Aidan Justice

CENG 320

20 April 2022

# Bigint Assembly Optimization

The objective of this assignment was to take the big integer IDE implemented in C and try to implement it in assembly to get the program to run faster. We were required to implement the bigint_adc function at a minimum in order to complete the assignment. I used a Raspberry Pi to write and test my code. Due to assembly being a challenging language for me to grasp, I ran into several problems while trying to implement the add with carry function which was the only function I could fully complete in the time period. I ran into segmentation faults, problems with the carry, and problems with signed numbers. After finally finishing implementing what I could, I was only able to achieve a speedup of 0.99, staying on par with the C implementation.

The fast version of adc works by first comparing the sizes of the bigints you want to add and then swapping them if the size of l is greater than the size of r, this is to maintain functionality throughout the rest of the function. After it allocates space for a new bigint, it runs through each chunk, adds them together, and stored them in the new sum until we hit the size of l. Next, we find the sign of l and sign extend it by setting a new chunk equal to either 0 or -1. Now, we add the remaining chunks of r to the sign of l and store it in our new sum. Finally, we find the final chunk in our new sum by adding the sign of l, the sign of r, and our carry together and then we trim the new sum with bigint_trim_in_place.

When I first wrote this function, the issue I ran into was segmentation faults. This was just an issue with how I was accessing the elements in the array of chunks. I initially was trying to increment through the pointer to the bigint struct and not the array of chunks. All I had to do to fix this, was to load the pointer to the array of chunks to a register and iterate through that.

After I fixed my issue with segmentation faults, I ran my program only to find that my answer was not close to the actual. Using the gdb debugger, I found that my issue ran with how

I got the carry. First, I was directly implementing the C code and that meant I was using the initial method of finding the carry by shifting the result by the chunk size and using that as a carry. This proved to not be very useful since assembly can set the carry bit by using the instruction “adds.” This will add two numbers and then set the carry flag if there is an overflow in the sum. By using cs to check the carry flag and cinc to set my carry, I was able to get past this issue.

After changing the implementation of how I set the carry, my function could now add two positive numbers but strayed far from the answer when negative numbers appeared. While debugging, I found the issue to lie with how I set the sign chunk. When I first looked at the C code, I interpreted “(schunk)-1” as, the size of schunk minus one, instead of typecasting negative one to a schunk. Once I found this issue, I changed the move instruction in the if-else statement to move either a 0 or a -1. This change solved my final issue with the function and it could now add together negative and positive numbers.

| | Original C | My bigint_adc |
|---|---|---|
| Time in seconds | 99.5746 | 101.491 |

As seen in the table, my implementation of bigint_adc was about 2 seconds slower than the original C implementation, with the original taking 99.5746 seconds and my implementation taking 101.491 seconds. Although I did not get the speedup I had hoped for when I was given the assignment, I am content that it did not hurt the performance in a substantial way as it was a 0.99 speedup. All in all, I’m starting to understand why assembly is capable of helping performance, however, since I don’t have a complete grasp of the language, I’m not capable of utilizing it to its full potential.