System program - Assignment 1

HPFP

The purpose of this assignment is to become familiar with data representation of computer systems, especially about floating point numbers. You will design and implement **16-bit floating** point type, so called here *half-precision* floating point or **hpfp** that is compatible with most of IEEE standard 754 behaviors (explained in the class material in 3rd week – Data representation floating point number); recall that you've learned about 32 bits (*single* precision) and 64 bits (*double* precision) standards.

In this assignment, you will implement the type cast and the addition operation of **hpfp** that are defined in the below specification section. It is not permitted to use any library that is specifically designed for such different floating point types.

Float to floating point converter: https://evanw.github.io/float-toy/ (Use this link to check your answer)

The format of **hpfp** contains

- bit sign (s),
- 5-bit exponent (exp),
- 10-bit significand (frac).

The format of **hpfp** is laid out as follows.

1. Specification of hpfp :

In C, hpfp is represented as below.

```
typedef unsigned short hpfp;
```

In order to utilize such a newly implemented data type, it is required to design and implement functions supporting conversion with several conventional data types. To do so, you will implement the following four type-cast functions (type conversion with **int** and **float**). In addition, you will also implement arithmetic operations and flip operation.

```
/* convert int into hpfp */
hpfp int_converter(int input);
/* convert hpfp into int */
int hpfp_to_int_converter(hpfp input);
/* convert float into hpfp */
hpfp float_converter(float input);
/* convert hpfp into float */
float hpfp_to_float_converter(hpfp input);

hpfp addition_function(hpfp a, hpfp b);
hpfp multiply_function(hpfp a, hpfp b);
char* comparison_function(hpfp a, hpfp b);
char* hpfp_flipper(char* input);
```

You will also implement the following bit stream function so that you will be able to return the bit stream of **hpfp** data for the result evaluation.

```
char* hpfp_to_bits_converter(hpfp result);
```

2. Detail specification of hpfp :

hpfp int_converter(int input), hpfp float_converter(float input)

- These functions are used to convert **int** data type, and float data type into **hpfp** data type, respectively. The return data type is **hpfp**.
- For the value which exceeds the range of **hpfp** (overflow), mark the result as ±∞. (The sign must be ensured clearly. Note that +∞ and -∞ are different.)
- Use round-to-zero as rounding mode.
- For **int** 0, mark the result as **hpfp** +0.0

Input (int, float)	Output (hpfp)
>maximum of hpfp	+∞
<minimum of hpfp	-∞
int 0	+0.0
Round-to-zero mode	

Special result values for int_converter and float_converter

int hpfp_to_int_converter (hpfp input)

- This function is used to convert **hpfp** data type into **int** data type. The return data type is **int**.
- +∞ and -∞ is represented as TMax and TMin of **int**, respectively.
- NaN is converted into TMin.
- Use round-toward-zero as rounding mode.

Input (hpfp)	Output (int)
+∞	TMax
-∞	TMin
±NaN	TMin
Round-to-zero mode	

Special result values for hpfp_to_int_converter

float hpfp_to_float_converter(hpfp input)

- This function is used to convert **hpfp** data type into **float** data type. The return data type is **float**.
- Note that there is no exception or error cases since **float** type is capable of covering all the value range of **hpfp**.

hpfp addition/_function(hpfp a, hpfp b)

- Two **hpfp** variables are given as inputs. The result is a **hpfp** data type value representing the sum of the inputs.
- For the result which **exceeds** the range of **hpfp** (overflow), mark the result as infinity. (the sign must be ensured clearly)
- Use round-to-even rounding mode.
- Casting **hpfp** to **float** or **double** for this addition is prohibited in the function. Manipulating the bits of the two **hpfp** variables is required.

in1	in2	result
+∞	+∞	+∞
+∞	-∞	NaN
+∞	Normal Value	+∞
-∞	-∞	-∞
-∞	Normal Value	-∞
NaN	Any Value	NaN

Special result values for addition_function

hpfp multiply_function(hpfp a, hpfp b)

- Two **hpfp** variables are given as inputs. The result is a **hpfp** data type value representing the multiplication of the inputs.
- For the result which **exceeds** the range of **hpfp** (overflow), mark the result as infinity. (the sign must be ensured clearly)
- Use round-to-even rounding mode.
- Casting **hpfp** to **float** or **double** for this multiplication is prohibited in the function. Manipulating the bits of the two **hpfp** variables is required.

in1	in2	result
+∞	+∞	+∞
+∞	-∞	-∞
-∞	-∞	+∞
+∞	Normal Value	±∞
-∞	Normal Value	±∞
±∞	0	NaN
NaN	Any Value	NaN

Special result values for multiply_function

char* comparison_function(hpfp in1, hpfp in2)

- Two **hpfp** variables are given as inputs. The result is a **string** type value representing comparison of the inputs.
- If in1 is greater than in2, return > and if in1 is less than in2, return <, otherwise return =.
- Casting **hpfp** to **float** or **double** for this comparison is prohibited in the function. Manipulating the bits of the two **hpfp** variables is required.

in1	in2	result
+∞	+∞	=
+∞	-∞	>
-∞	-∞	=
+∞	Normal Value	>
-∞	Normal Value	<
NaN	Any Value	=

