Genchen Li (Questions 1-3, 10, 13)

Nick Benevento (Questions 4-6, 11, 14)

Zach Coveno (Questions 7-9, 12, 15)

Project Answer Explanations

Question 1:

After z >>31, in order to get 1 as the output, z should be like 00000000000000000000000000000001. Therefore, 1st bit of z should be 1 before z right shift 31 times. Therefore, (~(y|x)) = 1, (y|x) = 0 for the first bit. In OR operation, the result will be 0 when both x and y are 0. For two's complement, the first bit will be 1 if x is negative and the the 1st bit of y will be 0.The first bit will be 0 if x is positive and the 1st bit will be 1. When x is 0, which means that the first bit is 0, and then y's 1st bit will also be 0. Therefore, if x is 0, the function output is 1. Else, the function output is 0.

Question 2:

Mask is the bit right shift of 31 on x, which is the 1st bit. Y is XOR mask, which means that it only result in x when x is positive. When x is positive, z is a string of 0. When x is negative, z is a string of 1. And this function returns the absolute value of x.

Question 3:

In order to get the output = 1, z should be 0. In OR operation, only when z and y both be 0, then the result could be 0. Int z = 0 after x >>31. Therefore, the first bit is 0 before x >>31, which means that x is a positive number. Also, y = !x. When y =0, !x = 0. Therefore, x is a positive number. Thus, if x is a positive number , the output is 1. If x is negative, the output is 0.

Question 4:

The function takes in one parameter n. The first line uses a double boolean not operator (!), which flips between false (0) and true (anything besides 0). Since there are 2 nots, this will always result in either 0 or 1. Then, this is bitwise shifted to the left 31 spaces. So, if the n was 0, then x will also be 0. Otherwise, the binary number becomes 1 followed by all 0s, or the number -2,147,483,648. Then, x is shifted back to the right 31 spaces, so it will either remain 0 or become -1, since the digits are padded with the most significant bit. Next, y is initialized as -2,147,483,648 because it is just a 1 followed by all 0s. Y is then shifted by n + ~0, which is the same as saying n-1. Since shifting bits to the right is the same as dividing them by 2, this essentially divides y by 2^{n-1}. The final statement will always return 0 if x is 0, because anything anded with all 0s is 0. Otherwise, it will return whatever y is, since anding with all 1s returns whatever the other binary string is. So, this function will return 0 if 0 is passed in, otherwise it will return -2,147,483,648 divided by 2^{n-1}.

Question 5:

This function starts by taking in a parameter x, and uses a bitwise shift to shift that number 31 bits to the left. In general, since any bits that were overflowed are lost, the only bit

that remains from x is the very first bit all the way to the right. Next, the function shifts the bits back to the right 31 times. In C, the shift is padded with the most significant bit, so if it is a 1, then the entire binary string is filed with ones. Overall, this means that if the original number was even (no 1 in the 1st bit slot), then the function will return 0. If the number was odd (and there was a 1 in the first bit slot), then the entire binary string will be 1's, so the function will return -1.

Question 6:

There are no inputs for this function, so it will always return the same thing. It starts by declaring the hex number 0x55, which is the same as 85 in decimal, or 0101 0101 in binary. Then, this is bitwise or'd with the same string except shifted over 1 byte to the left. So, int word becomes the string 0101 0101 0101 0101. Then, this same idea is repeated for the return statement. Word is shifted over 2 bytes to the left and then or'd with itself, so the string becomes 0101 0101 0101 0101 0101 0101 0101 0101. This is the number 1,453,655,765 in decimal. It is worth noting that the byte 0101 0101 represents the character U, so if each byte is taken to be it's ASCII value, the function would return UUUU.

Question 7:

Ques7 returns the value of the right-most 1 in the binary representation to the parameter value. So if x=6 the binary representation of 6 is 0110 so ques7 return 10 in binary or 2. The function stops after it reads the first 1 bit (reading right to left). Every odd number returns 1 because every odd number in binary has 1 as the first bit.

Ans7 accomplishes this by reading each bit one at a time and stops after reading a 1. It then raises 2 to the location of the bit, with zero being considered as the first location (It translates from binary to decimal, but only the first 1 in the sequence).

Question 8:

Ques8 is a sign check. If the parameter is negative the function returns -1. If the parameter is 0 the function returns 0. If the parameter is positive the function returns 1. It accomplishes this by setting y as x>>31. This replaces all the bits in the sequence with the sign bit (32nd bit). A negative number becomes all 1s or-1 and zero or positive value becomes all 0s or 0. Z = !!x distinguishes the differences between 0 and all other values. By using boolean not all values are reduced to either 1 or 0. Not not returns the boolean value back to its original state (anything but 0 is 1 and 0 is 0). By bitwise or-ing these values the result is 0 | 0, 0 | 1, or -1 | 1 depending on whether the input is zero, positive or negative. The outputs for these are 0, 1, and -1 respectively.

