1. Introduction

In this assignment, you are required to implement a function in C programming language which adds two floating-point numbers. The goal of this assignment is to understand the binary representation of floating-point numbers and their addition process.

2. Problem specification

2.1 Overview

Write a C function named `fpadd()` which receives two floating-point numbers and returns their sum. The prototype of `fpadd()` is as follows:

```c
unsigned fpadd (unsigned x, unsigned y);
```

Two arguments, `x` and `y`, indicate the floating-point numbers to be added. Note that each parameter is given by `unsigned integer` type, which corresponds to the binary representation of a single-precision floating-point number as defined in the IEEE 754 standard. When you want to perform `1.0 + 0.5`, be careful not to call `fpadd (1.0, 0.5)`. Instead, you should call `fpadd (0x3f800000, 0x3f000000)`. Here, `0x3f800000` and `0x3f000000` are the binary representations of `1.0` and `0.5`, respectively. The return value is also of unsigned integer type, and it should be the binary representation of the sum.

To make the problem simpler, we assume that `x` and `y` are non-negative numbers, i.e., `x ≥ 0` and `y ≥ 0`. If any of the arguments is a negative number, `fpadd()` simply returns NaN.

2.2 Background

2.2.1 Single-precision floating-point representation

A single-precision binary floating-point number is stored in 32 bits, as shown in the above figure. The most significant bit (bit 31) is the sign bit. The next 8 bits (bit 30-23) are the exponent field which is biased by 127. The remaining 23 bits (bit 22-0) represent the significand where the leading “1” is omitted (in case of normalized values). For denormalized numbers, the exponent value is 0. The exponent value of 255 is used to represent special values such as NaN or ∞.

2.2.2 Adding floating-point numbers

The general algorithm for binary floating-point addition is shown below. In step 1, we first align the binary point of the number that has the smaller exponent. In step 2, we add the two significands. Step 3 normalizes the result, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands. Recall that the pattern of all 0 bits in the exponent is reserved and used for the floating-point representation of zero. Moreover, the pattern of all 1 bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers. In step 4, we must round the significand as we have only the fixed number of bits to represent the significand. As a result of this rounding operation, the result may not be in a normalized form anymore. In this case, we go back to step 3, and normalize the result again. The rounding algorithm will be described in more detail in the next subsection.

2.2.3 Rounding algorithms

The IEEE 754-2008 standard defines the following five rounding algorithms.

(a) **Round to nearest, ties to even** – rounds to the nearest value; if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit, which occurs 50% of the time; this is the default algorithm for binary floating-point representation.
The detailed discussion on the rounding algorithms is beyond the scope of this assignment. You just remember that, although the default rounding scheme is “Round to nearest, ties to even,” we use the rounding scheme “Round toward 0” in this assignment. Under this rounding scheme, you can just truncate those bits that cannot fit into the significand field (i.e., 23 bits in the single-precision FP representation).

2.2.4 The simplified flow of floating-point addition for this assignment

In this assignment, the flow of floating-point addition can be simplified in three ways.

First, we assume that operands are non-negative numbers. Hence, the sum will be equal to or greater than any of the operands. This means that we only need to shift right and increment the exponent to normalize the sum.

Second, for the same reason, there will be no underflow.

Finally, if we use the “Round toward 0” scheme, the value will remain normalized as long as the value was normalized before the rounding process. Therefore, we don’t have to normalize the result again.

The following shows the simplified flow of floating-point addition you need to implement for this assignment.

![Flowchart of floating-point addition](image-url)
2.2.4 Example

Assume that we are to add two binary floating-point numbers: \(1.111 \times 2^2 = (7.5_{10})\) and \(1.110 \times 2^{-1} = (0.875_{10})\)

Then, the arguments for \(\text{fpadd}()\) are given as follows:

Step 1: Shift the smaller number (y) to the right until its exponent (-1) matches the larger exponent (2).

