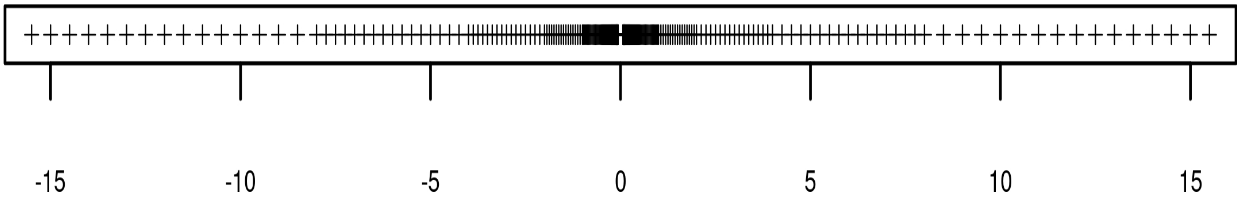
CISP440

Joseph Morgan

Homework 2

Number Line of All Possible Values



Floating Point Addition Output

This output was piped directly from my running program into a text file.
This was done via:

```
$: cat ./input.txt | ./test_software_float.out > output.txt
```

I've removed superfluous output lines.

```
**** Joseph Morgan's test main for software implementation of floats ****

* * * * * * * * * * * * * * * * *

First Number:   3.14159
Second Number:  2.75
#1 Expanded:    0000001100100000
#2 Expanded:    0000001011000000
Sum:            0000010111100000
Sum Normalized: 0000000010111100
Sum Packed:     00111110

* * * * * * * * * * * * * * * * *

Sum: 05.75000000

* * * * * * * * * * * * * * * * *


* * * * * * * * * * * * * * * * *

First Number:   -4.6
Second Number:  1.92
#1 Expanded:    0000010010000000
#2 Expanded:    0000000111100000
```

```
Sum:             0000001010100000
Sum Normalized: 0000000010101000
Sum Packed:     10101101

* * * * * * * * * * * * * * * * *

Sum: -02.62500000

* * * * * * * * * * * * * * * * *


* * * * * * * * * * * * * * * * *

First Number:   2.125
Second Number:  -7.79
#1 Expanded:    0000001000100000
#2 Expanded:    0000011111000000
Sum:            0000010110100000
Sum Normalized: 0000000010110100
Sum Packed:     10110110

* * * * * * * * * * * * * * * * *

Sum: -05.50000000

* * * * * * * * * * * * * * * * *



* * * * * * * * * * * * * * * * *

First Number:   -0.4887
Second Number:  -13.8
#1 Expanded:    0000000001111100
#2 Expanded:    0000110110000000
Sum:            0000110111111100
Sum Normalized: 0000000011011111
Sum Packed:     11011111

* * * * * * * * * * * * * * * * *

Sum: -13.50000000

* * * * * * * * * * * * * * * * *
```

