Quinn Roemer

CISP - 440

Assignment 2

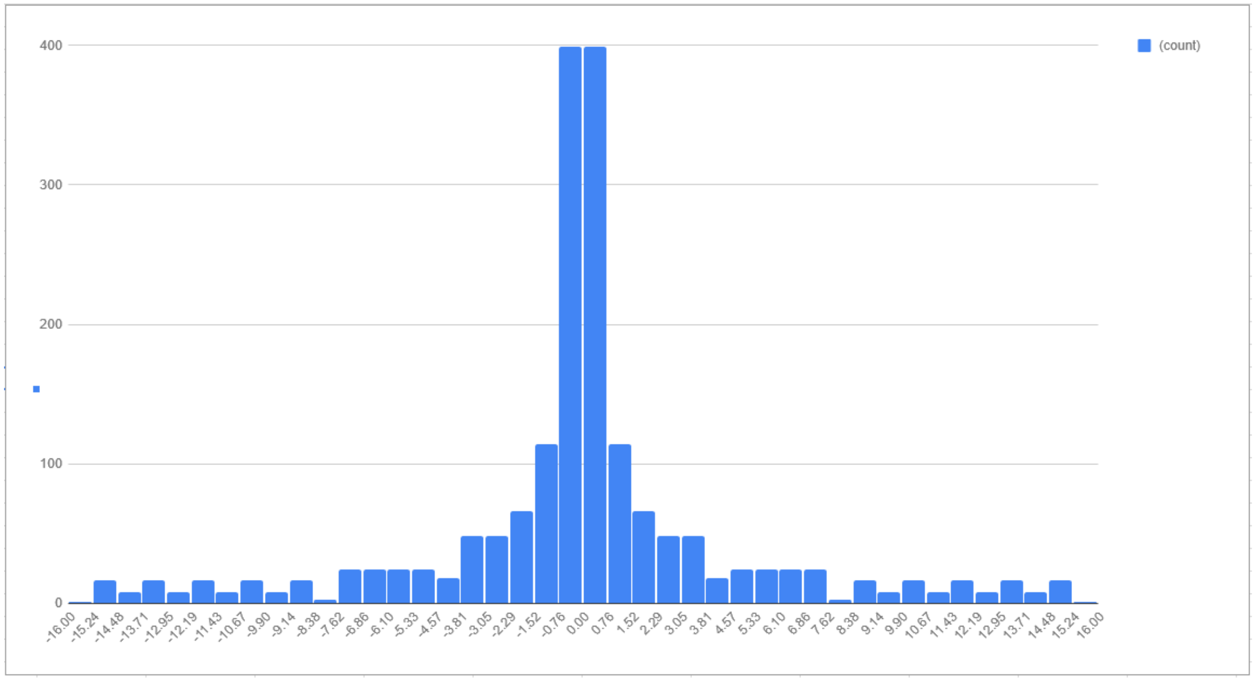
9/27/2018

Part 0 - Floating Point Implementation

Description:

The goal for this section of the assignment was to write code that was able to successfully multiply and add two packed floats together. To do this, I had to unpack, normalize, perform the desired operation, and then finally repack with the correct exponent. In addition, we were required to provide a graph that details the spread of all possible numbers that our 8 bit float could represent.

Graph:



Please note: The graph works by representing the spread of numbers in ranges. The bar gets taller for each possible number that can be represented in that range.

Source Code:

```
//Written by Quinn Roemer. Based on code by Professor Ross
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <string>
#include <math.h>
#include <stack>
```

```
using namespace std;

#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");
    }
}

//Packs a real number into a float.
```

```
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);

    if (i_whole > 15) {   // (2 ^ n) - 1   in general
        printf("atof conversion Overflow %s", str);
        exit(0);
    }

    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);

    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;
    }

    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 packed float into real numbers.
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);

    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;

    // 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
}
```

