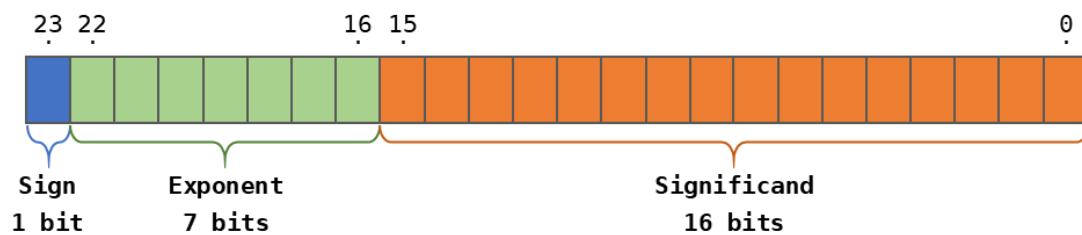


24-bit Small Floating Point

Due: 11:59 PM, 12th April 2020

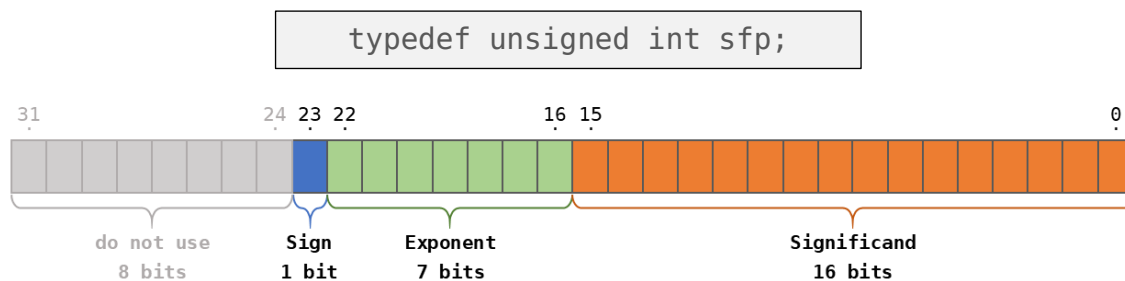
1. Objectives

Design and implement a 24-bit floating point type (**small floating point, or sfp**) and associated APIs: type casting functions from/to **int** and **float**, and **addition/multiplication** operators



2. Overview

The newly designed **small floating point (sfp)** type consists of 1 bit for sign, 7 bits for exponent, 16 bits for significand (fraction/mantissa). In this project, **sfp** is defined as below.



Since "**unsigned int**" is 32bit by default, sfp does not use first 8 bits as shown in the above figure. In order to utilize this new data type, it is mandatory to implement functions supporting type conversions from/to int and float types. You are supposed to program following 4 type-casting operators.

In addition, you are supposed to implement two arithmetic operations, multiplication and addition of sfp.

The below are the prototypes of the type-casting and arithmetic functions.

```
sfp int2sfp(int input);  
int sfp2int(sfp input);  
sfp float2sfp(float input);  
float sfp2float(sfp input);
```

sfp type conversion

```
sfp sfp_add(sfp in1, sfp in2);  
sfp sfp_mul(sfp in1, sfp in2);
```

sfp arithmetic function

3. Details

3.1 int2sfp, float2sfp

- These functions are used in order to convert **int** and **float** data type into **sfp** data type. Return data type is **sfp**.
- For the value which **exceeds** the range of **sfp** (overflow), mark the result as $\pm\infty$. The sign must be ensured clearly. $+\infty$ and $-\infty$ are different from each other.
- Use **round toward zero** as rounding mode.
- For int 0, mark the result as sfp +0.0.

Input (int, float)	Output (sfp)
Int 0	+0.0
Round toward zero mode	

Table 1. Special result values for int2sfp and float2sfp

3.2 sfp2int

- This function is used in order to convert **sfp** data type into **int** data type. Return data type is **int**.
- $+\infty$ and $-\infty$ are represented as **TMax** and **TMin** in **int**. **TMax** and **TMin** are maximum and minimum value of **int** data type, respectively.
- **NaN** is converted into **TMin**.
- Use **round toward zero** as rounding mode

Input (sfp)	Output (int)
$+\infty$	TMax
$-\infty$	TMin
$>TMax$	TMax
$<TMin$	TMin
$\pm NaN$	TMin
Round toward zero mode	

Table 2. Special result values for sfp2int

3.3 sfp2float

- This function is used in order to convert **sfp** data type into **float** data type. Return data type is **float**.
- There is no exception or error cases since **float** type can cover all the value range where **sfp** can express.

