2023 Fall CSED 211 Lab Report

Lab number: 2

Student number: 20220302

Name: 김지현

1. Introduction

- Lab 2 aims to learn bit-level representation of float. The first two labs are about two's complement and the last 4 labs are about the floating point operations and conversion between floating numbers and integers.

2. System Design/ Algorithm

1. Lab 2-1 negate(x)

Negating x without using '-' equals finding two's complement (because we assume that your machine uses a two's complement). Two's complement of some binary number can be solved as this equation below.

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

In the code, you can use bitwise NOT to invert all bits and add 1.

2. Lab 2-2 isLess(x,y)

This function returns 1 if y is greater than x and returns 0 if x is greater than y. First, we find the result of x – y using two's complement of y shown above. There are two cases in which y is greater than x. First, if x is less than 0 and y is greater than or equal to y. Secondly, if x and y have the same sign and if 'x-y' is a negative number, then y is greater than x.

So, the integer value check1 confirms the first case where x is a negative number and y is a positive number. Then, the integer value check2 confirms whether x and y have the same sign using the bitwise XOR operator and checks again with the sign of the result x-y using the bitwise AND operator.

Finally, the function checks whether the first case or the second case occurs by checking the MSB of check1 and 2 using the bitwise shift operator. The "& 1" is a helper to confirm that the initial result of bitwise shift yields 1 bit.

3. Lab 2-3 float_abs(uf)

First, I'll explain the integer values written in hexadecimal. An unsigned value mask is 011…1 in binary number Therefore, "uf & mask" drops the sign bit (or sets the sign bit equal to 0). And, minNaN is equal to 01111111100000000000000000000001. When we think of a case where the result is NaN(Not a Number) or infinity, the bits from the exponent part (30^{th} – 23^{rd} bit) all equal to 1. Therefore, if the input uf is greater than $01111111100000000000000000000000_2$, the result will be either NaN or infinity.

With these unsigned values, we first check whether the input argument unsigned uf is less than minNaN. In the instruction, if the argument is NaN, the function returns the argument. So, if the unsigned ab, which equals uf & mask, is greater than or equal to minNaN, the function returns the original argument uf. If not, the function returns ab, which is the absolute value of uf.

4. Lab 2-4 float_twice (uf)

First, I'll explain the unsigned values. An unsigned MSB is equal to (sign bit of uf)0…0. An unsigned exp_mask is a 32-bit representation where the first 8 bits from the MSB are 1 and the rest are 0 (In other words, mask = $01111111100000000000000000000000_2$). Finally, an unsigned exp is bitwise AND of uf and mask, which will give the first 8 bits of uf from the MSB, and the rest are 0.

Now, we check some conditions for exp. First, if the exp = 0 (a denormalized case), uf will be shifted bitwise by 1 so that the exponent is adjusted correctly. Then, we bitwise OR with MSB_shift. to keep the sign bit the same.

If exp is all equal to 1, the argument is either NaN or infinity. Therefore the function returns the argument.

Lastly, if the exp is normalized, by adding $00000001000000000000000000000000_2$, we can do 2 * f.

5. Lab 2-5 float_i2f (uf)

First, I'll explain integer values. A sign value keeps the sign bit of x by bitwise AND with 0x80000000. The frac_mask is $00000001111111111111111111111111_2$ where the fractional parts are 1 and the others are 0. The bit, round, exp, and frac will be used later during the if-else statement.

First, we'll consider a case when x = 0. If x = 0, the function returns 0. Next, we'll consider a case when x = 0x80000000, which is a minimum signed value. In this case, we set the exponent value to be 158. Therefore, the function returns 0xCF000000. Finally, in all other cases (the "else" part), we first check the sign bit of x. If x is negative, we convert x to -x using the two's complement. Then, using a while loop, check the position of MSB and save the value to "bit". Then, calculate exponent = bit (= MSB position) + 127 (which is a bias). Now, do the bitwise shift left by 31-bit so that the MSB is on the left-most side. Then, set the "frac" variable with the first 23 bits of x shifted to the right by 8.

Then, we need to check whether we need rounding or not. If the bit is greater than 23, meaning, there exists at least one bit with 1 in the original x on the first 9-bit places. Therefore, we need to do the rounding. Let the variable "round" be the last 8 bits of x using bitwise AND, and it is the same as the round bit and the sticky bit combined. The first if-statement checks for possible rounding up. For example, round > 128 is the case for round up, and the second part of the if statement is when the value is exactly halfway. After adding 1 to the fraction which represents rounding up, we check the condition for possible overflow of the fraction. If the overflow happens, exp will increase by 1 and the fraction will be set to 0. Finally, the function returns bitwise OR of sign, exp shifted left by 23, and frac.

6. Lab 2-6 float_f2i (x)

First, I'll explain integer values. An unsigned value uf_sign contains the sign bit of the argument uf, using the right shift. uf_exp contains the exponent part of the argument and stores the bits by shifting to the right using the right shift. uf_frac contains the fractional part of the uf. bias contains the number 127 in hexadecimal. res contains uf_frac, and will be manipulated in certain cases.

Now, we will check the conditions for the exponent of the uf. If exponent bits are equal to 1, the argument is either NaN or infinity. Therefore, the function returns the minimum signed value (0x80000000u). Next, if the exponent is smaller than the bias, it means that the argument is denormalized and very close to 0. Therefore, the function returns 0x0 (= 0 in decimal). Lastly, in all other normalized cases, the function subtracts bias from the exponent of the argument uf.

With the new exponent value, the function checks several conditions for the exponent. First, if the exponent is greater than 31, it means that the integer representation requires more than 32 bits. In other words, the overflow happens. Therefore, the function returns the minimum signed value (0x80000000u). Next, bitwise shift is needed to align the fractional part with the integer portion. If the exponent is greater than 22, the fraction part should shift to the left by uf_exp - 23. On the other hand, if the exponent is less than or equal to 22, the fraction part shifts to the right by uf_exp - 23.

The function covered the exponent and fractional part. Now, it must consider 1 from 1. xx…x. Therefore, we add 1 << uf_exp to the result. Then, finally, after checking the sign of the argument, if the argument is negative, the function negates the result using two's complement method.