//My code starts here.

```
//Converts binary into decimal. Including excess.
int bin2Dec(string number)
{
    int result = 0;
    int innerCount = 0;
    for (int count = number.length(); count > 0; count--)
    {
        if (number[count - 1] == '1')
        {
               result = result + pow(2, innerCount);
        }

        innerCount++;
    }

    return result - 4;
}

//Converts an exponent into binary. Including excess.
string getExp(int exp)
{
    exp = exp + 4;

    int rDivide, remainder, size;

    string result;

    stack<int> rStore;

    while (exp != 0)
    {
        rDivide = exp / 2;
        remainder = exp % 2;
        exp = rDivide;
        rStore.push(remainder);
    }

    size = rStore.size();

    //Converting the stack to a string.
    for (int count = 0; count < size; count++)
    {
        remainder = rStore.top();

        if (remainder == 1)
        {
               result += '1';
        }
        else
        {
               result += '0';
        }
        rStore.pop();
    }
    return result;
}

//Adds two strings
string addString(string one, string two)
{
    string sResult;

    int value1 = bin2Dec(one) + 4;
    int value2 = bin2Dec(two) + 4;

    sResult = getExp(value1 + value2 - 4);

    while (sResult.length() < 16)
    {
        sResult = "0" + sResult;
    }
    return sResult;
}

//Subtracts two strings.
string subString(string one, string two)
{
    //String one is larger.
    string sResult;

    int value1 = bin2Dec(one) + 4;
    int value2 = bin2Dec(two) + 4;

    sResult = getExp(value1 - value2 - 4);

    while (sResult.length() < 16)
    {
        sResult = "0" + sResult;
    }
    return sResult;
}

//Multiplies two binary strings.
string multString(string one, string two)
{
    string sResult;

    int value1 = bin2Dec(one) + 4;
    int value2 = bin2Dec(two) + 4;

    sResult = getExp((value1 * value2) - 4);

    //For debugging purposes.
    //cout << sResult << endl;

    while (sResult.length() < 32)
    {
        sResult = "0" + sResult;
    }

    return sResult;
}

//Adds two packed floats.
unsigned char addFloats(unsigned char f1, unsigned char f2)
{
    //Store all components of the packed floats.
    string signf1, signf2;
    string mantissaf1, mantissaf2;
    string exponentf1, exponentf2;
    string answer;

    //Holds the new Mantissa after addition.
    string newMantissa;

    int x, y, innercount = 0;

    //Char mask = 128.
    unsigned char mask = 0x80;

    //The answer to return.
    unsigned char theFloat = 0;

    //Negative flags.
    bool bothNegative = false;
    bool oneNegative = false;

    //For debugging purposes.
    //print8bits(f1);
    //cout << endl;
    //print8bits(f2);
    //cout << endl;

    //Grabbing all components of the packed float.
    for (int count = 0; count < 8; count++)
    {
        if (count < 1)
        {
                if (mask & f1)
                {
                        signf1.append("1");
                }
                else
```

```
            {
                    signf1.append("0");
            }
        }
        else if (count >= 1 && count < 5)
        {
            if (mask & f1)
            {
                    mantissaf1.append("1");
            }
            else
            {
                    mantissaf1.append("0");
            }
        }
        else
        {
            if (mask & f1)
            {
                    exponentf1.append("1");
            }
            else
            {
                    exponentf1.append("0");
            }
        }

        mask >>= 1;
    }

    mask = 0x80;

    //Grabbing all components of the packed float.
    for (int count = 0; count < 8; count++)
    {
        if (count < 1)
        {
            if (mask & f2)
            {
                    signf2.append("1");
            }
            else
            {
                    signf2.append("0");
            }
        }
        else if (count >= 1 && count < 5)
        {
            if (mask & f2)
            {
                mantissaf2.append("1");
            }
            else
            {
                mantissaf2.append("0");
            }
    }
    else
    {
            if (mask & f2)
            {
                exponentf2.append("1");
            }
            else
            {
                exponentf2.append("0");
            }
    }

    mask >>= 1;
}

//For debugging purposes.

/*cout << "Sign: " << signf1 << endl;
cout << "Mantissa: " << mantissaf1 << endl;
cout << "Exp: " << exponentf1 << endl << endl;

cout << "Sign: " << signf2 << endl;
cout << "Mantissa: " << mantissaf2 << endl;
cout << "Exp: " << exponentf2 << endl;*/

//Set negative flags.
if (signf1 == "1" && signf2 == "1")
{
    bothNegative = true;
}
else if (signf1 == "1" || signf2 == "1")
{
    oneNegative = true;
}

x = bin2Dec(exponentf1);
y = bin2Dec(exponentf2);

//Add leading 1 to Mantissa.
mantissaf1 = "1" + mantissaf1;
mantissaf2 = "1" + mantissaf2;

//Shift left by 11.
```

```
for (int count = 0; count < 11; count++)
{
    mantissaf1.append("0");
    mantissaf2.append("0");
}

//Shift right by 7 - exp.
for (int count = 0; count < 7 - x; count++)
{
    mantissaf1 = "0" + mantissaf1;
    mantissaf1.pop_back();
}
for (int count = 0; count < 7 - y; count++)
{
    mantissaf2 = "0" + mantissaf2;
    mantissaf2.pop_back();
}

//For Debuging Purposes.
cout << "#1 Expanded: " << mantissaf1 << endl;
cout << "#2 Expanded: " << mantissaf2 << endl;

//Add modified Mantissa's.
if (oneNegative == false)
{
    answer = addString(mantissaf1, mantissaf2);
}
else
{
    //Find out what mantissa is the bigger number.
    for (unsigned int count = 0; count < mantissaf1.length(); count++)
    {
            if (mantissaf1[count] == '1' && mantissaf2[count] == '0')
            {
                    //f1 is bigger and negative.
                    if (signf1 == "1")
                    {
                            answer = subString(mantissaf1, mantissaf2);
                            break;
                    }
                    //f1 is bigger and positive.
                    else
                    {
                            answer = subString(mantissaf1, mantissaf2);
                            oneNegative = false;
                            break;
                    }
            }
            else if (mantissaf1[count] == '0' && mantissaf2[count] == '1')
            {
```

