**Lab 1: Floating Point Arithmetic in C**

for

David Retz

Computer Architecture

Course: CPE 315 Section: 01

California Polytechnic State University

San Luis Obispo, California

by

Patrick Cook

Brett Nelson

October 4, 2016

**Introduction:**

For lab 1 we were required to create a program in C that simulates how floating point numbers are stored and used in simple arithmetic.

**Purpose:**

The purpose of this lab was to learn how floating point numbers are implemented in all modern computers, to learn about the IEEE 754 standard, implement floating point operations in a C program in order to understand how the hardware normally handles floating point numbers, and to create an emulation of basic floating point arithmetic.

**Functional Requirements:**

Our C program must be able to:

* Convert a floating point number into an unsigned int by extracting the sign, exponent, and fraction bits into an INTFLOAT\_PTR struct
* Have the ability to normalize an extracted float
* Perform basic addition and subtraction of floating point numbers using the extracted bit fields stored in an INTFLOAT\_PTR struct
* Pack the struct fields (sign, exponent, fraction) into a float
* Return a float that represents the results of an addition or subtraction

**Approach:**

In order to accomplish the goals of this lab we wrote 5 main functions:

extract\_float(): takes in a float variable, extracts the sign, exponent, and fraction bit fields using bit manipulation and bit masks. These fields were then stored in a struct INTFLOAT\_PTR which allowed other functions to access these specific bit fields easily

normalize(): We used a while loop that checks for the sign bit of the fraction. It continually left shifts the fraction field while decrementing the exponent until the sign bit and the bit to the right of it differ.

pack(): this function was used to pack all the bit fields stored in the INTFLOAT\_PTR struct into a float so that the result of an addition or subtraction can be represented as an IEEE 754 floating point number.

single\_float\_add: This function took in two floating point numbers which were then extracted into two INTFLOAT\_PTR structs using our extract\_float() function. The fraction fields of each struct were then placed on the same scale by shifting the smaller number to the right using and incrementing the exponent by the shift size. The fractions were then added together and placed into an INTFLOAT\_PTR struct which was use to store our result. The other fields (sign, exponent) were then populated. The result is then normalized, packed, cast to a float and then returned.

single\_float\_subtract: this function works off the same logic as single\_float\_add but instead of adding the fractions it subtracts them.

**Difficulties during implementation:**

Our biggest difficulty was understanding how to accurately use bit manipulation to alter many of the floating point number fields and struct fields that we were working with.

**Information learned from this lab:**

After completing this lab we have a stronger understanding of the IEEE 754 floating point number standard, bit manipulation, and basic floating point arithmetic and how those operations are completing using hardware instead of our C simulator.

**Source Code:**

**lab1.c**

#include "stdio.h"

#include "stdlib.h"

#include "lab1.h"

#include "string.h"

/\*

\* Author: Patrick Cook

\* Date: OCT 1, 2016

\*/

int main(int argc, char const \*argv[]) {

//\*\*\*\*CHANGE THESE NUMBERS TO TEST PROGRAM\*\*\*\*//

float a = 11.125, b = 8, res;

//Test addition

res = single\_float\_add(a, b);

printf("Add %f + %f = %f\n", a, b, res);

//Test substraction

res = single\_float\_subtract(a, b);

printf("Sub %f - %f = %f\n", a, b,res);

return 0;

}

/\*

\* continually left shifts the fraction field

\* while decrementing the exponent until the sign

\* bit and the bit to the right of it differ.

\*

\*/

void normalize(INTFLOAT\_PTR x){

while (x->fraction != 0) {

if (x->fraction > 0) {

if (x->fraction & 0x40000000) {

return;

}

} else {

if ((x->fraction & 0x40000000) == 0) {

return;

}

}

x->fraction <<= 1;

x->exponent--;

}

}

/\*

\* Pulls the bit fields from a floating point number

\* and populates the INTFLOAT\_PTR struct fields

\* Returns: void. struct passed in now contains the

\* bit fields held by the float passed in.