Floating Point Multiplication Output

```
**** Joseph Morgan's test main for software implementation of floats ****

Please enter a number to use in the test:
Menu options:

     1 - test atof
     2 - test ftoa
     3 - test add
     4 - test multiply
     5 - input new number
     0 - exit

Selection:
What would you like to multiply with your number?
Please enter a number to use in the test:

* * * * * * * * * * * * * * * * *

First Number:   2.7
Second Number:  1.5
#1 Expanded:      0000 0001 0101 0000
#1 Exponent:      1
#2 Expanded:      0000 0001 1000 0000
#2 Exponent:      0
Product:          0000 0001 1111 1000
Product Exponent: 1
Pruduct Packed:   0111 1101

* * * * * * * * * * * * * * * * *

Sum: 03.87500000

* * * * * * * * * * * * * * * * *



* * * * * * * * * * * * * * * * *

First Number:   -1.0
Second Number:  2.5
#1 Expanded:      0000 0001 0000 0000
#1 Exponent:      0
#2 Expanded:      0000 0001 0100 0000
#2 Exponent:      1
Product:          0000 0001 0100 0000
Product Exponent: 1
Pruduct Packed:   1010 0101

* * * * * * * * * * * * * * * * *
```

```
Sum: -02.50000000

* * * * * * * * * * * * * * * * *



* * * * * * * * * * * * * * * * *

First Number:   3.75
Second Number:  -.225
#1 Expanded:      0000 0001 1110 0000
#1 Exponent:      1
#2 Expanded:      0000 0001 1100 0000
#2 Exponent:      -3
Product:          0000 0011 0100 1000
Product Exponent: -1
Pruduct Packed:   1101 0011

* * * * * * * * * * * * * * * * *

Sum: -00.81250000

* * * * * * * * * * * * * * * * *



* * * * * * * * * * * * * * * * *

First Number:   -6.333
Second Number:  -2.1
#1 Expanded:      0000 0001 1001 0000
#1 Exponent:      2
#2 Expanded:      0000 0001 0000 0000
#2 Exponent:      1
Product:          0000 0001 1001 0000
Product Exponent: 3
Pruduct Packed:   0100 1111

* * * * * * * * * * * * * * * * *

Sum: 12.50000000

* * * * * * * * * * * * * * * * *



* * * * * * * * * * * * * * * * *

First Number:   -6.333
Second Number:  -2.1
#1 Expanded:      0000 0001 1001 0000
#1 Exponent:      2
#2 Expanded:      0000 0001 0000 0000
#2 Exponent:      1
```

```
Product:          0000 0001 1001 0000
Product Exponent: 3
Pruduct Packed:   0100 1111

* * * * * * * * * * * * * * * * *

Sum: 12.50000000

* * * * * * * * * * * * * * * * *
```

Source Code

main and helper functions

```
#include <cstdio>
#include "./float_software.h"

void get_number(char*);
void test_atof(char*);
void test_ftoa(char*);
void test_add(char*);
void test_multiply(char*);

const int INPUT_BUFFER_SIZE = 40;

int main() {
      char user_input[INPUT_BUFFER_SIZE];
      int selection;
      bool exit = false;
      char trash;

      printf(
                  "\n\n**** Joseph Morgan's test main for software implementation
of floats ****\n\n"
                  );
      get_number(user_input);

      while (!exit) {
            printf(
                        "Menu options:\n\n"
                        "     1 - test atof\n"
                        "     2 - test ftoa\n"
                        "     3 - test add\n"
                        "     4 - test multiply\n"
                        "     5 - input new number\n"
                        "     0 - exit\n\n"
                        "Selection: "
                        );

            while ((trash = getchar()) != '\n' && trash != EOF)
                  /* discard */;

            scanf("%i", &selection);
            printf("\n");
            switch (selection) {
                  case 0 :
                        exit = true;
                        break;

                  case 1 :
                        test_atof(user_input);
                        break;
                  case 2 :
                        test_ftoa(user_input);
                        break;

                  case 3 :
                        test_add(user_input);
                        break;

                  case 4 :
                        test_multiply(user_input);
                        break;

                  case 5 :
                        get_number(user_input);
                        break;

                  default :
                        printf("That didn't seem like an acceptable option\n");
            }
      }
}

void get_number(char* user_input) {
      printf("Please enter a number to use in the test: ");
      scanf("%s", user_input);
      printf("\n");
}

void test_atof(char* user_input) {
      unsigned char f = software_atof(user_input);

      printf("Converting %s to floating-point format...\n\n", user_input);

      printf("* * * * * * * * * * * * * * * * *\n\n");
      print8bits(f);
      printf("\n* * * * * * * * * * * * * * * * *\n\n");
}

void test_ftoa(char* user_input) {
      printf(
"This will show the exact value of your number as it's stored in memory in
floating point format\n"
"You can see how accurately your number is being represented in 8 bits!\n\n"
"Converting %s to floating-point and back to a string...\n\n", user_input
                  );

      unsigned char f = software_atof(user_input);
      char buffer[INPUT_BUFFER_SIZE];
      software_ftoa(f, buffer);

      printf("* * * * * * * * * * * * * * * * *\n\n");
      printf("%s\n\n", buffer);
      printf("* * * * * * * * * * * * * * * * *\n\n");
}
```

```
void test_add(char* user_input) {
      char second_input[INPUT_BUFFER_SIZE];
      unsigned char first_number;
      unsigned char second_number;
      unsigned char sum;
      char str_sum[INPUT_BUFFER_SIZE];;

      printf("What would you like to add to your number?\n");
      get_number(second_input);

      printf("\n"
                  "* * * * * * * * * * * * * * * * *\n\n");
      first_number = software_atof(user_input);
      second_number = software_atof(second_input);

      printf("First Number:   %s\n", user_input);
      printf("Second Number:  %s\n", second_input);

      sum = software_float_add(first_number, second_number);
      software_ftoa(sum, str_sum);

      printf("\n* * * * * * * * * * * * * * * * *\n\n");
      printf("Sum: %s\n\n", str_sum);
      printf("* * * * * * * * * * * * * * * * *\n\n");
}

void test_multiply(char* user_input) {
      // TODO
      printf("User input: %s", user_input);
}
```