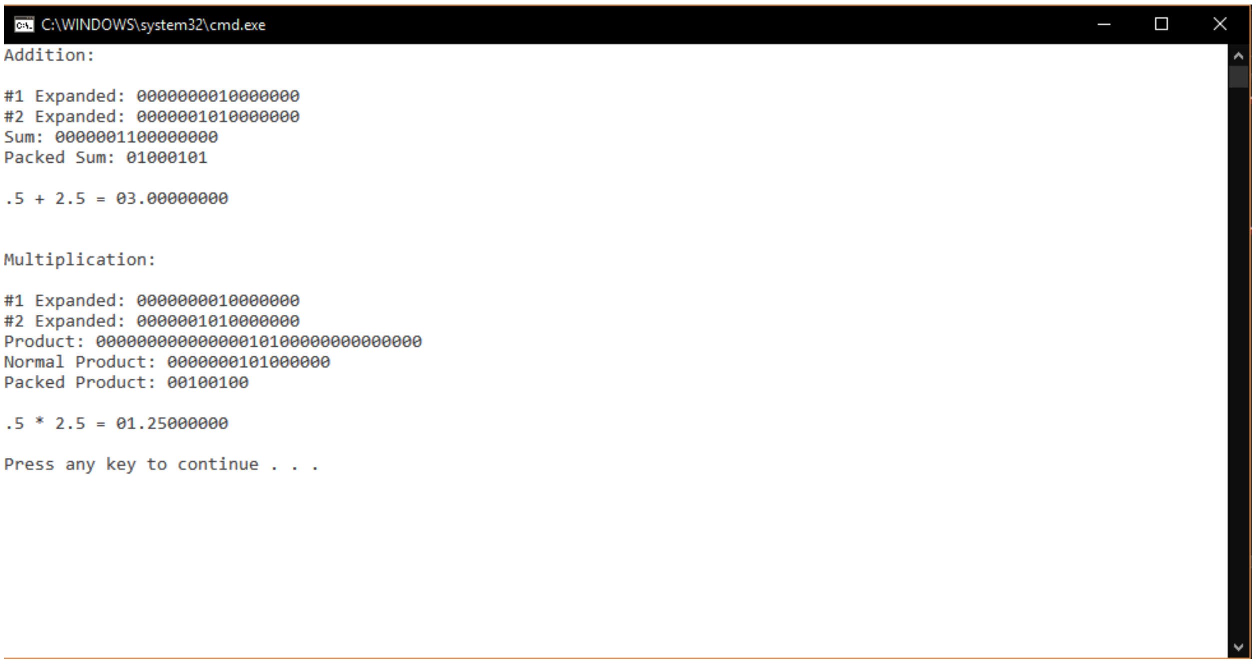
```
                        //f2 is bigger and negative.
                        if (signf2 == "1")
                        {
                                answer = subString(mantissaf2, mantissaf1);
                                break;
                        }
                        //f2 is bigger and positive.
                        else
                        {
                                answer = subString(mantissaf2, mantissaf1);
                                oneNegative = false;
                                break;
                        }
                }
                //F1 = F2. Thus the answer is 0.
                if (count == 15)
                {
                        theFloat = 0;
                        return theFloat;
                }
        }
    }

    //For debugging purposes.
    cout << "Sum: " << answer << endl;

    //Finding the position of the first 1 in the binary string.
    for (int count = 0; count < 16; count++)
    {
        if (answer[count] == '1')
        {
                x = count + 1;
                break;
        }
    }

    //Exponent equals x - 8.
    y = 8 - x;

    //Overflow and Underflow check.
    {
        if (y >= 4 || y <= -5)
        {
                cout << "An Overflow or Underflow occured in the addition. The answer is not 
valid." << endl;
                theFloat = 0;
                return theFloat;
        }
    }
    //Grabbing the mantissa.
    for (int count = 0; count < 4; count++)
    {
        newMantissa = newMantissa + answer[x];
        x++;
    }

    //Moving to correct position.
    for (int count = 0; count < 3; count++)
    {
        newMantissa.append("0");
    }

    x = bin2Dec(newMantissa) + 4;

    if (oneNegative == false && bothNegative == false)
    {
        //For positive results.
        y = y + 4;
        theFloat = x + y;
    }
    else
    {
        //For negative results.
        y = y + 4;
        theFloat = 128 + x + y;
    }

    //For debugging Purposes.
    //cout << "New MAN:" << newMantissa << endl;
    //cout << "EXP BIN: " << newExp << endl;

    cout << "Packed Sum: ";
    print8bits(theFloat);
    cout << endl;

    return theFloat;
}

//Multiplies two packed floats.
unsigned char multiplyFloats(unsigned char f1, unsigned char f2)
{
    string signf1, signf2;
    string mantissaf1, mantissaf2;
    string exponentf1, exponentf2;
    string answer;

    //Holds the new Mantissa after multiplication.
    string newMantissa;
```

```
int x, y, innercount = 0;

//Char mask = 128.
unsigned char mask = 0x80;

//The answer to return.
unsigned char theFloat = 0;

//Negative flags.
bool bothNegative = false;
bool oneNegative = false;