Ans8 simplifys this process by using classic greater than, less than and equal to integer operations.

Question 9:

ques9 takes in 3 integers and returns 1 back. It sets n8 as 8 times the inputted value for n. It accomplishes this with a bitwise shift left 3 operation which results in a multiplication by 2^3 or 8. It then creates an int mask which is 255 (decimal version of hex 0xff) and int cshift as parameter c and bitwise shifts them both left n8 times. For positive values of n greater than 3, the variables retain their initial values (255 and c) because n8 is at least 32 and we are only dealing with 32 bit numbers. Instead of dropping the initial 1's to make room for more shifts, the

function act as if they weren't shifted at all. For -1<n<4 mask and cshift shift accordingly. When -4<n<0 The function

The use of variables in the bitwise shift operations makes ques9 extremely confusing. If these variables were integers, the code would not compile for n>3 because it would be shifting a 32 bit number more than 32 bits. By using variables the code bypasses this error but leads to a total mess. This behavior, when shifting the left number by more bits than it contains, is undefined. Here is a reference link that explains this in more detail, but in general, it says to **not** do this.

Question 10:

When x is positive, !!x is positive and !(x+x) is negative. When x is negative, !!x is negative and !(x+x) is positive. When x is 0, !!x is 0 and !(x+x) is 0. In AND operation, the result will be 1 only when x and y both be 1. Else, it will all be 0. Therefore, whatever x is positive or negative or 0, the output will always be 0.

Question 11:

The function takes in 2 parameters, x and y. The first 2 statements simply shift each of these variables 31 spaces to the right, which pads all the bits with the most significant bit. This means that if the last bit (all the way to the left) is 0, then the entire string becomes 0. If the left most bit was a 1, meaning the number was negative, then the number becomes filled with 1s, which is the number -1. So, a and b will become 0 if they were greater than 0, and -1 otherwise.

The last statement has a lot of bitwise and logical operators. Instead of trying to decode each one, I built a truth table for each of the values, since there were only 4 possibilities.

a	b	f(a, b)
-1	-1	0
-1	0	1
0	-1	0
0	0	0

A much cleaner solution to this truth table is the expression (x && !y).

Question 12:

Ques12 has two 2's complement operations on a and b and then does integer addition before the return statement. The 2's complement sets a as -m and b as -x. The return statement is a bitwise or shifted 31 bits to the right and then boolean not. This a complicated way to check if a or b are negative. When either or both are negative the function returns 0 (because of the not). If both are zero or positive the function returns 1.

Ans7 accomplishes this by checking if m>x or if x>n. If either of these operations are true then that would mean a and b from ques12 were negative and the function returns 0. If both statement are false however the function returns 1.

Question 13:

This function counts the number of 1's in the binary string that is passed in. When x = 1, the output is 1 (0001). When x = 2, the output is 1 (0010). When x = 3, the output is 2 (0011)

Question 14:

Question is similar to question 13 in that it counts the number of 1's in the binary string that is passed in, but the difference is that it returns 1 if there are an odd amount and returns 0 if there are an even amount. It does this in an interesting way, using a for loop to loop through all the bits of x, the number passed in. It creates another variable, and then uses the exclusive or function to compare the right-most bit of x and this temp variable. So, if these are different, then the temp variable gains the value of 1. So, whenever there is a 1 in the binary string of x, the value of temp is flipped between 1 and 0. At the end, this function returns the value of temp, meaning that if there were an odd amount of 1's in x, then the function will return 1. Otherwise, if there were an even amount of 1's in x, then the function returns 0.

Question 15:

Ques15 truncates x (in binary representation) after the nth bit. By setting temp as 1 shifted over n bits, temp is 2 raised to the power n. Subtracting 1's complement 0 is equivalent to subtracting 1. This results in a binary string of all 0s followed by n 1s. This value (z) anded with x truncates all the bits after x's nth bit. Other than that x returns the same.

Ans15 accomplishes this by converting the first n bits of x to decimal. The while loop runs through the first n bit of x. Temp is set as 1 (all 0s ending with one 1) so when it is anded

with x only the first bit of x is read. If this bit is 1 then count is adjusted depending on what bit it is currently on (count = count + 2 raised to the i). This converts the first n bits of x to decimal.