Calvin Liu

Homework

804182525

2.83, 2.88, 2.89, 3.56

Fill in the return value for the following procedure, which tests whether its first argument is less than or equal to its second. Assume the function f2u returns an unsigned 32-bit number having the same bit representation as its floating-point argument. You can assume that neither argument is *NaN*. The two flavors of zero, +0 and −0, are considered equal.

int float\_le(float x, float y) {

unsigned ux = f2u(x);

unsigned uy = f2u(y);

/\* Get the sign bits \*/

unsigned sx = ux >> 31;

unsigned sy = uy >> 31;

/\* Give an expression using only ux, uy, sx, and sy \*/

return ((!sx & !sy) & (ux <=uy)) | (sx&!sy) | (sx&sy) & (uy <= ux));

}

We are running programs on a machine where values of type int have a 32- bit two’s-complement representation. Values of type float use the 32-bit IEEE format, and values of type double use the 64-bit IEEE format. We generate arbitrary integer values x, y, and z, and convert them to values of type double as follows:

/\* Create some arbitrary values \*/

int x = random();

int y = random();

int z = random();

/\* Convert to double \*/

double dx = (double) x;

double dy = (double) y;

double dz = (double) z;

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0. Note that you cannot use an IA32 machine running gcc to test your answers, since it would use the 80-bit extended-precision representation for both float and double.

1. (float) x == (float) dx – it will not always yield 1 because from double to float, the value can overflow to +∞ or −∞, since the range is smaller. Otherwise, it may be rounded, because the precision is smaller.
2. dx - dy == (double) (x-y) – Always yield 1 because ints to double does not lose precision and int – int will always yield an int which will then convert to a double. The left part is like saying int – int then converting to a double as well.
3. (dx+dy)+dz==dx+(dy+dz) – Does not always yield 1 because floating point addition is not associative. Pg 113 at the bottom.
4. (dx\*dy)\*dz==dx\*(dy\*dz) – Floating point multiplication is not associative as well. Pg 114 at the top.
5. dx/dx==dz/dz = Will always yield 1 because an int converted to a double doe not lose anything. Then dividing the double by a double is equivalent to multiplying by 1/ the double and since this is not associative, but commutative, it holds true.

You have been assigned the task of writing a C function to compute a floating- point representation of 2x. You decide that the best way to do this is to directly construct the IEEE single-precision representation of the result. When x is too small, your routine will return 0.0. When x is too large, it will return +∞. Fill in the blank portions of the code that follows to compute the correct result. Assume the function u2f returns a floating-point value having an identical bit representation as its unsigned argument.

float fpwr2(int x)

{

/\* Result exponent and fraction \*/

unsigned exp, frac;

unsigned u;

if (x < -126) {

/\* Too small. Return 0.0 \*/

exp = 0;

 frac = 0;

} else if (x < 0) {

/\* Denormalized result \*/

exp = -126;

frac = (2 << x);

} else if (x < 128) {

/\* Normalized result. \*/

exp = 127 + x;

frac = (2 << x);

} else {

/\* Too big. Return +oo \*/

exp = 0xFF;

frac = 0;

}

/\* Pack exp and frac into 32 bits \*/

u = exp << 23 | frac;

/\* Return as float \*/

return u2f(u);

}

int loop(int x, int n)

{  
 int result = -1;

int mask;

for(mask = 1; mask != 0; mask = mask << n) {

result ^= mask & x ;

}

return result;

}

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register %eax. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables.

1. Which registers hold program values x, n, result, and mask? – esi holds x, ebx holds n, edi holds result and mask is held by edx
2. What are the initial values of result and mask? – The initial value of result is -1 and mask is 1
3. What is the test condition for mask? – mask != 0
4. How does mask get updated? It gets shifted to the left n times
5. How does result get updated? – Result goes through ^= to get updated
6. Fill in all the missing parts of the C code.