//Grabbing all components of the packed float.
for (int count = 0; count < 8; count++)
{
    if (count < 1)
    {
            if (mask & f1)
            {
                    signf1.append("1");
            }
            else
            {
                    signf1.append("0");
            }
    }
    else if (count >= 1 && count < 5)
    {
            if (mask & f1)
            {
                    mantissaf1.append("1");
            }
            else
            {
                    mantissaf1.append("0");
            }
    }
    else
    {
            if (mask & f1)
            {
                    exponentf1.append("1");
            }
            else
            {
                    exponentf1.append("0");
            }
    }

    mask >>= 1;
}

mask = 0x80;

//Grabbing all components of the packed float.
for (int count = 0; count < 8; count++)
{
    if (count < 1)
    {
            if (mask & f2)
            {
                    signf2.append("1");
            }
            else
            {
                    signf2.append("0");
            }
    }
    else if (count >= 1 && count < 5)
    {
            if (mask & f2)
            {
                    mantissaf2.append("1");
            }
            else
            {
                    mantissaf2.append("0");
            }
    }
    else
    {
            if (mask & f2)
            {
                    exponentf2.append("1");
            }
            else
            {
                    exponentf2.append("0");
            }
    }

    mask >>= 1;
}

//Set negative flags.
if (signf1 == "1" && signf2 == "1")
{
    bothNegative = true;
}
else if (signf1 == "1" || signf2 == "1")
```

```
{
    oneNegative = true;
}

//Converting binary real number.
x = bin2Dec(exponentf1);
y = bin2Dec(exponentf2);

//Add leading 1 to Mantissa.
mantissaf1 = "1" + mantissaf1;
mantissaf2 = "1" + mantissaf2;

//Shift left by 11.
for (int count = 0; count < 11; count++)
{
    mantissaf1.append("0");
    mantissaf2.append("0");
}

//Shift right by 7 - exp.
for (int count = 0; count < 7 - x; count++)
{
    mantissaf1 = "0" + mantissaf1;
    mantissaf1.pop_back();
}
for (int count = 0; count < 7 - y; count++)
{
    mantissaf2 = "0" + mantissaf2;
    mantissaf2.pop_back();
}

//For debugging purposes.
cout << "#1 Expanded: " << mantissaf1 << endl;
cout << "#2 Expanded: " << mantissaf2 << endl;

//Multiply the two Mantissa's.
newMantissa = multString(mantissaf1, mantissaf2);

cout << "Product: " << newMantissa << endl;

//Grabbing rollover.
for (int count = 0; count < 32; count++)
{
    if (newMantissa[count] == '1')
    {
            x = 0;
            x = 15 - count;
            y = 0;
            y = count;
            break;
```

