COS 335 Spring 2011
Assignment 2
Due Thursday Feb 3

1.(2 pts) Decode the following IEEE 32-bit floating point numbers and express them using conventional decimal notation. Show your work. There are no infinities, denormalized numbers, or NANs in the list. For (d) you can express your answer with 3-4 decimal places.

a) `3DEC0000` b) `CC880000` c) `3E4CCCCD` d) `C1880000`

2. (2 pts) Encode the following decimal numbers as IEEE 32-bit floating point numbers. Show your work and express the results in hex.
a) `-6.75` b) `32777` c) `16777216.5` d) `1031.4644`

3. (1 pt) Explain why it is that you can encode either 67108864 or 67108872 as an IEEE 32-bit float, but not 67108868.

4. (1 pt) Text Ch. 2 problem 2.3 p.60
5. (1 pt) Text Ch. 2 problem 2.4 p.61
6. (1 pt) Text Ch. 3 problem 3.1 p.105
7. (1 pt) Text Ch. 3 problem 3.3 p.105

8 (1 pt) The associative rule of addition states that for any a, b, and c; (a+b)+c = a+(b+c). This is not necessarily true for computer addition of floating point numbers. The following program is designed to demonstrate this as well as another peculiarity of floating point arithmetic executed on Intel processors.

The C++ program float.cpp prompts for a,b and c. The code has a set of values for a, b and c in comments that will produce incorrect results for some addition sequences.

For this assignment run the C++ code and find two additional sets of values for a, b, and c that cause the computer to show incorrect results for at least one of the three additions., and find two <u>non-integer</u> sets of values where all additions are correct.

The "other peculiarity" of Intel floating point arithmetic is that results from floating point arithmetic may vary depending on whether or not intermediate values have been stored to memory during the computation. The reason for this is that the Intel x87 registers where floating point arithmetic is performed are all 80 bits wide. An IEEE 32-bit float (type float in C), however, is only 32 bits wide. When a float is loaded into an x87 register in preparation for an arithmetic operation, the processor converts it to 80 bits and all computations are performed using 80-bit arithmetic.

The java code in FloatTest.java will demonstrate the non-associativity of floating point arithmetic, but does not have the other peculiarity relating to storage of intermediate results. This is because the Java instructions are executed against a virtual machine that does NOT do 80-bit arithmetic internally.

In the following example, code is generated by the Microsoft Visual C++ 6 compiler. The mnemonics mean the following:

fld	Floating point load from mem, converts values to 80 bits internally
fadd	Floating point add from mem, values are converted to 80 bits before the addition
fstp	Float point store and pop. Value is saved to mem from x87 top-of-stack and the value is

removed from the register stack. Values are converted to memory size on store.

We have not covered enough assembly language for you to make sense of the following instructions yet, but just read an expression such as DWORD PTR _a$[ebp] as a reference to the variable a. Thus the first instruction below means "load the variable a from memory."

In the assignment expression **`d = (a+b)+c`** all computation takes place in the 87 in 80 bits:

```
fld   DWORD PTR _a$[ebp]
fadd  DWORD PTR _b$[ebp]
fadd  DWORD PTR _c$[ebp]
fstp  DWORD PTR _d$[ebp]
```

As a, b, and c are loaded from memory into the x87 registers they are converted to 80 bits. The expression is effectively evaluated using the type long double for all operations. When the result is stored back to memory in d it is converted back to a 32-bit IEEE float from an 80-bit IEEE float.

However, in the following sequence:

```
d = a + b;
cout << d;
d += c;
```

We have

```
fld   DWORD PTR _a$[ebp]
fadd  DWORD PTR _b$[ebp]
fstp  DWORD PTR _d$[ebp]
; d is stored in memory in prep for call to cout
fld   DWORD PTR _d$[ebp]
fadd  DWORD PTR _c$[ebp]
fstp  DWORD PTR _d$[ebp]
```

Here the intermediate value a+b is stored to memory and converted to a 32-bit float. After the call to cout it is again loaded into the 87 and converted to 80 bits prior to the add of c. However, the intermediate result has lost its 80 bits of precision.

float.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;

int main(int argc, char* argv[])
{
    float a, b, c, d, e;
    cout.setf(ios::fixed | ios::showpoint);
    cout << "Enter three floating point numbers a, b and c: ";
    cin >> a >> b >> c;

    //echo inputs
    cout << "a=" << a << '\t'<< "b=" << b << '\t' << " c=" << c << '\n';

    // try for example a = 0.1  b = 5.812  c = 78.876;
    // or a = 0.1 b = 7.6  c = 23.879

    // evaluate addition without intermediate storage of results
    cout << "Additions without intermediate storage of results\n";
    d = (a+b)+c;
    e = a+(b+c);
    cout << "(a+b)+c = " << setprecision(8) << d << '\n';
    cout << "a+(b+c) = " << setprecision(8) << e << '\n';

    // Intermediate storage and output
    cout << "Additions with intermediate storage AND output of intermediate
results\n";
    d = (a + b);
    cout << "(a+b) =   " << setprecision(8) << d << '\n';
    d += c;
    cout << "(a+b)+c = " << setprecision(8) << d << '\n';
    e = (b+c);
    cout << "(b+c) =   " << setprecision(8) << e << '\n';
    e += a;
    cout << "a+(b+c) = " << setprecision(8) << e << '\n';

    // Intermediate storage but not output
    cout << "Additions with intermediate storage but no output of intermediate
results\n";
    d = (a + b);
    d += c;
    cout << "(a+b)+c = " << setprecision(8) << d << '\n';
    e = (b+c);
    e += a;
    cout << "a+(b+c) = " << setprecision(8) << e << '\n';
    return 0;
}
```

FloatTest.java

```
class FloatTest
{
// not part of the homework assignment; this is included just in case you want
// to try it out. Results will usually vary from C++ but the values given
// demonstrate non-associativity (and incorrectness) of floating point
addition

  public static void main(String args[])
    {
      float a, b, c, d, e;
      a = (float) 0.1;
      b = (float) 5.812;
      c = (float) 78.876;

      // evaluate addition without intermediate storage of results
      System.out.println ("Additions without intermediate storage of
results\n");
      d = (a+b)+c;
      e = a+(b+c);
      System.out.println("(a+b)+c = " + d +"\n");
      System.out.println("a+(b+c) = " + e + "\n");


      // Intermediate storage and output
      System.out.println("Additions with intermediate storage AND output of
intermediate results\n");
      d = (a + b);
      System.out.println("(a+b) = " + d + "\n");
      d += c;
      System.out.println("(a+b)+c = " + d + "\n");
      e = (b+c);
      System.out.println("(b+c) =   " + e + "\n");
      e += a;
      System.out.println("a+(b+c) = " + e + "\n");

      // Intermediate storage but not output
      System.out.println("Additions with intermediate storage but no output of
intermediate results\n");
      d = (a + b);
      d += c;
      System.out.println("(a+b)+c = " + d + "\n");
      e = (b+c);
      e += a;
      System.out.println("a+(b+c) = " + e + "\n");
    }
}
```