Homework 2

Computer Architecture I @ ShanghaiTech University

Overview

In this homework, you are required to implement parsing of a modified floating-point format `ca25` and conversion from the format to IEEE 754 single-precision floating-point number (`fp32`). In this homework, you will

- be able to perform basic I/O using C language;
- know integer types, their sizes and ranges;
- understand bitwise operations and learn how to use them;
- understand the principles behind IEEE 754 floating-point numbers.

Before you start, you have to join the GitHub Classroom and accept the assignment via this link. A repository containing the starter code will be generated to you. You can then start on the assignment by cloning it to your Ubuntu system. To submit the assignment, you should push your local clone to GitHub and turn in your work on Gradescope by connecting to your GitHub repo.

Academic integrity is strictly enforced in this course and any plagiarism behavior *will* cause serious consequence. **You have been warned**.

Specification

The `ca25` numbers to be converted are stored in text files line by line and passed to your program by `Makefile`. Your program should directly read from `stdin` and print to `stdout`. Then the printed content is automatically saved into another text file by the test script. All input and output are ASCII text.

The C program should keep performing the following tasks until `EOF` is met:

1. Read a `ca25` floating-point number which is expressed in 16-digit hex.
2. Convert the loaded `ca25` number to single-precision `fp32`.

To examine the correctness of your work, for each `ca25` number, the program is required to produce three lines of output:

1. Print the sign, exponent and significand of the original `ca25`.
2. Print the sign, exponent and significand of the converted `fp32`.
3. Print the hex representation of the converted `fp32`.

Floating-Point Format `ca25`

The modified number format is analogous to IEEE 754, but with different field lengths and organization. It has one sign bit, a 32-bit exponent field and a 31-bit mantissa field.

```
 MSB                             LSB
+---+---------------+--------------+
| 63 | 62   ...   31 | 30   ...   0 |
+---+---------------+--------------+
 Sign     Exponent        Mantissa
```

- The most significant bit stores the sign of the number. Bit one stands for negative.
- The exponent field is a 32-bit unsigned integer in biased form: a value of `2^31-1` represents the actual exponent zero.
- For normal numbers, the mantissa includes 31 fraction bits to the right of the binary point and *an implicit leading bit one* to the left of the binary point. For subnormal numbers and zeros, the implicit leading bit is of value zero.

Exponent	**Mantissa** = **0**	**Mantissa** ≠ **0**
`0x00000000`	±zero	subnormal
`0xFFFFFFFF`	±infinity	NaN
otherwise	normal	normal

An Example Run

This example is provided to aid you in better understanding the input and output specification. For the following input

```
40000038da100000
3fffffb39cde0b01
```

the program must print the following text output

```
ca25 S=0 E=80000071 M=1.b4200000 normal
fp32 S=0 E=f1 M=1.b42000 normal
78da1000
ca25 S=0 E=7fffff67 M=1.39bc1602 normal
fp32 S=0 E=00 M=0.000002 subnormal
00000001
```

Input

Each line of the input consists of one `ca25` floating-point number in 16-digit lower-case hex. The number of lines is *unspecified*: the program keeps processing input until it reads `EOF`.

Output1: parsed `ca25`

The format is specified as follows with some examples.

```
ca25 S=<sign> E=<exponent> M=<implicit>.<fraction> <type>

  <sign>   = sign bit
<exponent> = biased exponent in 8-digit hex
<implicit> = implicit leading bit, 0 for zero or subnormal and 1 for others
<fraction> = fraction in left-adjusted 8-digit hex
             (so the LSB should always be zero)
  <type>   = "normal" | "subnormal" | "zero" | "nan" | "inf"

ca25 S=0 E=7fffffff M=1.80000000 normal    # 3fffffffc0000000 = 1.5
ca25 S=1 E=00000000 M=0.40000000 subnormal # 8000000020000000 = -2^(2^-34)
ca25 S=1 E=00000000 M=0.00000000 zero      # 8000000000000000 = -0
ca25 S=0 E=ffffffff M=1.fffffffe nan       # 7fffffffffffffff = nan
ca25 S=0 E=ffffffff M=1.00000000 inf       # 7fffffff80000000 = +inf
```

Recall that one hex digit can represent four binary bits. As `ca25` has 31 explicit bits of significand, when expressed in hex digits, there is one bit redundant, namely the least significant. That bit should always be treated as zero. As a result, the LSB of `<fraction>` should always be zero.

Rounding Away From Zero

The intermediate values during a floating-point arithmetic operation are generally of higher precision than the destination, hence the result has to be rounded before storing into the destination to fit into its precision. When converting a floating-point number from a higher-precision format into a lower-precision format, the rounding mode determines how the value should be adjusted.