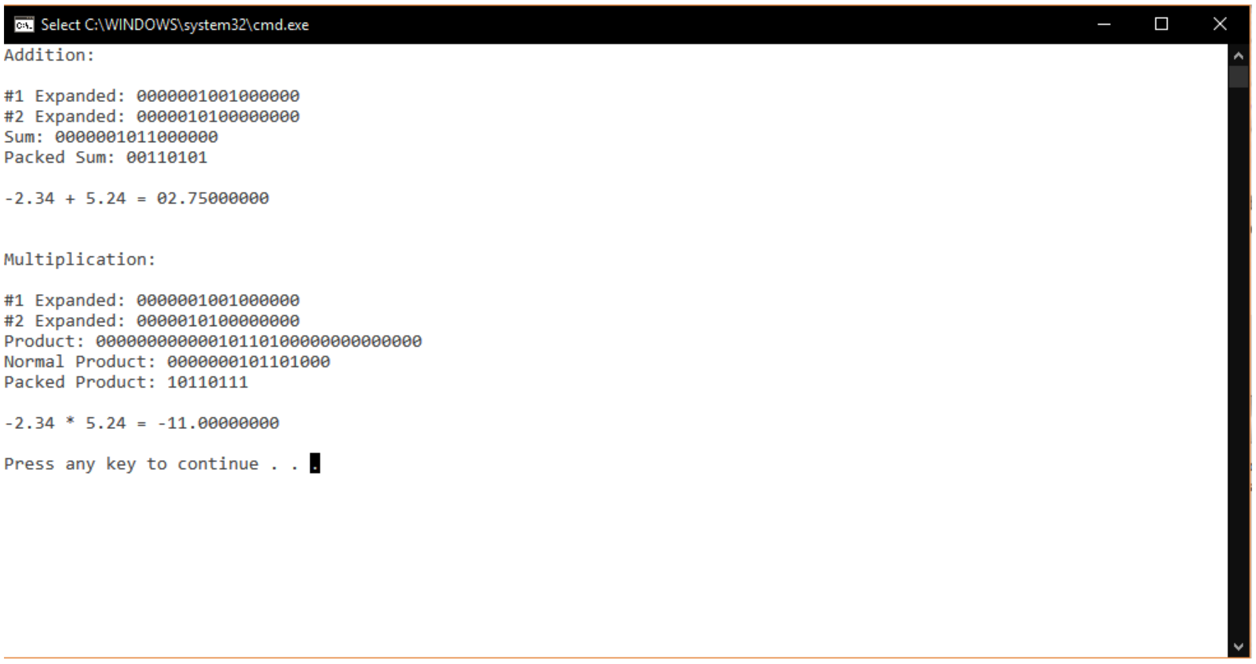
```
        }
    }

    //Checking for overflow or underflow
    if (x >= 4 || x <= -5)
    {
        cout << "Overflow or Underflow occured in the multiplication" << endl;
        return theFloat;
    }

    //For debugging purpose.
    //cout << "Rollover: " << x << endl;

    //Normalize result.
    for (int count = 0; count < 8 + x; count++)
    {
        newMantissa.pop_back();
    }

    while (newMantissa.length() > 16)
    {
        newMantissa.erase(0, 1);
    }

    //For debugging purpose.
    cout << "Normal Product: " << newMantissa << endl;

    //Grabbing the Mantissa.
    innercount = 8;
    for (int count = 0; count < 4; count++)
    {
        if (newMantissa[innercount] == '1')
        {
                answer.append("1");
        }
        else
        {
                answer.append("0");
        }
        innercount++;
    }
    for (int count = 0; count < 3; count++)
    {
        answer.append("0");
    }

    innercount = bin2Dec(answer);

    //Packing the float.
    if (oneNegative == true)
    {
        theFloat = 128 + (innercount + 4) + (x + 4);
    }
    else
    {
        theFloat = (innercount + 4) + (x + 4);
    }

    cout << "Packed Product: ";
    print8bits(theFloat);
    cout << endl;
    return theFloat;
}

int main()
{
    //Numbers to perform actions on.
    char strIn1[40] = ".5";
    char strIn2[40] = "2.5";
    char strOut[40];

    //Used to hold packed floats.
    unsigned char f1;
    unsigned char f2;
    unsigned char f3;

    //Converting both numbers to packed floats.
    f1 = my_atof(strIn1);
    f2 = my_atof(strIn2);

    //Adding the floats and printing out.
    cout << "Addition:" << endl << endl;
    f3 = addFloats(f1, f2);
    my_ftoa(f3, strOut);
    cout << endl << strIn1 << " + " << strIn2 << " = " << strOut << endl << endl;

    //Converting packed float to real number.
    cout << "\nMultiplication:" << endl << endl;
    f3 = multiplyFloats(f1, f2);
    my_ftoa(f3, strOut);
    cout << endl << strIn1 << " * " << strIn2 << " = " << strOut << endl << endl;
}
```

