Assignment 1

The differences between M1 and Intel processors while compiling a C code

Pedram Pasandide

Due Date: 12 February

1 Introduction to an Example

Take a look at the following code:

```
#include <stdio.h>

int main()
{
 const double a1 = 1e18;
 double a2 = 1e18;

 double c1 = 0.1 * a1 - 0.1 * a1;
 double c2 = 0.1 * a2 - 0.1 * a2;
 printf("c1 = % e\n", c1);
 printf("c2 = % e\n", c2);
}
```

If I compile the code with `gcc` on Linux or MacOS with Intel processor, and run the object file, I get:

```
c1 =  0.000000e+00
c2 =  0.000000e+00
```

However, when I compile and run the program on MacOS with M1 processor I get:

```
c1 =  0.000000e+00
c2 =  5.551115e+00
```

1.1 Why? (3 points)

In one paragraph explain why this is happening. Even if you don't have M1 processor, you should be able to answer this question by some research.

1.2 Why 5.551115? (2 points)

If you don't have MacOS with M1 processor, you can try the following code using `fma` function from `math.h` library.

```
#include <stdio.h>
#include <math.h>

int main()
{
 double m = 0.1;
 double a1 = 1e18;
 double c1 = m * a1 - m * a1;

 // Using fma function
 double c2 = fma(m, a1, -m * a1);

 printf("c1 = %e\n", c1);
 printf("c2 = %e\n", c2);
}
```

Don't forget to add `-lm` flag to compiler when compiling this code. Also add `-std=c11` flag, specifying the C11 standard, which includes support for the `fma` function. To understand what `fma` function does you can check Appendix. The purpose of `fma` is to increase the accuracy. However, even on Intel processor the code output is:

```
c1 =  0.000000e+00
c2 =  5.551115e+00
```

This is the same issue which was happening on M1 without using `fma`. In this code if we change the value of `m=0.1` to `m=0.2`, we get:

```
c1 =  0.000000e+00
c2 =  1.110223e+01
```

For `m` equal to `0.3`, `0.4`, and `0.5`, on **both Intel and M1 processors** we get

- `c2 = -1.110223e+01`,
- `c2 = 2.220446e+01`, and
- `c2 = 0.000000e+00`, respectively.

Why are we getting these numbers?

Did you find the answer? If you have MacOS with M1 processor, it is time to email Apple company and ask for the refund. Just kidding! Don't do that. The purpose of this question was just to remind you that on different Operating Systems, we **might** have different results. As a programmer, it is extremely important to always check your program many times, to make sure you have the same results on different Operating Systems. You can check your codes on online platforms (if you don't mind to share your codes!). For running code online, you can use platforms like:

- **OnlineGDB**: OnlineGDB provides an online compiler and debugger for C and other languages. While it doesn't specifically mention the processor architecture, you can use it for general C code testing.
- **Repl.it**: Repl.it supports various languages, including C, and provides an online coding environment. However, it may not specify the underlying hardware architecture.

2 Submission On Avenue to Learn

Only `report.pdf` file, including all your codes (if you have) and the answers to the questions.

3 Appendix

The `fma` function, which stands for "fused multiply-add," is a mathematical operation commonly used in numerical computing. It performs the calculation $x \times y + z$ without rounding the intermediate result of $x \times y$. The key advantage of the `fma` operation is that it **minimizes** rounding **errors** that can accumulate in floating-point arithmetic.

The typical sequence of operations $(x \times y) + z$ involves two separate steps: multiplication $(x \times y)$ and addition $(+z)$. In standard floating-point arithmetic, rounding errors can occur at each step, leading to loss of precision. The `fma` operation fuses these two steps into a single operation, reducing the potential for rounding errors.

The formula for the `fma` operation is:

$$fma(x, y, z) = (x \times y) + z$$