CISP 440

Austin Smothers

Assignment 2

/\*

This program builds on the float.cpp program

provided by Professor Ross

\*/

/\*

Floating point arithmetic the HARD way.

Dan Ross Jan 2013

Converts a floating point number from a string representation to

an 8-bit representation using only INTEGER operations. And vice-versa.

This is usually done for us by the preprocessor/compiler and library

functions like atof().

This program illustrates that float operations are computationally expensive.

Thus, they are either done in software to save chip space, or in specialized

hardware to save execution time - this is the classic tradeoff in computer science.

Here, we are pretending that neither our hardware or our compiler support floating

point operations. So we have to make our own!

BIT FORMAT:

1 sign bit

4 mantissa bits

3 exponent bits in excess 4 format (range is -4 to 3)

EXAMPLE: 3.5 = 11.1 = 1.1100 x 2^1 = 0 1100 101

RANGE:

smallest magnitude: binary 1.0000 x 2 ^ -4 = 00.06250000

largest magnitude: binary 1.1111 x 2 ^ 3 = 15.50000000 (2 leading digits req'd)

highest precision: binary 1.0001 x 2 ^ -4 = 00.06640625 (8 trailing digits req'd)

INPUT STRINGS:

Input strings must be of the form: -X.X

with 1 to 2 leading zeros and 1 to 8 trailing zeros

and an optional minus sign

Examples:

0.0

00.0000

-1.2

42.7777

00.06640625

Note: A Finite Acceptor in the preprocessor would check the string format

before converting it to binary

\*/

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#pragma warning( disable : 4996)

#pragma warning( disable : 4244)

/\*

Calculates base raised to the power exp

\*/

int my\_pow(int base, int exp)

{

int x = 1;

for(int i = 0; i < exp; i++)

x \*= base;

return x;

}

/\*

Prints each bit of a byte

\*/

void print8bits(unsigned char x)

{

for(unsigned char mask = 0x80; mask; mask >>= 1) {

if(mask & x)

printf("1");

else

printf("0");

}

}

/\*

Prints each bit of a 16 bit int

\*/

void print16bits(unsigned short x)

{

for(unsigned short mask = 0x8000; mask; mask >>= 1) {

if(mask & x)

printf("1");

else

printf("0");

}

}

/\*

Prints each bit of a 32 bit int

\*/

void print32bits(unsigned long x)

{

for(unsigned long mask = 0x80000000; mask; mask >>= 1) {

if(mask & x)

printf("1");

else

printf("0");

}

}

/\*

Converts a floating point number from a string representation to

an 8-bit representation using only INTEGER operations.

0) copy whole and fractional parts of string to lil buffers

1) convert string whole part to binary

2) convert string fractional part to binary x 100,000,000 decimal

3) convert integer fractional to float mantissa by subtracting

neg powers of 2 times 100,000,000 dec (integer subtraction)

to figure out the bits 2^1/2, 2^1/4 ... 2^1/256

4) normalize and calculate exponent

5) final packing

\*/

unsigned char my\_atof(char \* str)

{

char strIn[20];

// check if negative and remove sign for convenience

int negative = false;

if(str[0] == '-') {

negative = true;

strcpy(strIn, str + 1);

} else

strcpy(strIn, str);

int len = strlen(strIn);

// buffers for the whole and the fractional parts

#define S\_WHOLE\_SIZE 3

char s\_whole[S\_WHOLE\_SIZE] = "00";

#define S\_FRACT\_SIZE 9

char s\_fract[S\_FRACT\_SIZE] = "00000000";

// find the decimal point

int x = 0;

while(strIn[x] != '.')x++;

// copy wholepart to a string buffer in reverse order

for(int i = x - 1, j = 0; i >= 0; i--, j++)

s\_whole[j] = strIn[i];

// copy fraction to a string buffer in reverse order

for(int i = x + 1, j = S\_FRACT\_SIZE - 2; i < len; i++, j--)

s\_fract[j] = strIn[i];

// convert whole part from a string to binary

// these are all integer operations internally

unsigned char i\_whole = 0;

for(int i = 0; i < S\_WHOLE\_SIZE - 1; i ++)

i\_whole += (s\_whole[i] - '0') \* my\_pow(10, i);