3.4 sfp_add

- Two **sfp** data type variables are given as inputs. The result is a **sfp** data type value representing the sum of inputs.
- Before add operation, you should shift right the smaller **sfp** variable. If some bits exceed the range of fraction, apply **round toward even**.
- For the result which **exceeds** the range of **sfp** (overflow), mark the result as $\pm\infty$. The sign must be ensured clearly. $+\infty$ and $-\infty$ are different from each other.
- Use '**round toward even**' rounding mode for final result.
- **Casting sfp to float or double are prohibited in the function.**

In1	In2	Result
$+\infty$	$+\infty$	$+\infty$
$+\infty$	$-\infty$	NaN
$+\infty$	Normal Value	$+\infty$
$-\infty$	$-\infty$	$-\infty$
$-\infty$	Normal Value	$-\infty$
NaN		NaN

Table 3. Special result values for sfp_add

Addition with different exponents

$$\begin{array}{r}
 \text{E 5} \quad 1.111001 \\
 + \\
 \text{E 3} \quad 1.001110 \\
 \hline
 \end{array}$$

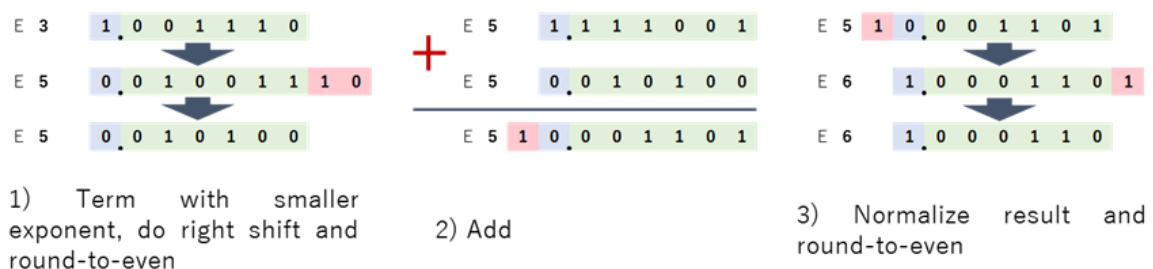


Figure 1. Addition with different exponents - 6 fraction bits example

3.5 sfp_mul

- Two **sfp** data type variables are given as inputs. The result is a **sfp** data type variable representing the product of inputs.
- On calculating fraction part, you can use 64-bit variables. *(for example, [unsigned] long long, double etc).* Then, normalize and round the value (**round toward even**).
- For the result which **exceeds** the range of **sfp** (overflow), mark the result as $\pm\infty$. The sign must be ensured clearly. $+\infty$ and $-\infty$ are different from each other.
- Use **round toward even** as rounding mode.
- **Casting sfp to float or double are prohibited in the function.**

In1	In2	Result
$+\infty$	$+\infty$	$+\infty$
$+\infty$	$-\infty$	$-\infty$
$+\infty$	Positive Normal Value	$+\infty$
$+\infty$	Negative Normal Value	$-\infty$
$-\infty$	$-\infty$	$+\infty$
$-\infty$	Positive Normal Value	$-\infty$
$-\infty$	Negative Normal Value	$+\infty$
$\pm\infty$	0	NaN
NaN		NaN

Table 4. Special result values for sfp_mul

Multiplication

\times

 E 5 1.111001
 E 3 1.001111

E 8 10.010101 | 010111

E 8 10.010101 | 010111

E 9 1.001010 1010111

E 9 1.001011

1) Multiply two sfp inputs

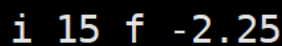
2) Normalize result and round toward even within fraction bits

Figure2. Multiplication - 6 fraction bits example

4. Given files

The given files are **hw1.c**, **sfp.c**, **sfp.h**, **Makefile**, and three **input** and **output examples**. You only need to modify **sfp.c** file. **answer1.txt**, **answer2.txt**, and **answer3.txt** are answers of **input1.txt**, **input2.txt**, and **input3.txt**, respectively.

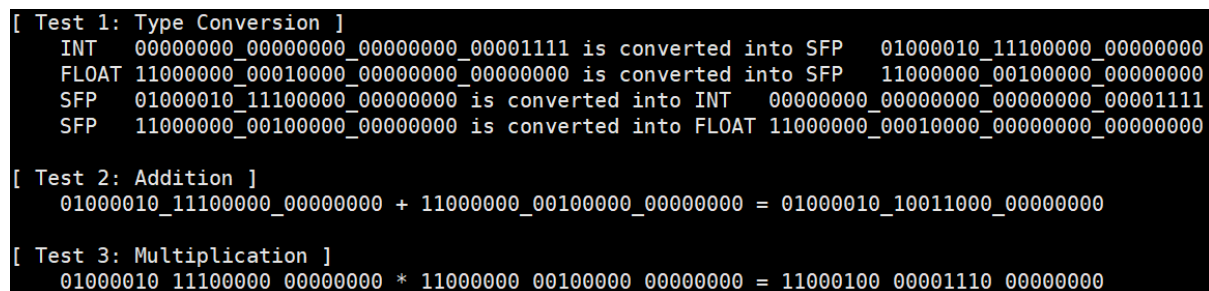
Input file must have four parameters. A pair of two parameters indicates the **type** and **value** of the input respectively. For example, **i 15** means **int 15** and **f -2.25** means **float -2.25**.