Output:



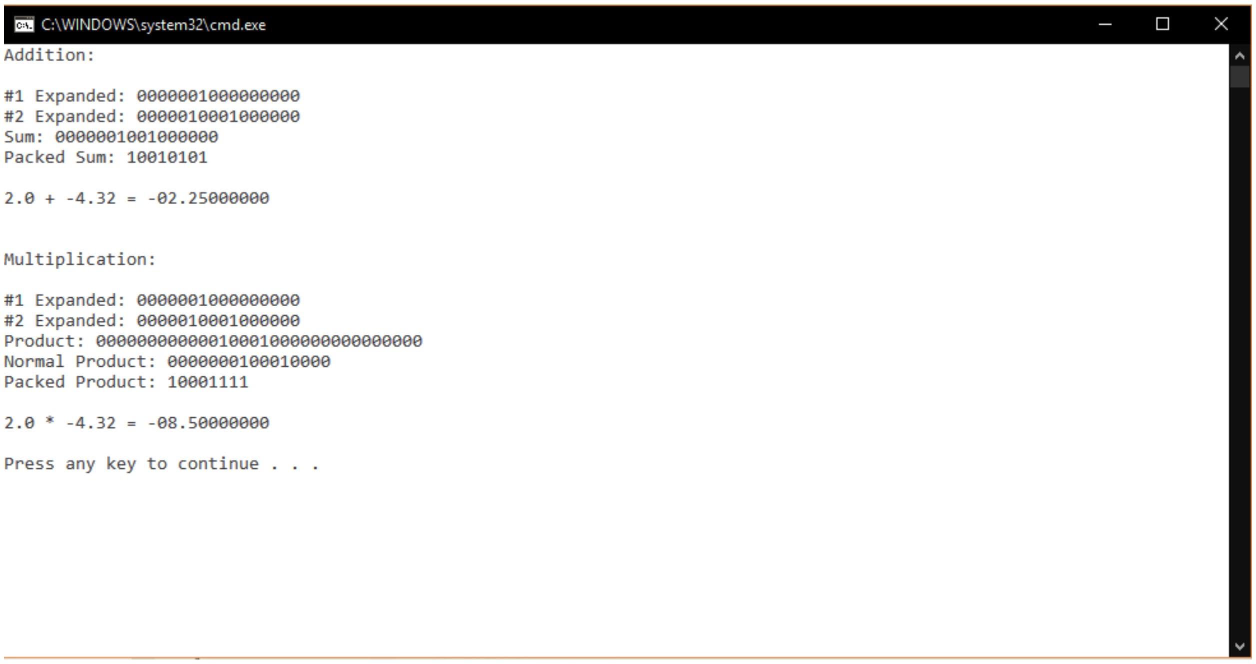
```
Addition:

#1 Expanded: 0000000010000000
#2 Expanded: 0000001010000000
Sum: 0000001100000000
Packed Sum: 01000101

.5 + 2.5 = 03.00000000


Multiplication:

#1 Expanded: 0000000010000000
#2 Expanded: 0000001010000000
Product: 00000000000000101000000000000000
Normal Product: 0000000101000000
Packed Product: 00100100

.5 * 2.5 = 01.25000000

Press any key to continue . . .
```

