Project 1

In this project you will combine most of what you have learned in the first few weeks of class — C and assembly code, the Linux command line, and binary number manipulation — as you build a program to manipulate floating-point numbers.

Intellectual integrity

This project is to be done individually. You may discuss it in general terms with other students, but may not share code or answers.

You may use an AI tool for clarifying the language of the assignment, for example, translating it into your native language or summarizing what you are being asked to do. You may use an AI tool to help understand the concepts surrounding an assignment. **Any other uses should be cleared with the instructor before using the tool** (ask over Slack), and in any case you must indicate in your answer or in a comment in the code that you got help from the AI and how it helped you.

Project parameters

Background

Floating-point numbers are stored on the computer in a vastly different way from integers. Rather than a straightforward binary representation, floats are stored in what boils down to base-2 scientific notation:

$$f = \pm\ 1.\mathit{(fraction)} \times 2^{\mathit{(exponent)}}$$

For example, if the number is positive, the fraction is 101, and the exponent is 5, the value of the number is

$$1.101 \times 2^5 = 110100 = 52.$$

A floating-point number like a `float` or `double` stores its sign (+ or –), fraction, and exponent together. By far the most common formats are the *IEEE 754 single-precision floating-point number* (also known as “float” in many languages) and the corresponding *double-precision floating-point number* (also known as “double”). Desktop-class CPUs included specialized hardware for decoding these numbers and performing calculations with them.

For purposes of this project, we will work with `double` values (double-precision floats). Wikipedia has a nice article on this format; for our purposes the important information is shown in the diagram:

- the sign (0 for +, 1 for –) is stored in the *top bit* (bit 63), the one furthest from the decimal point;

- the exponent is stored in the subsequent *11 bits* (bits 52–62); and
- the fraction is stored in the remaining *52 bits* (bits 0–51).

(`float`s are similar but use fewer bits to fit into a 32-bit field.)

Roadmap

The purpose of this project is to pull apart the bits of a `double`, manipulate them, and then reassemble them into a new `double` value. The C bit-manipulation operators (`&`, `|`, `<<`, etc.) work only on integer types, so we will write some assembly code to get the bits of a `double` as a `long` and vice-versa. (We use `long`s because, on the server, `double`s and `long`s are both the same size: 64 bits.) Specifically, the steps of this project are as follows:

Step 1
: Write assembly code to access the bits of a `double` value as a `long`. (We do this in assembly because it is not an operation permitted in C.)

Step 2
: Write C functions to extract the three fields (sign, exponent, and fraction) from a `long`.

Step 3
: Write a function to do the reverse: assemble a `long` from a sign, exponent, and fraction.

Step 4
: Write assembly code to take the bits of a `long` and access them as a `double`.

Step 5
: Pull it all together: write a C program that manipulates `double` values by altering the sign, exponent, and fraction fields.

There are tests you can run against your code at each step to make sure that your code is producing the correct results.

If you are not able to complete a step

This project is structured such that, except for the final step, you can complete each of the steps individually. If you get stuck on Step 1, then ask questions, but while waiting for assistance you can work on Step 2. In fact, if you wish you could do Steps 1–4 in any order, although the order they are in builds up to the full program naturally.

Grading

Grading of this project is described in the Canvas assignment for this project.

Step 1

Goal Write assembly code to access the bits of a `double` value as a `long`.

On the server, make a directory (perhaps `project1`) to hold your files for this project. Use `scp` to upload `project.c` and `double_long.c` into your directory.

The first step of the project is to write assembly code to interpret the bits of a `double` as a `long`. (This is different from just casting a `double` to `long`, as we need to keep the sign, exponent, and fraction bits intact; a cast such as `(long)3.14159265` would result in the `long` bits `0000...0011` (that is, 3), destroying the sign, exponent, and fraction.) We will do this in assembly because the C language does not give us a way to do it directly.

a. On the server, open `double_long.c`; this file defines two functions, `double_to_long_bits()` and `long_bits_to_double()`, that reinterpret the bits of a `double` as a `long` and vice-versa. This is separate from the main project file (`project.c`) because you will be manipulating the assembly code generated from this file. If they were together, you would need to redo your changes every time you compiled the project!

   Use `gcc -S` to compile the file, stopping after generating assembly code:

   ```
   gcc -S double_long.c
   ```

   This creates `double_long.s` with the assembly code for the two functions.

b. Open `double_long.s` and find the definition of the `double_to_long_bits()` function. The current assembly code loads the value of the parameter (`double x`) from memory (at an address involving `%rbp`) into a floating-point register `%xmm0`, uses the `cvttsd2siq` instruction to cast it to an integer in `%rax`, and then cleans up and leaves. (Recall that the return value of a function is normally kept in `%rax` for 64-bit values.)

   To get the desired result — returning the bits of the `double x` as a `long` — we simply need to skip the cast, copying the value of `x` directly into `%rax` where it will be returned.