// check for overflow

// 16 in binary is 10000 = 1.0000 x 2 ^ 4, exponent will not fit

// note that there are float values > 15.0 that will fit with ROUND error

if(i\_whole > 15){ // (2 ^ n) - 1 in general

printf("atof conversion Overflow %s", str);

exit(0);

}

// convert fractional part from a string to binary (x 100,000,000 decimal)

// these are all integer operations internally.

unsigned long i\_fract = 0;

for(int i = 0; i < S\_FRACT\_SIZE - 1; i ++)

i\_fract += (s\_fract[i] - '0') \* my\_pow(10, i);

// convert the fraction from integer to our floating binary format

// subtract negative powers of 2 (x100,000,000 decimal)

// Example: 2^-1 = 1/2 = 0.5, 2^-2 = 1/4 = 0.25, etc.

// Scaled: 50000000, 25000000, ... 00390625

// Could also use the multiplication method but that is more

// computationally expensive than subtraction:

// http://sandbox.mc.edu/~bennet/cs110/flt/dtof.html

unsigned long powof2 = 50000000;

unsigned char b\_fract = 0; // bits to the right of the decimal point

unsigned char mask = 0x80;

for(int i = 0; i < S\_FRACT\_SIZE - 1; i ++) {

if(i\_fract >= powof2) { // check if bit needed

b\_fract |= mask; // insert this bit

i\_fract -= powof2; // subtract value

}

mask >>= 1;

powof2 >>= 1;

}

//print8bits(b\_fract);

// now pack the unnormalized bits to a 'bit field'

// so we can normalize it

unsigned short buffer = 0;

buffer = i\_whole; // put the whole part in the high byte

buffer <<= 8;

buffer |= b\_fract; // put the fraction part in the low byte

// example: 14.00390625 will be 0000 1110.0000 0001

//print16bits(buffer);

// check for underflow - if everything is zero and there is still a remainder

if(!buffer && i\_fract){

printf("atof conversion Underflow %s", str);

exit(0);

}

// normalize - find the first 1 from left to right

int exponent = 7;

unsigned short mask2 = 0x8000;

while(!(buffer & mask2)) {

exponent--;

mask2 >>= 1;

}

//printf("exp: %d\n", exponent);

// another overflow check (redundant)

if(exponent > 3){

printf("atof conversion Overflow %s", str);

exit(0);

}

// another underflow check (for tiny powers of 2)

if(exponent < -4){

printf("atof conversion Underflow %s", str);

exit(0);

}

// final packing

unsigned char theFloat = 0;

buffer <<= (7 - exponent); // align mantissa

buffer >>= 8; // scoot into low byte

theFloat = buffer; // pack the mantissa

theFloat &= 0x78; // mask off the other stuff

exponent += 4; // the excess 4 thing

theFloat |= exponent; // insert the exponent

if(negative) theFloat |= 0x80; // insert sign bit

return theFloat;

}

/\*

Converts a floating point number from an 8-bit representation

to a string representation to using only INTEGER operations.

0) unpack whole and fractional parts

1) multiply fractional part by 100,000,000 decimal by

adding up the bit values

1) convert whole part to base ten string

2) convert fractional part to base ten string

\*/

void my\_ftoa(unsigned char f, char \* strOut)

{

int ch\_p = 0; // pointer to string chars

if(f & 0x80) strOut[ch\_p++] = '-'; // is it negative

int exponent;

exponent = (f & 0x07) - 4; // get the exponent

f &= 0x78; // mask off everything except mantissa

f |= 0x80; // put on the leading 1

//print8bits(f);

// now pack the normalized bits to a 'bit field' so

// so we can de-normalize it

unsigned short buffer = 0;

buffer = f;

buffer <<= 8; // scoot into high byte

buffer >>= (7 - exponent); // de-normalize

//print16bits(buffer);

// get the whole part

unsigned char i\_whole; // bits to left of decimal

i\_whole = (buffer & 0xFF00) >> 8;

// get the fractional part

unsigned char b\_fract; // bits to right of decimal

b\_fract = (buffer & 0x00FF);

// add up the bit values in the mantissa using INTEGERS only

// we are adding up negative powers of 2 scaled by 100,000,000 decimal

// NOTE: Could easily loopify this...

unsigned long i\_fract = 0;

if(b\_fract & 0x80) i\_fract += 50000000;

if(b\_fract & 0x40) i\_fract += 25000000;

if(b\_fract & 0x20) i\_fract += 12500000;

if(b\_fract & 0x10) i\_fract += 6250000;

if(b\_fract & 0x08) i\_fract += 3125000;

if(b\_fract & 0x04) i\_fract += 1562500;

if(b\_fract & 0x02) i\_fract += 781250;

if(b\_fract & 0x01) i\_fract += 390625;

// convert to decimal string format 00.00000000 with optional leading '-'

// Note: Could loopify this but need to calculate the subtractor

// values. Could do that using integer division (expensive), or

// integer multiplication (also expensive).