(1 of 5)



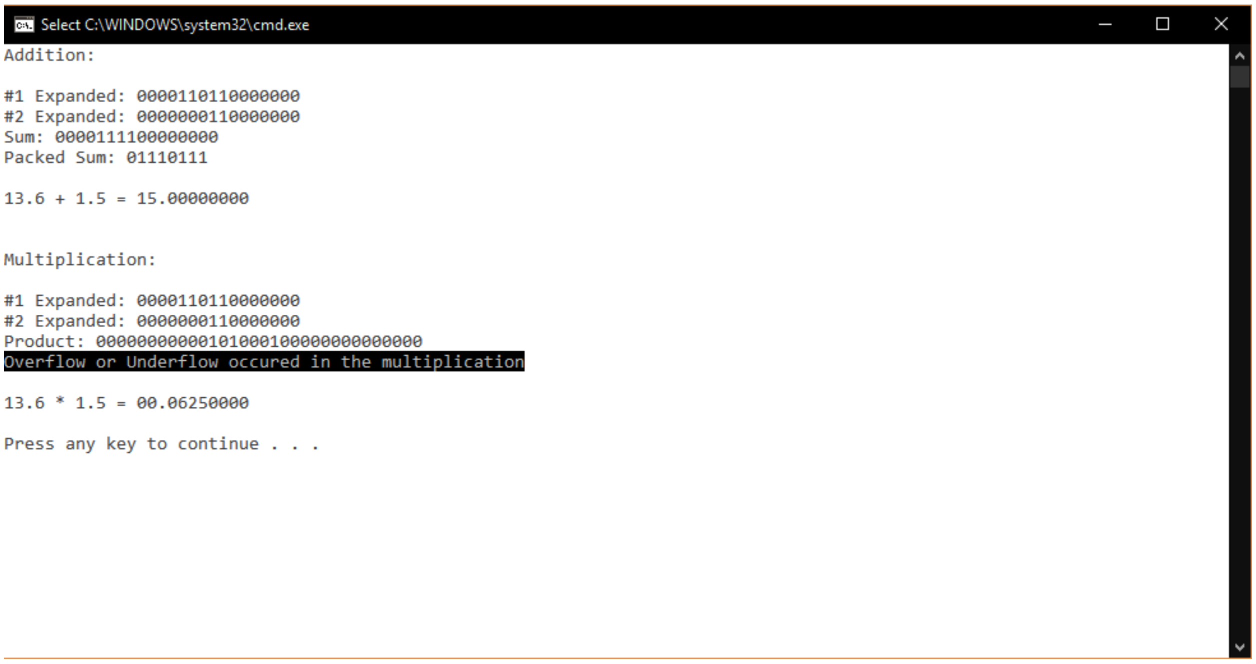
```
Addition:

#1 Expanded: 0000001001000000
#2 Expanded: 0000010100000000
Sum: 0000001011000000
Packed Sum: 00110101

-2.34 + 5.24 = 02.75000000


Multiplication:

#1 Expanded: 0000001001000000
#2 Expanded: 0000010100000000
Product: 00000000000010110100000000000000
Normal Product: 0000000101101000
Packed Product: 10110111

-2.34 * 5.24 = -11.00000000

Press any key to continue . .
```

(2 of 5)



