Inline Assembly Expression Tree Calculator

Eric Cacciavillani and Alice Easter

April 29, 2018

Research Topic and Goals

The goal of this project is to recreate a past program that used a tree to calculate any math equation passed to it. This system would be using C++ to create and handle the tree structure. Then using a combination of inline assembly and external assembly files to handle the logic/math to find the solution. In addition, we wanted to compare clock cycles from both implementations to document the level of improvement between the two variants. Continuing on, we also wanted to find ways of re-implementing library provided functionality in assembly. Finally, once the project is completed I am curious about seeing the overall speed differences between the original implementation versus the assembly execution.

Proposed Design Schema

1. Convert user input equation to a prefix notation
2. Using the prefix notation of the equation, create a tree structure in C++ to maintain values, operators, and user created functions
   - Deconstruct leaves as we move through the tree to ensure proper usage
3. Error handle any and all syntactical mistakes
4. Handle any mathematical errors that could be thrown
   - Ex: Divide by zero
5. Handle overflow and underflow on variables
6. Determine for the given operation to use either the FPU or CPU for the given data types.
   - Another option is just to use the FPU entirely and have floating bits cleared out

Anticipated Challenges

- One of the biggest concerns with this project was finding the proper syntax to properly convert and improve upon the preexisting code to it's assembly variant.
- Properly dealing with floating points in the FPU during floating point operations and being able tell which processing unit should be used with the provided value passed.
- Implementing negative number handling and negative operations.
- Accounting for overflow/underflow of values for user input and how to use register to better handle the problem.
- Speeding up the creation of the tree and syntax analysis of the equation.
- Adding more scalability to allow other users/programmers to create their own functionality calls in the calculator.

Describing Interest with the Specific Topic

- Data structure design has been a big part of our education, and as such has become something we have somewhat of a passion for. It may be rather strange, but it can be quite therapeutic to create high performance data structures.
- In general, the concept of taking the scalability of C++ and pairing it with the raw speed of assembly can be quite enjoyable.
- This was a project that Eric really appreciated in our Data Structure class so it felt enjoyable to have a throw back to it.
- We were curious to see how much improvement we could see between the assembly and C++ variant.

Implementation

In the actual implementation of this project we created two separate environments, one to run the C++ variant, and one for the assembly. While the C++ version was fairly straightforward, the assembly faced a few challenges. The biggest of said challenges is that the C++ compiler does a massive number of optimization to C++ code, while the assembly code remains unaffected and unoptimized at that level. As such when we began our project we noticed that the C++ version was beating the assembly version by a wide margin. We attempted to remedy this issue by using SSE functionality for our floating point operations, however as this application was not meant to handle multiple inputs and as such there isn't multiple data to pack.

The workload of this project was accomplished through a large number of late night and a lack of sleep. While the programming required wasn't as complex as it could have been, the sheer amount of code and analysis generate for this project was very time consuming. The original project needed to be improved through threading deletion of trees, syntax updates and bug fixes, an attempt at handling negative numbers (with little success, and not enough time to complete), and a proper file system for handling test cases. Eric handled most of the updates to the original project, as he was more familiar with the software, as well as calculating the comparisons between the two applications, while Alice handled most of the logic for asm translations.

