Timothy Lytkine

CSCI 2461 Project 1 Part A

Question 1:

This function has int x as a parameter.
Inside of the function:
An int z is declared.
An int y is initialized to the value of the binary one's complement operator (~)
which flips the bits of x added to the value of 1.
Then z is set equal to the binary ones complement of (y | x)
| is the BINARY OR operator which will check each bit of y and x and copy them to a new output if they exist in either one. Hence if one of the int's has a 1 in it it doesn't matter if the other does or does not have a 1. The 1 will still be copied to the output of OR. It will only copy a 0 if 0 exists at the same bit for both binary numbers. So y and x have the OR operation applied to them and then the one's complement operator (~) will flip the bits of the OR operation. This is what the value of z is set to at that line.
Then in the next line z is set equal to the value of z from the previous line shifted to the right by 31 bits. This means that 31 zeroes will be put in front of the previous binary value of z. Then z has the AND operator with 1 performed on it. This means that at the first bit from the right of z, it will return 1 only if this bit is also 1 in z, otherwise it will just return 0.

Test:

The function returns 0 for all values of x that are not 1, therefore this confirms the explanation provided above to be true.

Question 2:
This function takes x as a parameter.
First, int mask is set equal to the value of x right shifted by 31 bits. This means that the binary value of x will have 31 zeroes placed in front of it.
Second, int y is set equal to the value of x XOR the value of mask from the previous step. After that, an int z is set equal to the sum of the ones complement of the value of mask which means all the bits are flipped and 1.
Finally, the value of y + the value of z is returned by the function.

Test:
The test performed returned the absolute value of the parameter of x each time. While my explanation was correct, I did not realize that the sum of the final values of y and z produces a result equivalent to the value of x.
This makes sense because shifting x to the right by 31 bits makes mask equal to the most significant bit of x or the sign bit.
Then setting y makes it x equal XOR mask depending on whether the number is positive or negative (0 is positive, 1 is negative). If it is 0 it is simply equal to 1 as the least significant bit (when the number is positive). If it is 1 it is equal to the value of x.
Then setting z equal to the complement of mask + 1 makes it equivalent to the most significant bit and 30 zeroes when the number is negative or simply the value of x when the number is positive.
Adding y and z together when the number is negative means adding the value of x to the most significant bit and all zeroes, what this does is gets rid of 1 as the most significant bit due to the carry causes by addition for it to overflow past the 32-bit limit of the number. This makes the negative number positive.
Adding y and z together when the number is positive simply means adding y and z in a way that makes the 1's overflow to form the value of x.
Regardless of whether the input x is negative or positive, it will always return the positive version of that value, hence the absolute value.

Question 3:
This function takes int x and int y as parameters.
The ones complement of x is taken so all of its bits are flipped and the ones complement of y is taken so all of its bits are flipped as well. Then the OR operator is applied to both of these computed values which means that if 1 exists in either binary number it will be copied to a new output, otherwise the value out that output will be 0. Finally, the ones complement operator is applied to the value from the previous step, flipping all the bits. This is the value that the function returns.

Test:
Testing this function allowed me to verify that my explanation above is indeed what the function performs. However, it made me realize that for the two inputs, int x and int y, what the function does is returns the absolute value of x/y if they are equal to each other. Otherwise, the function just returns 0.

Question 4:
This function takes int x and int y as parameters. First, an int sum is set equal to the value of x added to the value of y. Then, an int x_neg is set equal to x right shifted by 31 bits meaning that it will be equal to the binary value of x with 31 zeroes in front of it. Then an int y_neg is set equal to y right shifted by 31 bits putting 31 zeroes once again but this time in front of y. Then, an int s_neg is set equal to the value of sum right shifted by 31 bits so the binary value of x + y has 31 zeroes placed in front of it. The return has many operations being performed:
From the inside out:
1.x_neg ^ y_neg is the value of x_neg XOR the value of y_neg
2.This value has the ~ operator applied to it so the ones complement of the value from the previous step is taken and all the bits are flipped.
3.x_neg ^ s_neg is the value of x_neg XOR the value of s_neg
4. The & operator is applied to the value computed from step 2 and the value computed from step 3. This means that an output is generated where 1 has to exist in both of these values to be included in the output so due to the previous operations being applied that bit will be 0.
5.The NOT (!) operator is finally applied to the value computed from step 4, this means that the the output will be 1.

Test:
The tests confirmed my explanation above to be true. For a variety of different numbers tested, the output was always 1.

Question 5:
The function takes int x as a parameter.
int result is declared an initialized to the value of x left shifted by 31 bits meaning that the binary value of x has 31 zeroes placed to the right of the first bit.
Then result is set equal to the value of result from the previous step right shifted by 31 bits which means that 31 zeroes are placed to the left of the value computed from the previous step.
The function returns -1 or 0 depending on if the shifts cause a 1 to stay at the 32nd bit of the number or not based on the binary value of x. An even value of x will make the output 0 and an odd value of x will make the output -1.