The rounding mode in this homework is *rounding away from zero*. An informal but intuitive example is that, when converting real numbers to integers with this mode, `-4.5` would be rounded to `-5` and `4.5` would be rounded to `5`. Formally, this mode rounds a number to the nearest *representable* value whose *magnitude (absolute value)* is no lesser than the original number.

Hint: For each type of `ca25` number, consider whether this rounding mode will affect its type and how.

Conversion to `fp32`

Recall that a `fp32` has one bit for sign, 8-bit for 127-biased exponent and 23-bit explicit significand bits. The conversion has several rules:

- A number is rounded away from zero, i.e., it rounds to the nearest representable value whose magnitude is no lesser.
- If the number has magnitude greater than `fp32` could represent, it becomes infinity.
- If the number is nan, the upper 23 bits of its mantissa field should be preserved and the lower 8 bits are *truncated without rounding*. (This is because there are

nans making use of the mantissa field to carry information.)
- Regardless of the original number type, the sign must be preserved.

As `fp32`'s exponent and mantissa fields are both shorter than the `ca25` format, careful you should be for subtle things when doing the rounding, including, but not limited to the following points:

- Due to reduced exponent range, normal `ca25` may become subnormal `fp32`, resulting in more bits to be considered in rounding.
- Rounding may consequently cause adjustment of the exponent field.
- Rounding may cause a subnormal result to become a normal value.

Output2: parsed `fp32`

The format is specified as follows with some examples.

```
fp32 S=<sign> E=<exponent> M=<implicit>.<fraction> <type>

  <sign>   = sign bit
<exponent> = biased exponent in 2-digit hex
<implicit> = implicit leading bit, 0 for zero or subnormal and 1 for others
<fraction> = fraction in left-adjusted 6-digit hex
             (so the LSB should always be zero)
  <type>   = "normal" | "subnormal" | "zero" | "nan" | "inf"

fp32 S=1 E=01 M=1.54e44a normal    # 80aa7225
fp32 S=0 E=00 M=0.000002 subnormal # 00000001
fp32 S=1 E=00 M=0.000000 zero      # 80000000
fp32 S=0 E=ff M=1.20219a nan       # 7f9010cd
fp32 S=1 E=ff M=1.000000 inf       # ff800000
```

Recall that one hex digit can represent four binary bits. As `fp32` has 23 explicit bits of significand, when expressed in hex digits, there is one bit redundant, namely the least significant. That bit should always be treated as zero. As a result, the LSB of `<fraction>` should always be zero.

Output3: converted `fp32` in hex

No matter what type the converted result `fp32` is, output the 8-digit lower-case hex representation.

Tips

- Carefully choose integer types in your program to make sure integer wrap-around or overflow will not surprise you.
- Unions in C language come handy for outputting hex representation, but it is not the only way.
- Take into account all the possible cases to earn a full mark.
- To know more details of `EOF` handling, you might like to run `man scanf` in your terminal. For output formatting, `man printf` offers useful knowledge. For those who

prefer reading on a modern interface or on a Windows system, google should lead you to similar online docs.

Build and Test

Makefile, testcases and tester script are provided in the starter code, with which you can compile and test your code.

Compile

In your assignment directory, run the following command.

```
make convert
```

This will effectively compile your code using the following command.

```
gcc -std=c11 -Wall -Wextra -Wpedantic -Werror -O2 -ggdb convert.c -o convert
```

The command enables C11 standard and many warning options. Most importantly, all warnings are treated as error and will cause the compilation to fail. This decision is made to help you avoid some tricky bugs, such as unintentionally comparing signed and unsigned integers.

Test

In your assignment directory, run the following command.

```
make test
```

The command will run your program with the testcases and compare your output against the correct solution. It will output the number of lines which matches the expected output. In addition, it will also create `testcases/total-output.mark` to indicate which output lines are incorrect. An example is as follows.

```
[ ] ca25 S=0 E=7ffffffc M=1.da21a280 normal
[x] fp32 S=0 E=7c M=1.da21a6 normal
[x] 3e6d10d3
[ ] ca25 S=0 E=7ffffffc M=1.9dc84602 normal
[x] fp32 S=0 E=7c M=1.9dc84c normal
[x] 3e4ee426
```

The mark `[x]` means that line is incorrect.

Submission

To make a submission, you will take the following steps:

1. Run `git add convert.c` in the assignment directory to stage your changes.
2. Run `git commit` and write appropriate commit message to commit your changes.
3. Run `git push` to push commits in your local clone to GitHub.
4. Submit on Gradescope by selecting your GitHub repository and the right branch.

Only the *last active submission* will be accounted for your homework grade. Make sure it is your best version.

The following TA(s) are responsible for this homework:

Yizhou Wang <wangyzh2024 AT shanghaitech.edu.cn>