Name:\_\_Xiang, Xin\_\_ Date: \_\_\_\_ \_Mar 6, 2020\_ \_ \_

Section: \_ 005\_ \_ \_ Github Username: \_\_FreyaXiang\_ \_

**Lab 1**

Total in points (100 points total):

Professor’s Comments:

Honor Pledge: I have neither given nor received aid on this assignment.

Signature: Xin Xiang

**bitXor:**

x^y = ((~y)&x) | ((~x)&y) = ~((~(y&~x))&(~(x&~y)))

**tmin:**

The minimum two’s complement integer is 10000000 00000000 00000000 00000000. So just left shift one 31 bits, which is 1<<31.

**isTmax:**

The maximum two’s complement integer is 0x7fffffff. If we plus one on this integer, we will get 0x80000000 and we can observe that all the digits are flipped. Because of the flipped digits, !(~(0x7fffffff^0x80000000))=1.

Almost all the numbers do not have this plus-one-digits-flipped property. However, we have an exception. if we plus one on 0xffffffff, we get 0x0 and all the digits are flipped as well. And !(~(0xffffffff^0x0) also equals to 1. To rule this exception out, the integer x+1 cannot be 0. If x+10, !(!(x+1))=1.

Therefore, we should return !(!(x+1)) & !(~(x^(x+1))).

**allOddBits:**

If all odd-numbered digits are set to 1, all the digits of x|0x55555555 should be 1. So !(~(x|0x55555555))=1. However, since we are not allowed to use big integer constant bigger than 0xff, we must create 0x55555555 by a 0x55. First make 0x00005555: lower\_16= 0x55 |(0x55<<8). Then make 0x55555555: complementary=lower\_16 | (lower\_16<<16). Finally we can return !(~(x|complementary)).

**negate:**

Because the bits of 0-1 is all 1, and bits of x+(~x) is also all 1, so 0-1= x+(~x), so x+(~x)+1=0, Therefore, the additive inverse of x is (~x)+1. So the function should return (~x)+1.

**isAsciiDigit:**

The integer x should be in [0x30, 0x39]. So the difference of x and 0x30 should be in [0,9], which means difference=x+(~0x30)+1[0,9]. We divide [0, 9] into two parts: [0, 7] and [8, 9]. If difference [0,7], difference>>3 must be all 0, so !(difference >> 3)=1 If difference [8,9], difference should be 1000 or 1001, so (difference>>1)^0x04 must be 0, so !((difference>>1)^0x04)=1.

Therefore, the function should return (!(difference >> 3)) | (!((difference>>1)^0x04)).

**conditional:**

x is the condition. If x=0, then !!x=0. If x0, then !!x=1.

Then let x=~x+1. Then if x=0, then x should also be 0. If x0, x should be changed to all 1.

If x=0, x&y=0, ~x&z=z. If x is all 1, x&y=y, ~x&z=0. Therefore, the function should return (x&y)|(~x&z).

**isLessOrEqual:**

First, we should judge the sign bit of x and y. Let sign=(!(x>>31)) ^ (!(y>>31)). If the sign bit of x and y are different, sign=1. Otherwise sign=0.

Case 1: If sign bit is different (sign=1) and x is negative, return 1. If x is negative, x>>31 must be all 1, so case 1=sign & (x>>31)=1.

Case 2: If sign bit is same(!sign=1) and y-x0 , return 1. If y-x0, y+(~x)+10. So (y+(~x)+1)>>31 must be all 0, so !((y+(~x)+1)>>31)=1, so (!sign) & !((y+(~x)+1)>>31)=1.

Therefore, the function should return case1 | case2.

**logicalNeg:**

The function will return 1 only if x=0. If we perform x | (~x+1) on 0, we will find that all the digits of 0 | (~0+1) are 0 because ~0+1 is still 0. But if we perform x | (~x+1) on any other number except 0, we will find that the sign bit of x | (~x+1) is always 1. Therefore, (x ^ (~x+1)) >>31 is either all 0 or all 1. All 0 means that x=0, and if we plus 1, the function will return 1. All 1 means that x is not 0, so we can also plus 1, and the function will return 0 (overflow).

**howManyBits:**

First, judge whether the number x is positive or negative. If it is positive, we should find the most significant bit of 1. If it is negative, we should find the most significant bit of 0. After finding the most significant bit of 1/0, we should plus 1 to the final result because we also need a sign bit.

For convenience, we can flip the bits for negative numbers and find the most significant bit of 1 for both positive and negative numbers. So let x=(~sign&x)|(sign&~x), where sign=x>>31.

The final result minus one must be in [0,31], which is at most 5 bits.

First, check if the 5th () bit is one. If the length is more than 16, there must be at least one in bits higher than the 16th bit, which means !!(x>>16) must be one. Left shift 4 to make one in the 5th bit of the final result and right shift x 16 bits. Similarly, check if the 1st, 2nd, 3rd, 4th bit is one. Finally, add the 5 bits together and x, plus one, and return the final answer.

**float\_twice:**

First get sign bit by right shift uf 31 bits. Then get “clean” exp by (uf>>23)&0xff.

Case1: If exp is all 1, then argument is either infinity/NaN. Just return uf.

Case2: If exp is all 0, then argument belongs to denormalized case. Left shift uf one bit and add sign.

Case3:

1. If exp+1 is all 1, we should return the number in the sequence of sign bit (0/1), all-ones-exp, all zeros frac. Therefore, return (sign<<31) | ((exp+1)<<23).
2. If exp+1 is not all 1, just add 1 to exp, so return uf + (1<<23);

**float\_i2f:**

Let initial exp be 158 (the exponent for 0x80000000). Then let sign=x&(1<<31).

Case1: x=0. Just return 0 because the interpretations are the same for integers and floating point numbers.

Case2: x=0x80000000. Return (1<<31) | (158<<23).

Case3: If the argument is negative integer, negate it. Then keep left shift x one bit and subtract exp by one until the first 1 reaches the most significant bit. Then move frac to the least 23 bits (frac=(x&(~m))>>8) and round frac (add one to frac if necessary). Return sign+(exp<<23)+frac.

**float\_f2i:**

Because argument is interpreted as floating point numbers, first get “clean” sign, exp and frac: sign=uf>>31, exp=(uf>>23)&0xff, frac=uf&0x7fffffff, and bias=0x7f.

Case1: NaN and Infinity, which is when exp is all ones. Check by (exp==0xff) and if so, return 0x80000000u.

Case2: In the case of denormalized encoding/normalized exp less than bias, just return 0 because the result will not be an integer but a less-than-one floating point number.

Case3: Normalized cases, let exp=exp-bias.

1. If exp>=31, then this is overflow case. Return 0x80000000u.
2. If the given argument is not exponent overflow:

First get integer result by shifting corresponding bits.

Then add one to the very front (1<<exp).

If the given argument is negative, change the sign of the final result and return it.