// BUT, could use a (fast) lookup table for the subtractor values

// to avoid division.

// first do the integer part

// do the tens

strOut[ch\_p] = '0';

while(i\_whole >= 10){ // tens

strOut[ch\_p]++; // count by characters

i\_whole -= 10;

}

ch\_p++; // next write spot

// do the ones

strOut[ch\_p] = '0';

while(i\_whole >= 1){

strOut[ch\_p]++;

i\_whole -= 1;

}

ch\_p++;

strOut[ch\_p] = '.'; // decimal point

ch\_p++;

// now do the fractional part

// do the '10,000,000'

strOut[ch\_p] = '0';

while(i\_fract >= 10000000){

strOut[ch\_p]++;

i\_fract -= 10000000;

}

ch\_p++;

// do the '1,000,000'

strOut[ch\_p] = '0';

while(i\_fract >= 1000000){

strOut[ch\_p]++;

i\_fract -= 1000000;

}

ch\_p++;

// do the '100,000'

strOut[ch\_p] = '0';

while(i\_fract >= 100000){

strOut[ch\_p]++;

i\_fract -= 100000;

}

ch\_p++;

// do the '10,000'

strOut[ch\_p] = '0';

while(i\_fract >= 10000){

strOut[ch\_p]++;

i\_fract -= 10000;

}

ch\_p++;

// do the 'thousands'

strOut[ch\_p] = '0';

while(i\_fract >= 1000){

strOut[ch\_p]++;

i\_fract -= 1000;

}

ch\_p++;

// do the 'hundreds'

strOut[ch\_p] = '0';

while(i\_fract >= 100){

strOut[ch\_p]++;

i\_fract -= 100;

}

ch\_p++;

// do the 'tens'

strOut[ch\_p] = '0';

while(i\_fract >= 10){

strOut[ch\_p]++;

i\_fract -= 10;

}

ch\_p++;

// do the 'ones'

strOut[ch\_p] = '0';

while(i\_fract >= 1){

strOut[ch\_p]++;

i\_fract -= 1;

}

ch\_p++;

strOut[ch\_p] = 0; // null terminator

}

/\*

Adds two 8-bit floating point numbers.

Puts them into 16 bit buffers, denormalizes them

Adds them as integers, normalizes the result.

Only works for positive numbers.

Based more or less kinda on:

http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/arith.flpt.html

Example: 2.125 + 0.125 = 2.25

Binary: 1.0001 x 2^1 + 1.0000 x 2^-3 = 1.0010 x 2^1

Packed: 0 0001 101 + 0 0000 001 = 0 0010 101

Unpacked: 0000 0010.0010 0000

+ 0000 0000.0010 0000

= 0000 0010.0100 0000

\*/

unsigned char addFloats(unsigned char f1, unsigned char f2)

{

unsigned char theFloat = 0; // the answer to return

char storage1[40] = "0"; // placeholder for strOut when unpacking f1

char storage2[40] = "0"; // placeholder for strOut when unpacking f2

int ch\_p = 0; // pointer to string chars

if (f1 & 0x80) storage1[ch\_p++] = '-'; // is it negative

int exponent;

exponent = (f1 & 0x07) - 4; // get the exponent

f1 &= 0x78; // mask off everything except mantissa

f1 |= 0x80; // put on the leading 1

//print8bits(f);

// now pack the normalized bits to a 'bit field' so

// so we can de-normalize it

unsigned long buffer1 = 0;

buffer1 = f1;

buffer1 <<= 8; // scoot into high byte

buffer1 >>= (7 - exponent); // de-normalize// get the exponent

// get the whole part

unsigned char i\_whole; // bits to left of decimal

i\_whole = (buffer1 & 0xFF00) >> 8;

// get the fractional part

unsigned char b\_fract; // bits to right of decimal

b\_fract = (buffer1 & 0x00FF);