   Modify the assembly code to make this happen. (You will need to delete the `cvttsd2si` line and, in its place, use `movq` to copy the value of the parameter into `%rax`.) Save your changes when done.

c. Now we will test your `double_to_long_bits()` function. Open `project.c`; you can see that there is a prototype for the function (with `extern`, meaning that the function is external — coming from another file). In your `main()`, write code to call `double_to_long_bits()` and print out the return value.

d. To compile the complete program (`project.c` and your modified `double_long.s` together), just name both source files in the `gcc` command:

   ```
   gcc project.c double_long.s -o project
   ```

   (Make sure you say `double_long.`**`s`**, not .c, so that it uses your modified assembly code!) You can then run `./project` to test your code.

Here are a few `double` values to check, with the `long`s that you should get from `double_to_long_bits()`:

`double` value	`long` result
0.0	0
5.0	4617315517961601024
-1.0	-4616189618054758400
3.14	4614253070214989087

e. If your code works, then I **strongly** recommend that you make a backup copy of your `double_long.s` file, then move on to Step 2. If it's not working, you can fix up your `double_long.s` file and try again, or start over by re-running the `gcc -S` command.

Step 2

Goal Write C functions to extract the three fields (sign, exponent, and fraction) from a `long`.

Now you will fill in the `sign_of()`, `exponent_of()`, and `fraction_of()` functions to extract the sign, exponent, and fraction fields from the bits of a `long`. To do this, you will use C's bit operators (such as `&` and `>>`).

You may want to review §4.6 (Bitwise Operators) of Dive Into Systems for an explanation of the `&`, `|`, `<<`, and `>>` operators.

a. Start with `fraction_of()`: as we saw above, the fraction field is the *bottom 52 bits* of the number, so this function should return just those bits (with zeros in the higher bits).

Hint: write a binary number (a bitmask) that represents the 52 bits we want. Use a base converter (there are plenty online) to get this number in hexadecimal or decimal, and write code employing the `&` operator to get the desired bits. (Recall that to include a hexadecimal number in a C program, you must put `0x` before it, e.g. `0xFF` for the hexadecimal number FF.)

Write some code in your `main()` to test your function. If you have finished Step 1 successfully, you can start with the `double` value and call your `double_to_long_bits()` to get its bits; if not, you can start directly with the `long` bits. Below are some test cases.

`double` value	`long` bits	Fraction field
0.0	0	0
5.0	4617315517961601024	1125899906842624
-1.0	-4616189618054758400	0
3.14	4614253070214989087	2567051787601183

b. Now do something similar for `exponent_of()`. This will be more complex because the exponent bits will need to be shifted to the right by 52 bits. I recommend *first* shifting the number (moving the exponent bits to the bottom) and *then* using `&` to get just the desired 11 bits with a bitmask. Below are some test cases.

`double` value	`long` bits	Exponent field
0.0	0	0
5.0	4617315517961601024	1025
-1.0	-4616189618054758400	1023
3.14	4614253070214989087	1024

If the exponent values seem nonsensical to you (why would 5.0 have an exponent of 1025? Wouldn't that give us a number like 2^{1025}?) then congratulate yourself for paying attention! This is because the exponent is stored in a "biased" form, being 1023 higher than the true exponent value. This is just how the IEEE 754 floating-point formats store exponents, to allow for both positive and negative exponent values for very large and very small numbers.

c. Finally, write and test code for `sign_of()` against the test cases shown below. This will be quite similar to the code you wrote for `exponent_of()`.

`double` value	`long` bits	Sign field
0.0	0	0
5.0	4617315517961601024	0
-1.0	-4616189618054758400	1
3.14	4614253070214989087	0

(In general, any positive number should have a sign of 0 and any negative number should have a sign of 1.)

Step 3

Goal Write a function to assemble a `long` value from a sign, exponent, and fraction.

Now that we can disassemble the bits into the sign, exponent, and fraction fields, we will code the reverse operation: taking a sign, exponent, and fraction, and place them together, ready to get reinterpreted as a `double`.

a. In `project.c`, fill in code for the `assemble_long_bits()` function. This function needs to left-shift each of the fields (sign, exponent, and fraction) by the correct amounts and use the `|` operator to combine them into a single value.

   Test your code by using the fractions, exponents, and signs from above in reverse; for example, a fraction of 1125899906842624, exponent of 1025, and sign of 0 should assemble to the long with value 4617315517961601024 (corresponding to the `double` value 5.0).

Step 4

Goal	Write assembly code to take the bits of a `long` and access them as a `double`.