Test:
The output tested at numerous even and odd values as inputs (int x) confirmed the explanation above to be true.

Question 6:
This function takes void as a parameter which just means that it does not take any parameters. This means that int ques6(void) is the same thing as just writing int ques6().
An int byte is set to the hexadecimal value of 0x55. 0x55 in decimal form is 85 so this is what byte is set equal to. Then, an int word is set equal to OR performed on the value of byte which is 85 and the value of byte shifted to the left by 8 bits which is 21760. This OR of 85 and 21760 is equal to 21845. Then the value of word which is 21845 computed from the previous step has the OR operation performed with word shifted to the left by 16 bits. The value of word of 21845 shifted to the left by 16 bits is equal to 141633920. The OR operator performed on 1431633920 with 21845 is equal to 1431655765. Hence, the value returned by the function is 1431655765.

Test:
When tested the function returned a value of 1431655765 which confirmed the above explanation to be true.

Question 7:
This function takes an int x as the parameter.
The AND operator is applied to the value of x with the value of the ones complement of x added to 1. All the bits of x are flipped and then one is added. When this happens, depending on where there are 1's there could be a carry. When the AND operator is applied to the value of one's complement of x +1, if the value of x has just one of the bits as 1 (regardless of the position of the bit) and the rest are 0 then the function returns x. If not, then the function returns 1.

Test:

After testing the function with various different outputs the output was 0 with values such as 1,2,4… since they only use 1 bit to store their value. Other values that use more than one bit return 0. This confirms the explanation above to be true.

Question 8:
This function has an int x as the parameter.
An int xm1 is declared and set equal to the value of x + the complement of 0 which is 1 so it is just x + 1. Then an int possible is set equal to the NOT operation applied to the AND of the values of x and xm1. The AND produces an output with all the 1's at each bit if both x and xm1 contain them at a particular bit and the rest of the bits are 0's. The NOT operator will make the value of possible equal to 0 since either x or xm1 will have the first bit from the right as 0 and AND of 0 and 1 is 0 unless x is -1 which will make
possible equal to 1. Therefore, possible will be 1 or 0.An int result is set equal to the AND of the value of possible (1 or 0) and the values of NOT NOT applied to x which is just x AND the value of not of (X & 1 shifted by 31 bits to the left (equal to 10000000000000000000000000000000 which is 2147482648 in decimal)) However due to the not, the value will be changed to 0. !!x will be equal to 0 unless the value of x is equal to 1. These two values computed from the 2 previous steps (0 and 0 or 1) will have the AND operator performed on them which will be 0 no matter what. Then the AND of possible with 0 regardless of its computed value will be 0.
The value of result is returned which would be 0.

Test:

Testing the function showed my explanation to be false. The function returns either 0 or -1.

Question 9:
This function takes 3 parameters: int x, int n, and int c.
An int n8 is set equal the value of n left shited by 3 bits so the value of n has 3 zeros put to the right of it. An int mask is set equal to 0xff shifted to the left by the value of n8. 0xff is the hexadecimal equivalent of the decimal value of 255. 255 is shifted to the left by the value of n8. Then an int cshift is set equal to the parameter c shifted by the the value of n8 bits to the left so n8 zeroes are put in front of the value of c. After this, an int z is set equal to the & of x and the complement of mask. Then the OR operation is applied on z and c shift and is returned.

Test:
Testing this function produced various values and I am not exactly sure how to simplify it. My explanation above seems to be true however.

Question 10:
his function takes no parameters.
An int byte is set equal to 0xAA in hexadecimal which is 170 in decimal.
Then, an int word is set equal to the OR of byte and byte shifted by 8 bits to the left.
The value of byte is 170 which is equal to 10101010. Performing a left shift of 8 bits on 10101010 makes the value of byte<<8 equal to 1010101000000000 which is equivalent to 43520 in decimal form. Performing OR on 43520 and 170 would equal to 43690 which is the same as just adding the two numbers. Hence. the value of word is 43690 (42350 + 170).
The value of word of 43690 has the OR operator performed on it with word shifted by 16 bits to the left. 43690 shifted by 16 bits to the left is 2863267840. The OR of 43690 and 2863267840 is 2863311530. However, due to the numbers consisting of 32 bits, the MSB is the sign bit which would be 1 so it would mean 2863311530 - 2^32 which would be 1431699456 but since the sign bit is 1 it would be -1431699456 and this is what the function returns.

Test:
The function returns -1431655766 which confirms my explanation above to be true.

