Profiling and CPU Architecture Report

https://github.com/grettebyang/Systems-Assignment-1.git

For this assignment, I used the std::chrono high resolution clock to track the time spent in a part of the program. I also used the google profiler to profile the program, but the timer provided sufficient enough information for this task.

Part 1:

Instead of 1M elements, I used 1B elements to get a better measurement of how long the program spent in the loop. In part 1, using the chrono clocks to measure the time, the program spent 122 nanoseconds adding every element to the sum.

Part 2:

After adding in the three additional operations to my loop, the time spent clocked in at 103 nanoseconds. This is unexpected to me, since I would have thought that because there are four times as many operations happening, it should take about four times longer to run. This is because of instruction level parallelism. Because the operands are already in the registers at the time of performing the four operations in one loop, the processor can just execute them all at once.

Part 3:

When putting multiple operations in one line and adding it to a single sum, the time spent was 181 nanoseconds, which is longer than the previous two versions of the function. This is because the processor needs to compute each stage of the expression separately before it can multiply the operands by each other. So it must finish computing the first expression before it can compute the second one, meaning the operands won't stay in the same registers and be reused for the next operation like in part 2. As a result, the processor takes almost twice as long to compute the entire expression in each loop.

Part 4:

After implementing the if statement to the previous part and running the program several times, the time spent in the loop varies from around 106 nanoseconds or up to around 470 nanoseconds. This is happening because of predictive pipelining. The processor is trying to predict what the outcome of the if statement will be based on previous iterations of the loop. If it predicts the statement to be false, it will pipeline the next operation to optimize the performance. But if it is wrong and it learns that it should not execute the operation, then it will need to do a pipeline flush, which is an expensive occurrence. Because the value is completely random, there is no way for the prediction to be mostly right or mostly wrong every time, so this gives us varying results which could either improve the performance of our program or slow it down.