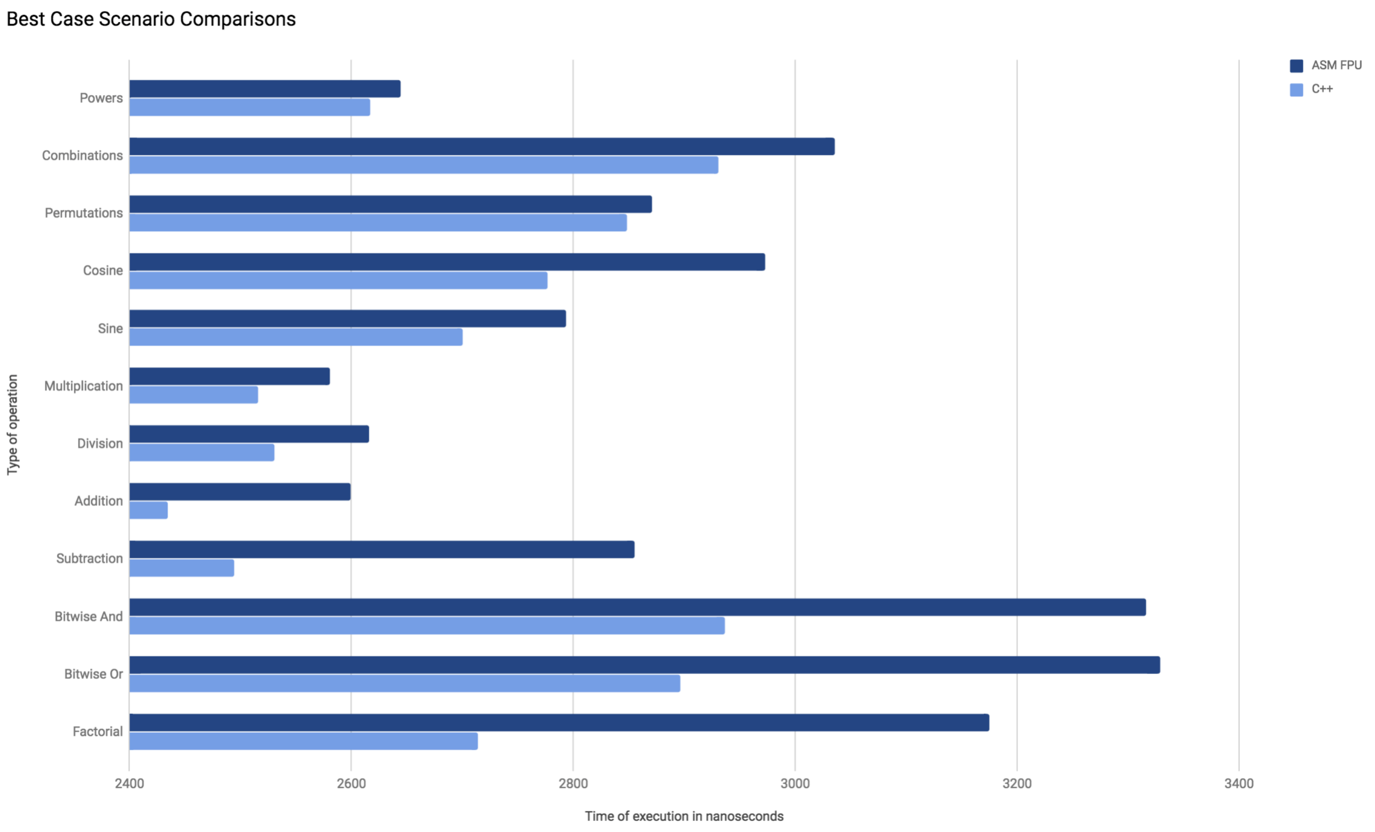
There were a number of features that we were originally going to incorporate into this project that ultimately we left out from our final design. From our proposed design schema alone, we decided not to go with prefix notation for expression input as infix comes more natural to standard users and didn't cause any more work relatively than the prefix variant. We didn't include any checks for division by 0 or other mathematical errors as we deduced that we would be giving the C++ version an unfair advantage as we know how to do proper try / catch's in C++ easily enough, but would need to hack something together to establish this on the assembly side. The only other thing we really ignored was using assembly from an external file. We determined that there was no real benefit to using external assembly, and as such did all of our assembly coding inline.