Question 11:
This function takes int x as a parameter. An int m8 is set equal to 0xAA which is the hexadecimal form of 170. An int m16 is set equal to the OR of m8 and m8 left shifted by 8 bits. m8 left shifted by 8 bits is equal to 43520. m8 is equal to 170. The OR of 170 and 43520 is equivalent to 43690. This is what the value of m16 is set eqal to. An int m32 is set equal to the OR of m16 and m16 left shifted by 16 bits. m16 is equal to 43690. 43690 (m16) left shifted by 16 bits is 2863267840. 43520 OR 2863267840 is equal to 2863311530. Hence, m32 is equal to 2852170240. (--1431699456 if you consider the sign bit) An int fillx is set equal to the binary OR operation to the value of x OR the value of m32. Then what is returned is the complement of fillx with the NOT operator applied to it. If the MSB (most significant bit) is 1 then the value has 2^32 subtracted from it and is negative.

Test:
Testing this function proved the above explanation to be true.

Question 12:
The function has 3 parameters: int x, int m, and int n.
An int a is set equal to the complement of m (all the bits are flipped) added to 1.
An int b is set equal to the complement of x (all the bits are flipped) added to 1.
Then a is set equal to x added to a, and b is set equal to b added to n. The OR of a and b is taken. then it is right shifted by 31 bits. Then the NOT operator is applied to this which will return 0 since it would not be possible for the operations performed in the parentheses to be equivalent to just one bit of 1. Therefore, the function will always return 0.

Test:
The function always returns 0 confirming my explanation above to be true.

Question 13:

The function takes an int x as a parameter. Then 5 int's: mask1, mask2, mask4, mask8 and mask 16 are declared. First, mask 2 is set equal to the sum of 0x33 which is the decimal equivalent of 51 and 0x33 shifted to the left by 8 bits which is equal to 13056 in decimal form. 51 + 13056 equal to 13107. Hence, this is what the value of mask2 is set to. Then, mask2 is set equal to mask2 added to mask2 shifted to the left by 16 bits. mask2 is equal to 13107 and mask2 shifted to the left by 16 bits is equal to 858980352. Adding these two values produces a sum of 858993459 which is what mask2 is set equal to. On the next line, mask1 is set equal to mask2 XOR mask2 shifted to the left by 1 bit. mask2 is equal to 858993459 and mask2 shifted to the left by 1 bit is equal to 1717986918. 858993459 XOR 1717986918 is an extremely large number which will max out the 32 bits allowed for a number. In this application, we can just assume that it is infinity. Therefore, mask1 is equal to infinity. On the next line mask4 is equal to 0x0F + 0x0F shifted to the left by 8 bits. 0x0F is equivalent to 15 in decimal form. 15 shifted to the left by 8 bits is equal to 3840. 15 + 3840 = 3855. Hence, mask4 is equal to 3855. On the next line, mask4 is equal to the current value of mask4 which is 3855 added to that value shifted to the left by 16 bits. 3855 shifted to the left by 16 bits is equal to 252641280. 252641280 + 3855 is equal to 252645135. Hence, mask4 is set to 252645135. On the next line, mask8 is set equal to 0xFF + OxFF shifted to the left by 16 bits. 0xFF is equal to 255 in decimal form. 255 shifted to the left by 16 bits is equal to 16711680. 16711680 + 255 = 16711935. Hence, mask8 is set to 16711935. On the next line, mask16 is set equal to 0xFF + 0xFF shifted to the left by 8 bits. 0xFF is equal to 255 in decimal form. 255 shifted to the left by 8 bits is equal to 65280. 65280 + 255 = 65535. Hence, mask16 is set to 65535.

The value of x returned is the number of bits set to 1 in x.

All 1's of x

Next

Next

Next

next

Second AND operation:

11111111111111111111111111111111

01111111111111111111111111111111

00011111111111111111111111111111

00000011111111111111111111111111

00000000000000011111111111111111

00000000000000000000000000000001

Adding these 2 AND operations together produces an output of the number of 1's in x.

Test:

Testing this function produced an output of the number 1's in the binary of value of x so this confirms my explanation above to be true.

Question 14:
The function has an int x as a parameter. An int result is declared and initialized to 0. Then an int i is declared. A for loop goes from 0 to 31 incrementing after performing this operation: The value of x is left shifted by the value of i and the AND operator is applied to this with 0x1 which is the hexadecimal equivalent of the decimal value of 1. Then result is set equal to the value of the current value of result (0 at first and then its value from the previous time the operation was performed) to the XOR of the value of the AND operation previously mentioned. After the for loop increments i till 31, the value of result is returned.

Test:

The function returned if the number of 1's in x is even or odd. It returns 1 for an odd number of 1's and 0 for an even number of 1's.

Question 15:
The function takes an int x and an int n as a parameter.
An int temp is set equal to 1 left shifted by n bits. (n can not be a negative number so the shift will always put n zeros to the right of 1). An int z is set equal to the value of temp from the previous line added to the value of the complement of 0 which is just 1. This means that z is just equal to temp + 1. Finally, the value of z has the AND operation performed on it with the value of x and this is what the function returns.

Test:
The function above always returns 1.