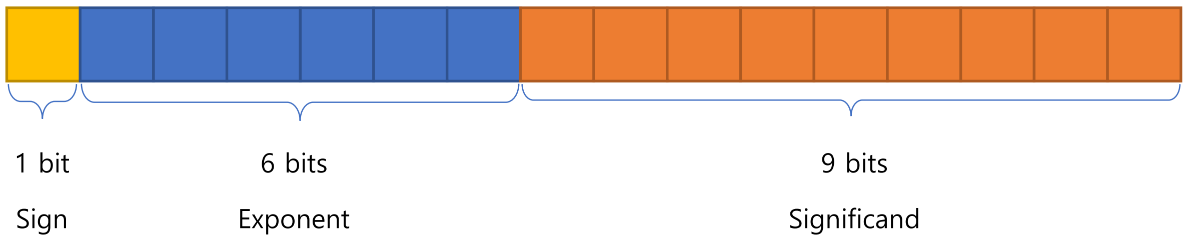
16-bit Small Floating Point

1. Objectives

Design and implement 16-bit floating point type **(small floating point, or sfp for short)**, type cast operations between **int** and **float**, and its associated **addition** and **multiplication** operations as well.



2. Overview

The newly designed **small floating point(sfp)** type consists of 1 bit for sign, 6 bits for exponent, and the rest for significand. In this project, **sfp** is represented as below.

```
typedef unsigned short sfp;
```

In order to utilize this newly implemented data type, it is mandatory to design functions supporting conversion against conventional data types. In our assignment, we aim **int** and **float** as targets. You are supposed to program following 4 kinds of type-cast operators.

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

Two arithmetic functions should be implemented associated with **sfp** data type. Return data type is **sfp** as well.

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

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 ±∞**.** The sign must be ensured clearly. +∞ and -∞ are different from each other.
- Use **round-to-zero** as rounding mode.
- For int 0, mark the result as sfp +0.0.

Input (int, float)	Output (sfp)
Int 0	+0.0
Round-to-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**.
- +∞ and -∞ 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-to-zero** as rounding mode.

Input (sfp)	Output (int)
+∞	TMax
-∞	TMin
>TMax	TMax
<TMin	TMin
±NaN	TMin
Round-to-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.
- For the result which **exceeds** the range of **sfp** (overflow), mark the result as ±∞**.** The sign must be ensured clearly. +∞ and -∞ are different from each other.
- Use **round-to-even** rounding mode.
- **Casting sfp to float or double are prohibited in the function.**

In1	In2	Result
+∞	+∞	+∞
+∞	-∞	NaN
+∞	Normal Value	+∞
-∞	-∞	-∞
-∞	Normal Value	-∞
NaN		NaN

Table 3. Special result values for sfp_add

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.
- For the result which **exceeds** the range of **sfp** (overflow), mark the result as ±∞**.** The sign must be ensured clearly. +∞ and -∞ are different from each other.
- Use **round-to-even** as rounding mode.
- **Casting sfp to float or double are prohibited in the function.**

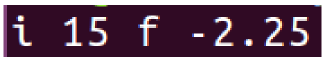
In1	In2	Result
+∞	+∞	+∞
+∞	-∞	-∞
+∞	Positive Normal Value	+∞
+∞	Negative Normal Value	-∞
-∞	-∞	+∞
-∞	Positive Normal Value	-∞
-∞	Negative Normal Value	+∞
±∞	0	NaN
NaN		NaN

Table 4. Special result values for sfp_mul

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.

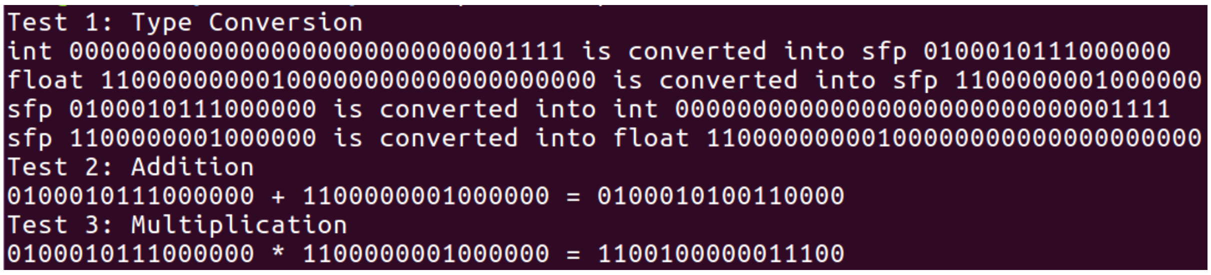
Input file must have four parameters. A pair of two parameters indicates the **type** and **value** of the input respectively. For example, **i 15** means **int 15** and **f -2.25** means **float -2.25.**



```
i 15 f -2.25
```

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.



```
Test 1: Type Conversion
int 00000000000000000000000000001111 is converted into sfp 0100010111000000
float 11000000000100000000000000000000 is converted into sfp 1100000001000000
sfp 0100010111000000 is converted into int 00000000000000000000000000001111
sfp 1100000001000000 is converted into float 11000000000100000000000000000000
Test 2: Addition
0100010111000000 + 1100000001000000 = 0100010100110000
Test 3: Multiplication
0100010111000000 * 1100000001000000 = 1100100000011100
```

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 create a PDF file that contains a description of your sfp.c file. The PDF file name must have a form of "**StudentID.pdf**" (e.g. 2018012345.pdf)
- You should create a compressed file that contains your code. 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=2018012345" command create "2018012345.tar.gz" file)
- You must submit **both the pdf file and the tar file** by email.
- There is a penalty of 10% per day when you miss the deadline.
- **<u>If plagiarism is detected, 0 point for your score. Never copy other's work.</u>**

7. Helpful things

3.1 Bitwise AND, OR Operators

The bitwise AND operator is a single ampersand (**&**). Bitwise binary AND does the logical AND of the bits in each position of a number in its binary form. Like bitwise AND, bitwise OR only operates at the bit level. Its symbol is (**|**) which can be called 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 ">>". For its operation, it requires two operands. It shifts each bit in its left operand to the right. The number following the operator decides the number of places the bits are shifted. Thus, by doing "**i >> 3**" all the bits will be shifted to the right by three places and so on.

The symbol of left shift operator is "<<". It shifts each bit in its left-hand operand to the left by the number of positions indicated by the right-hand operand. It works opposite to that of 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 whose size is 'num', located at 'source' to 'destination'. Since float data type does not accept bit operations, copying the float data into another data type accepting bit operations (for example, unsigned int) might help you conduct conversions as you have learned. See the following example.

```
#include <stdio.h>
int main(void) {
      float f = 31;
      unsigned int ui;
      memcpy((void*)(&ui), (void*)(&f), 4);
      printf("%f, %u\n", f, ui);
}
```

3.4 diff command

diff analyzes two files and prints the lines that are different. Essentially, it outputs a set of instructions for how to change one file to make it identical to the second file. You can use this command to compare your output file to the answer file. 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
```