\*/

void extract\_float(INTFLOAT\_PTR x, float f) {

unsigned extract = \* (unsigned int \*) &f;

//extract sign bit

x->sign = (extract & 0x80000000);

//Extract exponent into struct

x->exponent = (extract >> 23) & 0x000000FF;

x->exponent -= 127;

//everything except the hidden one being anded into fraction

x->fraction = (extract << 7) & 0x3FFFFF80;

//Hidden 1

x->fraction |= 0x40000000;

if (x->sign)

x->fraction = -x->fraction;

}

/\*

\* Takes in two floating point numbers, extracts the bit fields,

\* performs bit shifts in order to keep both floats on the same

\* scale, and then proceeds to add the numbers.

\* Returns a normalized and packed float

\*/

float single\_float\_add(float a, float b){

int expDiff = 0;

INTFLOAT\_PTR ptrA = malloc(sizeof(INTFLOAT));

INTFLOAT\_PTR ptrB = malloc(sizeof(INTFLOAT));

INTFLOAT\_PTR result = malloc(sizeof(INTFLOAT));

//populate struct with values of IEEE 754 floating point

extract\_float(ptrA, a);

extract\_float(ptrB, b);

//shift the lower scaled number to match scales

expDiff = ptrA->exponent - ptrB->exponent;

if (expDiff > 0) { //num A is higher scale than B

ptrB->fraction >>= expDiff;

ptrB->exponent += expDiff;

} else if (expDiff < 0) { //num B is higher scale than A

ptrA->fraction >>= -expDiff;

ptrA->exponent += expDiff;

}

//Add both fractions after shifting them. (shift retains sign bit)

result->fraction = (ptrA->fraction >> 1) + (ptrB->fraction >> 1);

result->exponent = ptrA->exponent + 1;

normalize(result);

return pack(result);

}

/\*

\* Packs bit fields from floating point arithmetic and then

\* places them in an unsigned int which is then cast to a float

\* Input: INTFLOAT\_PTR struct containing filled bit fields

\* Returns: a float that contains all bit fields held in pointer

\*/

float pack(INTFLOAT\_PTR x) {

unsigned int result = 0;

float ret = 0;

if (x->sign) {

result |= 0x80000000;

}

result |= ((x->exponent + 127) << 23);

result |= (x->fraction >> 7) & 0x003FFFFF;

ret = \*((float \*)&result);

return ret;

}

/\*

\* Takes in two floating point numbers, extracts the bit fields,

\* performs bit shifts in order to keep both floats on the same

\* scale, and then proceeds to substract the numbers.

\* Returns a normalized and packed float

\*/

float single\_float\_subtract(float a, float b){

int expDiff = 0;

INTFLOAT\_PTR ptrA = malloc(sizeof(INTFLOAT));

INTFLOAT\_PTR ptrB = malloc(sizeof(INTFLOAT));

INTFLOAT\_PTR result = malloc(sizeof(INTFLOAT));

//populate struct with values of IEEE 754 floating point

extract\_float(ptrA, a);

extract\_float(ptrB, b);

//shift the lower scaled number to match scales

expDiff = ptrA->exponent - ptrB->exponent;

if (expDiff > 0) { //num A is higher scale than B

ptrB->fraction >>= expDiff;

ptrB->exponent += expDiff;

} else if (expDiff < 0) { //num B is higher scale than A

ptrA->fraction >>= -expDiff;

ptrA->exponent += expDiff;

}

//Substract both fractions after shifting them. (shift retains sign bit)

result->fraction = (ptrA->fraction >> 1) - (ptrB->fraction >> 1);

result->exponent = ptrA->exponent + 1;

normalize(result);

return pack(result);

}

**lab1.h**

#ifndef LAB1\_H

#define LAB1\_H

typedef struct \_intfloat {

int sign;

int fraction;

int exponent;

} INTFLOAT, \*INTFLOAT\_PTR;

void extract\_float(INTFLOAT\_PTR x, float f);

float single\_float\_add(float a, float b);

float single\_float\_subtract(float a, float b);

void normalize(INTFLOAT\_PTR x);

float pack(INTFLOAT\_PTR x);

#endif