The image shows a black rectangular box with the text "i 15 f -2.25" in white, representing the input file format.

Figure 1. Input file format

Two inputs should be delivered from **input.txt**, and each input is converted into **sfp** data type. The results are shown in the bit stream form. These values are used as addition and multiplication tests again, and the results of calculation are printed out in the bit stream form as well.



The image shows a black rectangular box with white text representing the output file format. It contains three test results: Test 1 (Type Conversion) showing INT, FLOAT, and SFP conversions to bit streams; Test 2 (Addition) showing the addition of two SFP values; and Test 3 (Multiplication) showing the multiplication of two SFP values. All results are displayed in bit stream format.

Figure 2. Output file format

5. Execution

- After implementing all functions, type **"make"** on your terminal.
- You can execute the program with input file by typing **"./hw1 < input.txt"**.
- If you want to create a output file, type **"./hw1 < input.txt > output.txt"**.

6. Submission

- **Comments for explaining your code should be contained in your sfp.c file.**
- You should submit a report on PDF file that contains a description of your codes. The name of the PDF file should be your student id **"StudentID.pdf"** (e.g. 2019012345.pdf)
- For submission, you should compress your files(sfp.c sfp.h). You can create a compressed

file named "**StudentID.tar.gz**" by typing "**make tar STUDENT_ID=your_student_id**". (e.g. "make tar STUDENT_ID=2019012345" command create "2019012345.tar.gz" file)

```
~/hw1$ make tar STUDENT_ID=2019012345
tar -cvzf 2019012345.tar.gz sfp.c sfp.h
sfp.c
sfp.h
~/hw1$ ls
2019012345.tar.gz hw1.c Makefile sfp.c sfp.h
```

- Submit **both pdf file and the tar file** on *canvas*.
- There is a delay penalty of 10% per day.
- **Plagiarism will result in a grade of zero. Never copy other persons's work.**

7. Helpful things

3.1 Bitwise AND, OR Operators

The bitwise AND operator is a single ampersand (&). Bitwise binary AND perform a logical AND of the bits at each position of a number in binary format. Like bitwise AND, bitwise OR only works at the bit level. The symbol is (|) and can be called a pipe. See the following example.

```
#include <stdio.h>
int main(void) {
    unsigned int a = 21;        /* a = 10101 in binary */
    unsigned int b = 7;         /* b =   111 in binary */
    unsigned int c = a & b;     /* c =   101 in binary */
    unsigned int d = a | b;     /* d = 10111 in binary */
    printf("%u, %u\n", c, d);   /* print 5, 23 */
}
```

3.2 Shift operators

There are two bitwise shift operators. They are right shift and left shift. The symbol of right shift operator is ">>". Two operands are needed for this operation. It shifts each bit in its left operand to the right. The number that follows the operator determines the number of bits should be shifted. Thus, by doing "**i >> 3**" all the bits will be shifted to the right by three times and so on.

The symbol of left shift operator is "<<". It shifts each bit in its left operand to the left by the number of positions the right-hand operand points to. It works opposite direction to the right shift

operator. See the following example.

```
#include <stdio.h>
int main(void) {
    unsigned int a = 29;      /* a = 11101 in binary */
    unsigned int b = a >> 2; /* b = 111 in binary */
    unsigned int c = a << 1; /* c = 111010 in binary */
    printf("%u, %u\n", b, c); /* print 7, 58 */
}
```

3.3 memcpy

```
memcpy (void* destination, const void* source, size_t num);
```

This function copies the data size of 'num' located at 'source' to 'destination'. Since bit operations cannot be done on float data types, copying the float data into another type of data (e.g, unsigned int) might help you perform data type conversions. See the following example.

```
#include <stdio.h>
int main(void) {
    float f = 31;
    unsigned int ui;
    memcpy((void*)&ui, (void*)&f, sizeof(float));
    printf("%f, %u\n", f, ui);
}
```

3.4 diff command

The **diff** command analyzes and compares two files, then prints out the different lines. You can use this command for comparing output files with the answer files. See the following example.

```
$ cat output1.txt
Result1 = 11
Result2 = 22
Result3 = 33
$ cat answer1.txt
Result1 = 11
Result2 = 222
Result3 = 333
$ diff output1.txt answer1.txt
2,3c2,3
< Result2 = 22
< Result3 = 33
- - -
> Result2 = 222
> Result3 = 333
```