Keiran Berry

CENG320

Program 1 Report

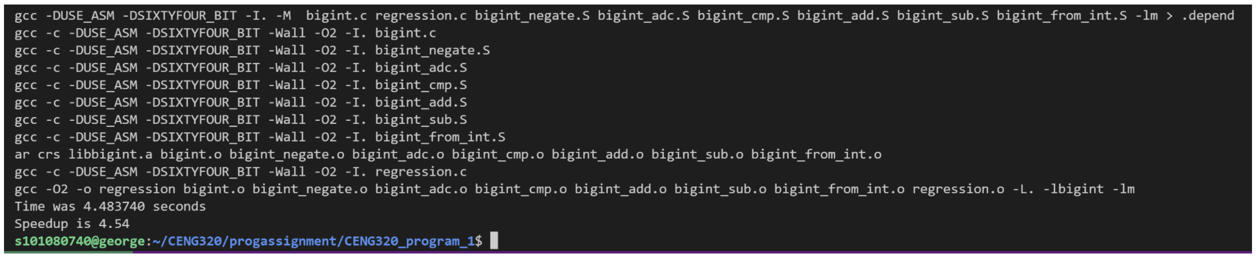
20 November, 2022

The goal for this project was to take an existing, running program which has been implemented in C and learn how it can be optimized using Assembly. There were few specifications, other than to implement bigint_adc in Assembly, and make the program run as quickly as possible. This program served as exceptional practice for both optimization and assembly, as well as implementation of assembly functions into an existing C program. It also forced me to learn much more debugging, since the integers used are so big that it is near-impossible to calculate by hand and check.

My optimization of the bigint program ended at 4.54 speedup, and the history of optimizations which I made is as follows:

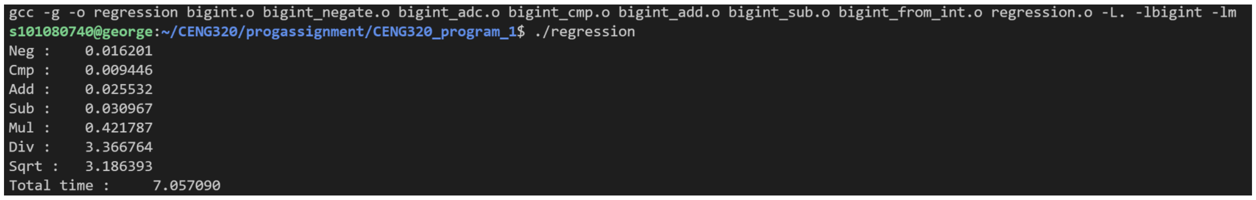
Change Made	Resultant Speedup
Start (no optimizations)	1.00
bigint_adc in Assembly	1.23
bigint_trim_short implemented in adc only	1.84
bigint_cmp implemented in Assembly	1.8
bigint_trim_short implemented throughout	2.26
bigint_cmp in Assembly with no subtraction	3.81
Optimizations in bigint_cmp	3.84
bigint_add implemented in Assembly	3.86

Fixed off-by-one bug in bigint_cmp	3.78
Optimized C code for bigint_shift_left	3.88
Optimized bigint_adc	3.9
Optimized bigint_negate	3.92
Optimized bigint_sub	3.94
Optimized bigint_free	4.01
bigint_sub in Assembly	4.45
Optimized bigint_from_int	4.48
Changed bigint_add calls	4.54
End (all optimizations)	4.54



```
gcc -DUSE_ASM -DSIXTYFOUR_BIT -I. -M  bigint.c regression.c bigint_negate.S bigint_adc.S bigint_cmp.S bigint_add.S bigint_sub.S bigint_from_int.S -lm > .depend
gcc -c -DUSE_ASM -DSIXTYFOUR_BIT -Wall -O2 -I. bigint.c
gcc -c -DUSE_ASM -DSIXTYFOUR_BIT -Wall -O2 -I. bigint_negate.S
gcc -c -DUSE_ASM -DSIXTYFOUR_BIT -Wall -O2 -I. bigint_adc.S
gcc -c -DUSE_ASM -DSIXTYFOUR_BIT -Wall -O2 -I. bigint_cmp.S
gcc -c -DUSE_ASM -DSIXTYFOUR_BIT -Wall -O2 -I. bigint_add.S
gcc -c -DUSE_ASM -DSIXTYFOUR_BIT -Wall -O2 -I. bigint_sub.S
gcc -c -DUSE_ASM -DSIXTYFOUR_BIT -Wall -O2 -I. bigint_from_int.S
ar crs libbigint.a bigint.o bigint_negate.o bigint_adc.o bigint_cmp.o bigint_add.o bigint_sub.o bigint_from_int.o
gcc -c -DUSE_ASM -DSIXTYFOUR_BIT -Wall -O2 -I. regression.c
gcc -O2 -o regression bigint.o bigint_negate.o bigint_adc.o bigint_cmp.o bigint_add.o bigint_sub.o bigint_from_int.o regression.o -L. -lbigint -lm
Time was 4.483740 seconds
Speedup is 4.54
s101080740@george:~/CENG320/progassignment/CENG320_program_1$ 
```