Step 2: Add significands

Step 3: Normalize the sum

Step 4: Round the significand: In this case, there’s nothing to truncate since the sum already fits into 23 bits.

Finally, the sum \((1.000011_2 \times 2^3 = 8.375_{10})\) is encoded in single-precision floating-point format and the final number \(0x41060000\) will be the return value of \(\text{fpadd}()\).
### 2.3 Restrictions

(a) When you implement the function \texttt{fpadd()}, you should use only "\texttt{unsigned int}" or "\texttt{int}" type variables and you are allowed to use only integer arithmetic and logical operations inside \texttt{fpadd()}.

(b) Do not use any standard library functions inside \texttt{fpadd()}. The use of \texttt{printf()} is ok for debugging purpose.

(c) Your implementation should work for denormalized values, as well as normalized values.

(d) You should handle some special cases which involves NaN or \(\infty\). Note that adding any value to NaN results in NaN. Likewise, the result of adding any (positive) value other than NaN to \(\infty\) is \(\infty\).

(e) Simply return NaN if one of the arguments is a negative number.

### 3. Verification of your implementation

You can verify your implementation by comparing the result of \texttt{fpadd()} with the value obtained by the real floating-point addition. The following code fragment shows how to do this:

```c
#include <stdio.h>

unsigned fpadd (unsigned x, unsigned y)
{
    // Use (unsigned or signed) integers only here
    // Only integer arithmetic and logical operations allowed
    ...
    return ...;
}

void verify (float x, float y)
{
    float sum = x + y;
    unsigned r = fpadd(*(unsigned *)&x, *(unsigned *)&y);

    printf("%f (0x%08x) + %f (0x%08x)\n",
    x, *(unsigned *)&x, y, *(unsigned *)&y);
}

int main ()
{
    verify (3.14, 0.2003);
}
```
The output of the above code fragment will look like this:

```
3.140000 (0x4048f5c3) + 0.200300 (0x3e4d1b71)
= 3.340300 (0x4055c77a) ; by fpadd()
= 3.340300 (0x4055c77a) ; by FP addition
```

Note that since we are using a different rounding scheme, the result can be slightly different in some cases. For example, the following shows the output when we add 3.14 and 2003.0.

```
3.140000 (0x4048f5c3) + 2003.000000 (0x44fa6000)
= 2006.139893 (0x44fac47a) ; by fpadd()
= 2006.140015 (0x44fac47b) ; by FP addition
```

4. Hand in instructions

- Make sure you have included your name and the student ID in the header comment of your program.
- The source file name which contains the function `fpadd()` should be “YourStudentID.c” (e.g., 2008310123.c).
- “YourStudentID.c” file should contain only the source code of `fpadd()`. **Do not include any other functions such as main().**
- Prepare a separate document in PDF format (most preferred, but other formats, such as .txt, .doc, and .hwp, are also allowed), which explains the design and implementation of your code. The document should be named “YourStudentID.pdf”. The document can be written in Korean if you wish.
- Send a mail to [jinsookim at skku.edu] AND [cse2003skku at gmail.com] with attaching two files, “YourStudentID.c” and “YourStudentID.pdf”. The subject line of the mail should be
  
  **[CSE2003-A] PA#1, YourStudentID, YourName**

  if you belong to the Class 41 (Korean class), or

  **[CSE2003-B] PA#1, YourStudentID, YourName**

  if you belong to the Class 42 (English class).

5. Logistics

- You will work on this assignment alone.
- The submission status will be posted on the course homepage at [http://csl.skku.edu/CSE2003S10](http://csl.skku.edu/CSE2003S10).
- Only the assignments submitted before the deadline will receive the full credit. 25% of the credit will be deducted for every single day delay.
- Any attempt to copy others’ work will result in heavy penalty (for both the copier and the originator). Don’t take a risk.
Good luck!

---

Jin-Soo Kim
Computer Systems Laboratory
School of Information and Communication Engineering
Sungkyunkwan University