Now you will fill in the `long_bits_to_double()` function (from `double_long.c`/`double_long.s`) to take the 64 bits of a long and reinterpret them as a `double` — the reverse of Step 1.

a. Return to your `double_long.s` and locate the section corresponding to `long_bits_to_double()`. The current version of the assembly code uses the `cvtsi2sd` instruction to convert the parameter (`long l`) value into a double and then returns it by placing it in `xmm0` (functions that return `double` values put their return values there instead of in `rax`). You will want to delete the `cvtsi2sd` line and instead copy the parameter directly into `xmm0` using `movq`.

b. Back in `project.c`, test your function by reversing the test cases from Step 1, checking that `long` values result in the correct `double` values. Here are some additional tests you can run:

`long` bits	`double` result
4613302810693613912	2.718
4728057454355411894	123456789.123
-4495314582499363914	-123456789.123

Step 5

Goal	Write a C program that manipulates `double` values by altering the sign, exponent, and fraction fields.

Now we can put together these pieces to manipulate `double` values!

Prerequisites

If you were able to get Steps 1–4 working, then you can implement all four of these functions. The `log_base_2()` function will only work correctly if you have completed Steps 1 and 2. The `negative_zero()` and `infinity()` functions will not work correctly unless you have completed Steps 3 and 4.

a. **`log_base_2()` function.** *(Requires completion of Steps 1 and 2.)* The logarithm base 2 of a number is what power 2 must be raised to in order to get the number. (For example, $\log_2 8.9$ is about 3.15381 because $2^{3.15381}$ is 8.9.) The logarithm base 2 is useful in many contexts; for example, $\log_2 n + 1$ is how many bits are needed to represent n in binary.

   It is very easy to calculate the logarithm base 2 of a floating-point number because it is simply the exponent. (More precisely, the exponent is the result of *rounding down* the logarithm.) Fill in the `log_base_2()` function, using `double_to_long_bits()` and then `exponent_of()` to get the parameter's exponent. (You will need to subtract 1023 from the exponent because the actual exponent value is 1023 smaller than what is stored in the number.)

Here are some test cases.

Number	Logarithm base 2
1.0	0
1.999	0
2.0	1
7.999	2
8.0	3
1234567890.0	30

b. **`negative_zero()` function.** *(Requires completion of Steps 3 and 4.)* One of the oddities about the floating-point format is that it contains both positive and negative zeros (of course, strictly speaking zero is neither positive nor negative!). "Negative zero" is represented by a `double` having a sign of 1, exponent of 0, and fraction of 0.

   Fill in the `negative_zero()` function to return the value of negative zero. (Use your `assemble_long_bits()` and `long_bits_to_double()` functions.) Test your function by printing out the result and verifying that it displays as –0.

   Note that you *could* implement this function with a simple "`return -0.0;`". For purposes of this project, however, you must use the functions you developed. 😉

c. **`infinity()` function.** *(Requires completion of Steps 3 and 4.)* Floating-point formats also have special values for ∞ and –∞. (These come up when, for example, you divide a humongous value by a miniscule one, resulting in something too large to represent in a `double` — for example, 10^{200} / 10^{-200}.) ∞ is represented by a `double` having a sign of 0, exponent of 0x7FF (the largest possible exponent), and a fraction of 0. (–∞ is the same but with a sign of 1.)

   Fill in the `infinity()` function to return the value of ∞. (Use your `assemble_long_bits()` and `long_bits_to_double()` functions.) Test your function by printing out the result.

d. **`times_power_2()` function.** *(Requires completion of Steps 1–4.)* We can multiply or divide integers by a power of 2 by doing shifts: `n >> 3` is dividing by 8 (2^3), `n << 4` is multiplying by 16 (2^4), etc. The >> and << operators don't work with floating-point types, but we can get the same effect by adjusting the exponent field of the number, adding (or subtracting) to it. For example, multiplying a floating-point number by 8 is the same as increasing its exponent by 3, and so forth.

   Fill in the `times_power_2()` function to do this; you will need to get the number as a `long`, split it into its three fields, add the power to the exponent, reassemble the fields into a `long`, and finally turn that `long` into a `double`.

   To test your function, plug in various numbers and check that it multiplies (or divides) by the right power of 2. For example, `times_power_2(1.0, -3)` should result in `0.125` and `times_power_2(3.14, 1)` should give `6.28`.

e. **`round2()` function.** *(Requires completion of Steps 1–4.)* Since floating-point numbers are stored in base-2 scientific notation, we can round a `double` to a certain number of bits by cutting off the fraction after that many bits. For example, rounding π to 0 bits would give `2.0` ($1.00000... \times 2^1$), to 1 bit is `3.0` ($1.10000... \times 2^1$), and so on.

Fill in the `round2()` function to do this. This will be similar to how you built `times_power_2()`, but instead of adding to the exponent, you will need to adjust the fraction field to cut off bits from the right end. The easiest way to do this is to right-shift it, moving the bits you don't want past the decimal point so they disappear, then left-shift it again by the same amount. (Be careful that you're shifting by the right amount! Remember that there are 52 bits in the fraction.)

Here are some test values.

Number (`x`)	Number of bits	Result of rounding
3.141593	0	2.0
3.141593	1, 2, or 3	3.0
3.141593	4, 5, or 6	3.125000
3.141593	13	3.141357
-1.0/3	4	-0.328125