*screenshot displaying final run of ./report_time.sh with speedup of 4.54



```
gcc -g -o regression bigint.o bigint_negate.o bigint_adc.o bigint_cmp.o bigint_add.o bigint_sub.o bigint_from_int.o regression.o -L. -lbigint -lm
s101080740@george:~/CENG320/progassignment/CENG320_program_1$ ./regression
Neg :    0.016201
Cmp :    0.009446
Add :    0.025532
Sub :    0.030967
Mul :    0.421787
Div :    3.366764
Sqrt :   3.186393
Total time :     7.057090
```

*screenshot displaying successful final run of regression

The first performance enhancement is the bigint_adc function, which is an add with carry. My implementation of this function loops through the chunks of the two bigints, adding any previous carry and then setting the next carry, if there is to be one. This goes until there are no more chunks which need to be summed up, and then the resultant sum is returned. This change from C to Assembly brought the speedup to 1.23.

The next step which was taken to optimize the program was shortening the bigint_trim function, as the last bit of it is only necessary for the bigint_div function. By taking out the last if-else statement, we can see significant performance improvements in the other functions. I implemented a separate, second function called bigint_trim_short, which would be the shortened version of bigint_trim to be plugged into the functions which could use it. In just bigint_adc, the implementation of bigint_trim_short brought the speedup to 1.84. Later on, when the function was used in all functions but bigint_div, the speedup jumped to 2.26.

Since the heart of all comparisons in the program is the bigint_cmp, I translated the bigint_cmp which was implemented in C into Assembly. This ended up bringing the speedup down, since the main issue with bigint_cmp was the call to bigint_sub. After working through ideas on how to compare without subtraction, I settled on comparing the sizes, and then comparing the individual chunks starting with the most significant until one or the other is proven greater. If the size is greater, then the number is greater, unless any negatives are encountered. With two negatives, the opposite is true, and with one negative, the negative is the smaller one. In either case, if the sizes are the same, the chunks can be checked one by one until a difference is found. Implementing this new bigint_cmp skyrocketed the speedup to 3.84.

The bigint_cmp function had one bug which I corrected, in that if two chunks were off by one the function would not detect it. This was due to the fact that both chunks are shifted right by one in order to not worry about a sign bit, since a 1 in the most significant bit signified a negative sometimes where there should not be one. This change works in all cases except for when a chunk is off by one, which was not encountered in the tests included with the programs;

however, I wanted to ensure that I was writing responsible code which would work correctly. Fixing this bug brought the speedup down a bit, but having working code is more important than having slightly quicker code.

From here, I employed many small improvements in order to get some speedup where I could. These performance enhancements were less notable, such as implementing the bigint_add function in Assembly. This was later changed to be a direct call to bigint_adc with a carry of 0, since that is all that bigint_add does anyway. Some speedup was achieved through changing bigint_sub from a bigint_complement and bigint_adc with carry of 1 to be a bigint_negate and bigint_adc with a carry of 0. Later on, I implemented bigint_sub in Assembly, which resulted in significant performance improvement. This algorithm for bigint_sub is extremely similar to that of bigint_adc.

Another small improvement was taking the call to shift by chunks out of bigint_shift_left, since each function which calls bigint_shift_left checks for chunk shifts beforehand. Finally, taking out unnecessary ifs and allocations in bigint_negate, bigint_free, and bigint_adc each got small speedups. All of these miscellaneous optimizations resulted in a speedup of 4.54, with a final runtime of just under 4.5 seconds, which is a major improvement over the original time of well over twenty seconds.