Special result values for comparison_function

char* hpfp_to_bits_converter(hpfp result)

- This function is used to return the bit stream of **hpfp** data type. (e.g. if **hpfp** val means 15 in decimal, char *string=hpfp2bits(val); has “0100101110000000”)
- Use malloc() in the function for returning a string. The returned string can be freed by the caller of hpfp_to_bits_converter().

char* hpfp_flipper(char* input)

- This function change **hpfp** input to **float (or int)**, then **reverse** the **float (or int)**, and turn **float (or int)** back into **hpfp**.
- When flipping a float, the decimal point remains in the same position as before the flip. That is, since “52.75” has its integer part up to the second integer, “5725” becomes “57.25”.
- Example: hpfp(0101001010011000) -> float(52.75) -> flaot(57.25) -> hpfp(0101001100101000)
- Example: hpfp(0101000011100000) -> int(39) -> int(93) -> hpfp(0101010111010000)

3. Example :

Each line of an input file contains only a number. The first line denotes the number of inputs as **int**. Given the number of inputs, the following lines include **int** type inputs. Same format of input lines is followed for the case of **float** type inputs and **hpfp** inputs.

Input file : input.txt

```
2
49
27
2
-19.1875
85.3125
2
0101001010011000
0101000011100000
```

Execution result

```
Test 1: casting from int to hpfp
int(49) => hpfp(0101001000100000), CORRECT
int(27) => hpfp(0100111011000000), CORRECT

Test 2: casting from hpfp to int
hpfp(0101001000100000) => int(49), CORRECT
hpfp(0100111011000000) => int(27), CORRECT

Test 3: casting from float to hpfp
float(-19.1875) => hpfp(1100110011001100), CORRECT
float(85.3125) => hpfp(0101010101010101), CORRECT

Test 4: casting from hpfp to float
hpfp(1100110011001100) => float(-19.1875), CORRECT
hpfp(0101010101010101) => float(85.3125), CORRECT

Test 5: Addition
0101001000100000 + 0101001000100000 = 0101011000100000, CORRECT
0101001000100000 + 0100111011000000 = 0101010011000000, CORRECT
0100111011000000 + 0100111011000000 = 0101001011000000, CORRECT
1100110011001100 + 1100110011001100 = 1101000011001100, CORRECT
1100110011001100 + 0101010101010101 = 0101010000100010, CORRECT
0101010101010101 + 0101010101010101 = 0101100101010101, CORRECT
0101001000100000 + 1100110011001100 = 0100111101110100, CORRECT
0101001000100000 + 0101010101010101 = 0101100000110010, CORRECT
0100111011000000 + 1100110011001100 = 0100011111010000, CORRECT
0100111011000000 + 0101010101010101 = 0101011100000101, CORRECT

Test 6: Multiplication
0101001000100000 * 0101001000100000 = 0110100010110000, CORRECT
0101001000100000 * 0100111011000000 = 0110010100101011, CORRECT
0100111011000000 * 0100111011000000 = 0110000110110010, CORRECT
1100110011001100 * 1100110011001100 = 0101110111000001, CORRECT
1100110011001100 * 0101010101010101 = 1110011001100101, CORRECT
0101010101010101 * 0101010101010101 = 0110111100011100, CORRECT
0101001000100000 * 1100110011001100 = 1110001101011000, CORRECT
0101001000100000 * 0101010101010101 = 0110110000010101, CORRECT
0100111011000000 * 1100110011001100 = 1110000000001100, CORRECT
0100111011000000 * 0101010101010101 = 0110100010000000, CORRECT

Test 7: Comparison
0101001000100000 = 0101001000100000, CORRECT
0101001000100000 > 0100111011000000, CORRECT
0100111011000000 = 0100111011000000, CORRECT
1100110011001100 = 1100110011001100, CORRECT
1100110011001100 < 0101010101010101, CORRECT
0101010101010101 = 0101010101010101, CORRECT
0101001000100000 > 1100110011001100, CORRECT
0101001000100000 < 0101010101010101, CORRECT
0100111011000000 > 1100110011001100, CORRECT
0100111011000000 < 0101010101010101, CORRECT

Test 8: hpfp -> float -> flipped float -> hpfp
hpfp(0101001010011000) => hpfp(0101001100101000)
hpfp(0101000011100000) => hpfp(0101010111010000)
```

4. Note

- Skeleton code is given in **hpfp.c**. Input cases and scoring system will be implemented by TA.
- **Include a pdf file (or doc file) that explains your design and code in your hpfp.c file.** The file name is studentid.pdf.
- After implementing all functions, type “make” on your terminal, and execute “hw1”.
- Compress all the files relevant to this assignment such as **hpfp.c, hpfp.h, hw1.c, Makefile, studentid.pdf**. The Compressed file name must have a form of “**studentid.tar**” (Use “tar” compression command on Ubuntu. e.g. tar -cvf 2017719486.tar SP_HW1_2024s).
- Submit your assignment by uploading the “**studentid.tar**” file to icampus.
- If **plagiarism** is detected, 0 point will be given and additional penalty will be considered.