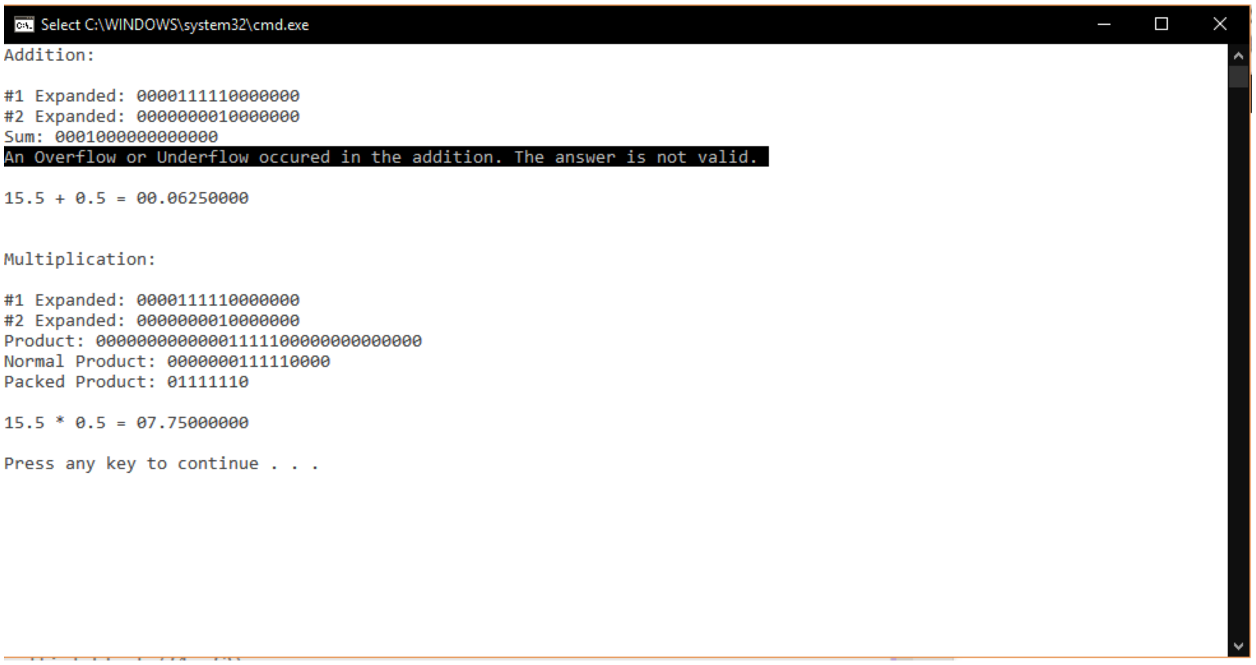
```
Addition:

#1 Expanded: 0000001000000000
#2 Expanded: 0000010001000000
Sum: 0000001001000000
Packed Sum: 10010101

2.0 + -4.32 = -02.25000000


Multiplication:

#1 Expanded: 0000001000000000
#2 Expanded: 0000010001000000
Product: 00000000000100010000000000000000
Normal Product: 0000000100010000
Packed Product: 10001111

2.0 * -4.32 = -08.50000000

Press any key to continue . . .
```

(3 of 5)



```
Addition:

#1 Expanded: 0000110110000000
#2 Expanded: 0000000110000000
Sum: 0000111100000000
Packed Sum: 01110111

13.6 + 1.5 = 15.00000000


Multiplication:

#1 Expanded: 0000110110000000
#2 Expanded: 0000000110000000
Product: 00000000000101000100000000000000
Overflow or Underflow occured in the multiplication

13.6 * 1.5 = 00.06250000

Press any key to continue . . .
```

(4 of 5)



```
Addition:

#1 Expanded: 0000111110000000
#2 Expanded: 0000000010000000
Sum: 0001000000000000
An Overflow or Underflow occured in the addition. The answer is not valid.

15.5 + 0.5 = 00.06250000


Multiplication:

#1 Expanded: 0000111110000000
#2 Expanded: 0000000010000000
Product: 00000000000001111100000000000000
Normal Product: 0000000111110000
Packed Product: 01111110

15.5 * 0.5 = 07.75000000

Press any key to continue . . .
```

(5 of 5)

Conclusion

Once again, I am happy to complete an assignment that involves programming. However, this one proved to be more challenging than the last. Mostly because the concept was much more difficult to grasp. Yet, after studying and reviewing my notes, meticulously, I managed to figure out the process. Looking forward to the next assignment!