/\*

DO AGAIN FOR SECOND NUMBER

\*/

ch\_p = 0; // reset the pointer to string chars

if (f2 & 0x80) storage2[ch\_p++] = '-'; // is it negative

exponent = (f2 & 0x07) - 4; // get the exponent

f2 &= 0x78; // mask off everything except mantissa

f2 |= 0x80; // put on the leading 1

//print8bits(f);

// now pack the normalized bits to a 'bit field' so

// so we can de-normalize it

unsigned long buffer2 = 0;

buffer2 = f2;

buffer2 <<= 8; // scoot into high byte

buffer2 >>= (7 - exponent); // de-normalize// get the exponent

// get the whole part

i\_whole = (buffer2 & 0xFF00) >> 8;

// get the fractional part

b\_fract = (buffer2 & 0x00FF);

// add them with ordinary integer addition!

unsigned long buffer3 = buffer1 + buffer2;

// normalize - find the first 1 from left to right

exponent = 7;

unsigned short mask2 = 0x8000;

while (!(buffer3 & mask2)) {

exponent--;

mask2 >>= 1;

}

//printf("exp: %d\n", exponent);

// another overflow check (redundant)

if (exponent > 3){

printf("atof conversion Overflow %s", buffer3);

exit(0);

}

// another underflow check (for tiny powers of 2)

if (exponent < -4){

printf("atof conversion Underflow %s", buffer3);

exit(0);

}

// final packing

buffer3 <<= (7 - exponent); // align mantissa

buffer3 >>= 8; // scoot into low byte

theFloat = buffer3; // pack the mantissa

theFloat &= 0x78; // mask off the other stuff

exponent += 4; // the excess 4 thing

theFloat |= exponent; // insert the exponent

return theFloat;

}

/\*

Multiplies two 8-bit floating point numbers.

Puts them into 16 bit buffers, NORMALIZED

Multiplies them as integers

Normalizes the result

Adds the exponents.

Based more or less kinda on:

http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/arith.flpt.html

Example: 1.5 \* 1.5 = 2.25

Binary: 1.1000 x 2^0 \* 1.1000 x 2^0 = 1.0010 x 2^1

Packed: 0 1000 100 \* 0 1000 100 = 0 0010 101

Unpacked NORMALIZED: 0000 0001.1000 0000 e0

\* 0000 0001.1000 0000 e0

= 0010.0100 0000 0000 0000 e1 = e0 + e0 + e1rollover

normalize = 0000 0001 0010 0000 e1

\*/

unsigned char multiplyFloats(unsigned char f1, unsigned char f2)

{

unsigned char theFloat = 0; // the answer to return

// repack the bits to a 'bit field' keep it normalized

// get the exponent

// mask off everything except mantissa

// put on the leading 1

// scoot into normal position

// REPEAT FOR SECOND NUMBER

// multiply them with ordinary integer multiplication!

//buffer3 = buffer1 \* buffer2;

// normalize the result

// check if answer is >= 2

// normalize and set rollover value

// calculate exponent for answer

//int exponent3 = exponent1 + exponent2 + rollover;

//printf("exp: %d\n", exponent3);

// overflow check - if exponent is greater than 3

// underflow check - if exponent is less than -4

// final packing

// scoot into normal position

// pack the mantissa

// mask off the other stuff

// add the excess 4 thing

// insert the exponent

// insert sign bit

printf("Packed Prod: "); print8bits(theFloat); printf("\n");

return theFloat;

}

void main()

{

char strIn1[40] = "0.125";

char strIn2[40] = "4.5";

char strOut[40];

char strOut2[40]; //will store decimal f1+f2

unsigned char f1;

unsigned char f2;

unsigned char f3;

unsigned char f4; //will store f1 + f2

// multiply and add some stuff

f1 = my\_atof(strIn1);

printf("#1 Expanded: ");

print16bits(f1);

printf("\n");

f2 = my\_atof(strIn2);

printf("#2 Expanded: ");

print16bits(f2);

printf("\n");

f3 = multiplyFloats(f1, f2);

f4 = addFloats(f1, f2);

my\_ftoa(f3, strOut);

my\_ftoa(f4, strOut2);

printf("%s \* %s = %s\n", strIn1, strIn2, strOut);

printf("%s + %s = %s\n", strIn1, strIn2, strOut2);

}

