 0, no right shift is performed since the first 1 is on the right 16 bits. Repeat these steps with bits8, bits4, bits2, bit1, and bit0. Summing up these results and adding 1 for the sign bit will give us the minimum number of bits needed.
11. float\_twice
    1. We first determine the sign bit by conducting a bitwise & operation on uf and 0x80000000, then get the exponent bits by conducting a right shift of 23 on uf to get the leftmost 9 bits (8 are exponent, 1 is sign), then the bitwise & with 255 ensures that we focus on the 8 exponent bits. We then must test a couple input-output cases:
       1. (1) Infinity or NaN (exponent is 11111111, or 255 in decimal): and if so, we just return the argument.
       2. (2) Denormalized number (exponent is 0): return uf left shifted one space since that signifies a multiplication by 2 and add the sign bit with the | operator.­­
    2. Now, for the rest of these values, we can create the bit-level equivalent of 2\*f by adding 1 to exponent since this signifies a multiplication by 2, but we must check if this will cause exponent to become 11111111, which is a special value encoding. If the exponent is too large (equal to 255), return infinity (the hex form is 0x7F800000, then add the sign bit). Finally, if uf passes all these tests, create the fraction bits by using the bitwise & operator on uf and 0x7FFFFF (leftmost 0, rest 23 bits are 1) to focus on the rightmost 23 fraction bits. Adding 23 bits of fraction, 8 bits of exponent, and 1 bit of sign together, we get the floating-point representation of 2\*f.
    3. Issues: I had trouble visualizing what floating point looked like when multiplied by 2, especially in denormalized form vs normalized.
12. float\_i2f
    1. The goal is to find the sign of the integer and its representation in scientific notation with binary. We first determine the sign of x using the bitwise & operator with x and a sign bit mask of (1 << 31). Next, we set exponent to 158 since it is the largest exponent and the bias (31 + 127). Now, we have two special cases that prevent us from effectively running lines 326–334:
       1. If x = 0: we just return 0
       2. If minimum two’s complement number: return floating-point representation, which is 0xCF000000
       3. If negative: turns it positive with two’s complement
    2. Now, we go through a while loop that loops through the bits of x from left to right if the bits are 0, then stops when it finds the first 1. Multiplication by 2 is represented with left shifts, then exponent decreases with each iteration since the number is getting smaller. Next, we remove the first bit of x with (x & ~(1 << 31)) since normalized values drop that 1 when making the fraction. A right shift of 8 bits moves the mantissa field away from the exponent field. Now, we must look at rounding:
       1. If the bit before the last one is 1, and the last bit is 1 or any bits before the second to last are 1: round up by incrementing fraction by 1
       2. If the bit before the last one is 0: no rounding
    3. The expression that checks this is ((x & 128) && ((x & 127) > 0 || fraction & 1)). x & 128 sees if the bit before the last is 1. x & 127 checks if any bit before the second to last is also 1. fraction & 1 checks if the last bit is 1. Afterwards, we add the sign bit, exponent, and fraction together.
    4. Issues: I struggled the most with rounding–I initially could not figure out how to make each case round differently.
13. float\_f2i
    1. We start out by breaking the floating-point representation into:
       1. Exponent: right shift 23 places to get the leftmost 9 bits (8 are exponent, 1 is sign), then the bitwise & with 255 ensures that we focus on the 8 exponent bits
       2. Fraction: use the bitwise & operator on uf and 0x7FFFFF (leftmost 0, rest 23 bits are 1) to focus on the rightmost 23 fraction bits
       3. Sign: conduct a bitwise & operation on uf and 0x80000000
       4. E: exponent – bias = exponent – (2k–1 – 1) = exponent – 127
    2. Now we have a couple of cases to examine:
       1. If exponent is all 1s: special values (infinity, NaN) return 0x80000000u
       2. If e is greater than 30: the max value in an int is 231 – 1 so if we are looking at 231 or larger, then the number will be too large to represent as an integer and is returned as 0x80000000u
       3. If exponent is 0 (denormalized): the number is too small and must be rounded down to 0
       4. If e is less than 0: means that the mantissa is being multiplied by a negative power of 2, which means it is too small a small number, so it is rounded down to 0
    3. To add the hidden 1 in front of the fraction, we append it with the expression fraction | 0x800000. Next, we need to shift based on the value of the actual power of 2 value e:
       1. E is greater than 23: left shift e – 23 to make fraction larger
       2. E is less than 23: right shift 23 – e to make fraction smaller
    4. If the sign bit is 1, we negate the fraction with Two’s Complement with the bitwise NOT operator ~ and adding 1. Then, we return fraction.
    5. Issues: I struggled the most with determining the different cases in (b).