Implementation

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "float_software.h"

unsigned char software_atof (char* input_str) {
      char formatted_str[20];
      bool negative = false;

      // Set negative flag
      if (input_str[0] == '-') {
            negative = true;
            strcpy(formatted_str, input_str + 1);
      } else {
            strcpy(formatted_str, input_str);
      }
      int len = strlen(formatted_str);

      // Variables to hold string representation of whole and fractional parts
      // of the number.
#     define S_WHOLE_SIZE 3
#     define S_FRACT_SIZE 9
      char string_whole[S_WHOLE_SIZE] = "00";
      char string_fract[S_FRACT_SIZE] = "00000000";

      // Store the index of the decimal in x
      int x = 0;
      while (formatted_str[x] != '.') {
            ++x;
            if (x > len) printf("You need to include a decimal point - exiting\n"),
exit(1);
      }

      // Reverse whole part of number, store as string.
      // ex. 3.1416 would be stored as: 30
      for (int i = x - 1, j = 0; i >= 0; --i, ++j) {
            string_whole[j] = formatted_str[i];
      }

      // Reverse fractional part of number, store as string.
      // ex: 3.1416 would be stored as: 00006141
      for (int i = x + 1, j = S_FRACT_SIZE - 2; i < len; ++i, --j) {
            string_fract[j] = formatted_str[i];
      }

      // Convert string representation of whole part to internal representation of
      // the integer.
      unsigned char integer_whole = 0;
      for (int i = 0; i < S_WHOLE_SIZE - 1; ++i) {
            integer_whole += (string_whole[i] - '0') * software_float_pow(10, i);
      }

      // Convert string representation of fractional part to binary representation
      // of the number times 100,000,000 in decimal
      unsigned long integer_fract = 0;
      for (int i = 0; i < S_FRACT_SIZE - 1; ++i) {
            integer_fract += (string_fract[i] - '0') * software_float_pow(10, i);
      }

      // This is some wacky voodoo magic. To manually find the bits needed to
      // represent the fractional point of a decimal number, we subtract by known
      // powers of two if the fractional part is greater than the known power of
two.
      // Our integer_fract contains (fractional part of the integer * 1,000,000),
which
      // means we can subtract by (known powers of two * 1,000,000) to find which
      // known powers of two it contains. Then mask off those specific bits.
      unsigned long power_of_two = 50000000;
      unsigned char binary_fract = 0; // Bits to the right of decimal point.
      unsigned char mask = 0x80; // initial bit to potentially mask.
      for (int i = 0; i < S_FRACT_SIZE - 1; ++i) {
            if (integer_fract >= power_of_two) {
                  binary_fract |= mask;
                  integer_fract -= power_of_two;
            }

            mask >>= 1;
            power_of_two >>= 1; // This is the same as multiplying by .5, ignoring
remainder
      }

      //print8bits(integer_whole);
      //print8bits(binary_fract);

      unsigned short buffer = 0;
      buffer = integer_whole;
      buffer <<= 8;
      buffer |= binary_fract;

      //print16bits(buffer);

      // Normalize the fraction - find the first 1.
      int exponent = 7;
      unsigned short bitfinder_mask = 0x8000;
      while (!(buffer & bitfinder_mask)) {
            --exponent;
            bitfinder_mask >>= 1;
      }

      // Final packing of the float
      unsigned char the_float = 0;

      buffer <<= (7 - exponent);
      buffer >>= 8;
      the_float = buffer;
      the_float &= 0x78;
      exponent += 4;
      the_float |= exponent;
      if (negative) the_float |= 0x80;

      return the_float;
}

void software_ftoa (unsigned char f, char* strOut) {
      // This function copy/pasted from assignment
      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
}

void print_all_in_range() {
      unsigned char byte;
      char output[40];

      for (int i = 0; i <= 255; ++i) {
            byte = i;
            software_ftoa(i, output);
            printf("%s", output);
            if (i != 255) printf(", ");
      }
}

void print8bits (unsigned char buffer) {
      for (unsigned char mask = 0x80; mask; mask >>= 1) {
            if (mask & buffer) printf("1");
            else printf("0");
      }
      printf("\n");
}

void print16bits (unsigned short buffer) {
      for (unsigned short mask = 0x8000; mask; mask >>= 1) {
            if (mask & buffer) printf("1");
            else printf("0");
      }
      printf("\n");
}

void print32bits (unsigned long buffer) {
      for (unsigned long mask = 0x800000; mask; mask >>= 1) {
            if (mask & buffer) printf("1");
            else printf("0");
      }
      printf("\n");
}
```

```
unsigned char software_float_add(unsigned char float1, unsigned char float2) {
      unsigned short buffer1;
      unsigned short buffer2;
      unsigned short sum_buffer;
      int exponent;
      bool is_negative_float1 = false;
      bool is_negative_float2 = false;
      bool is_negative_sum = false;

      if (float1 & 0x80) is_negative_float1 = true;
      if (float2 & 0x80) is_negative_float2 = true;

      exponent = (float1 & 0x07) - 4;
      float1 &= 0x78;
      float1 |= 0x80;

      buffer1 = float1;
      buffer1 <<= 8;
      buffer1 >>= (7 - exponent);

      exponent = (float2 & 0x07) - 4;
      float2 &= 0x78;
      float2 |= 0x80;

      buffer2 = float2;
      buffer2 <<= 8;
      buffer2 >>= (7 - exponent);


      // Maybe I'm totally missing something here, but I belive the addition of
      // some logic to deal with negative addends is required for this to
      // function properly.
      if (is_negative_float1 || is_negative_float2) {

            if (is_negative_float1 && is_negative_float2) {
                  is_negative_sum = true;
                  sum_buffer = buffer1 + buffer2;
            }

            else if (buffer1 >= buffer2) {
                  sum_buffer = buffer1 - buffer2;
                  if (is_negative_float1) is_negative_sum = true;
            }

            else {
                  sum_buffer = buffer2 - buffer1;
                  if (is_negative_float2) is_negative_sum = true;
            }

      }

      else sum_buffer = buffer1 + buffer2;

      printf("#1 Expanded:    ");
      print16bits(buffer1);
      printf("#2 Expanded:    ");
      print16bits(buffer2);
      printf("Sum:            ");
      print16bits(sum_buffer);

      exponent = 7;
      unsigned short bitfinder_mask = 0x8000;
      while (!(sum_buffer & bitfinder_mask)) {
            --exponent;
            bitfinder_mask >>= 1;
      }

      if (exponent > 3 || exponent < -4) {
            printf("Exponent of sum not within valid bounds.");
            exit(0);
      }

      unsigned char the_float;

      sum_buffer <<= (7 - exponent);
      sum_buffer >>= 8;
      printf("Sum Normalized: ");
      print16bits(sum_buffer);
      the_float = sum_buffer;
      the_float &= 0x78;
      exponent += 4;
      the_float |= exponent;
      if (is_negative_sum) the_float |= 0x80;
      printf("Sum Packed:     ");
      print8bits(the_float);

      return the_float;
}

unsigned char software_float_multiply(unsigned char, unsigned char) {
      //TODO
      return 0x00;
}

int software_float_pow(int base, int exponent) {
      int x = 1;
      for (int i = 0; i < exponent; ++i) {
            x *= base;
      }

      return x;
}
```