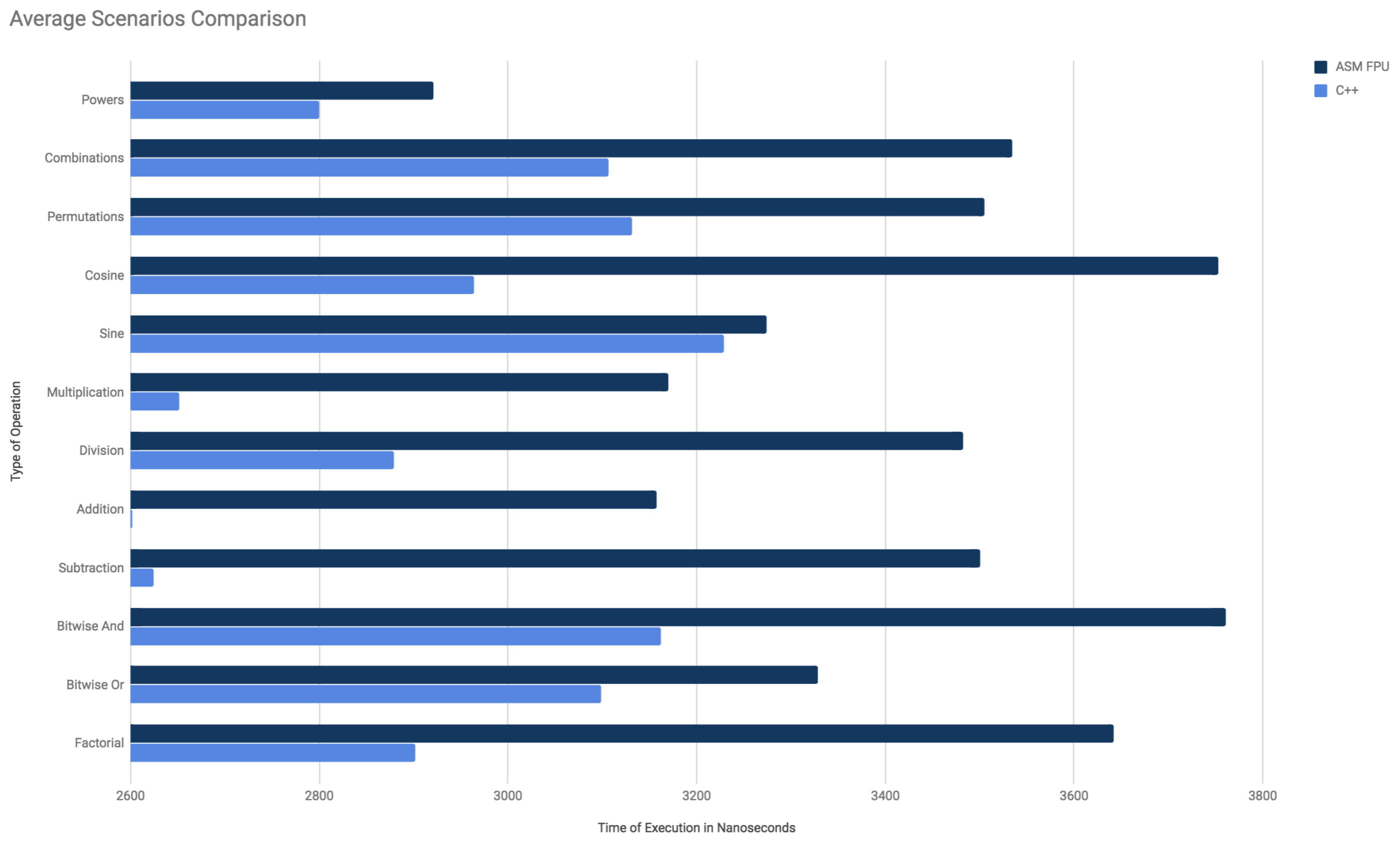
We also originally anticipated adding in a few more mathematical features, namely bit-shifting and xor, but we came up with issues for both. While xor is really easy to implement, our system for parsing the input used the variable x for all of its data validation, and adding in xor caused some issues parsing said text, and rather than rewrite our entire system at the very end we decided to leave the feature out. Furthermore, in the case of bit-shifting, the main reason we wanted to use bit-shifting was more or less to give the assembly side a win. However, we learned that with bit-shifting we had to give the number of bits shifted as a static number, not a register or variable, so the only way we could figure out how to implement it would be with jumps, which would not give assembly the advantage we wanted.

Data Visualizations

First the operation creates a tree with infix notation. We do not have the creation of the binary tree as part of the timing process because both implementing have the same exact process for creating the tree and the extra data could leave room for error. Afterwards, we have a highly optimized clock that calculates the time it took to evaluate the tree. Note the tree does not deconstruct the nodes as we recursively iterate through it; nodes are deleted upon creation of new tree systems and during the deconstruction of the entire system. We then insert the given time in nanoseconds to an array to get the given standard deviation, mean, and best times. Finally we pass the output of overall information into a test file. After retrieving the actual data, we used Google Sheets to create proper graphs of the data, and used MiniTabs to establish that C++ is in fact faster than Assembly using 2-sample T calculations.

As far as the actual separation of the data in the graphs, we determined that we would make graphs for two separate instances, the best case scenarios over 10,000 tests, and the average case over 10,000 tests. Best case scenarios had the two variants relatively close to each other for most operations, however the average scenarios is where the most difference was seen.





In the MiniTabs section we discovered that almost all of our results when comparing the results of C++ (Sample

1) to our ASM(Sample 2) had p-value bellow .0001. Thus, meaning we can accurately discern the given range differences between the two sample sets in each given operator case. See our full project repository to see our all the data we have gathered between these results. Calculator Comparison Project

Data Evaluation

In the end, our data supported the idea that C++ is significantly faster than inline assembly. There are a number of reasons why this is not entirely accurate, and that is absolutely up to to the current skill levels of us as programmers. The reason that the C++ code is so much faster is due to the fact that the compiler optimizes the code so that when it compiles to assembly it is as efficient as humanly possible, and those individuals who have created these compilers have dedicated their lives to said projects. While it is not impossible for us as programmers to beat those individuals who create compilers, it definitely requires a large amount of skill and/or luck.

There was in fact one instance where the inline assembly beat the compiler, and that was with the equation of 1 divided by 9. The assembly side beat the C++ side by 265 nanoseconds in this one instance, however, in every other instance of division that we tested we found that the compiler beat us, and as such we decided to note it in this report, but ultimately leave it out of the graphs as it was outlier data.

Conclusion

Overall, we were wrong in our initial assumption that assembly would add any benefit to C++ programming. As it turned out, C++ beat assembly at almost every turn by quite a wide margin. Although this project was a lot of fun, and assembly does in fact offer quite a lot of raw speed, it does come down to how you use such a language. Ultimately in future programs, especially those where we aren't taking advantage of things such as packed SIMD operations, it would be relatively useless to incorporate such a language.

References/Citations/Tools Material

- Online prefix notation calculator
- Detailed example of setting up a expression tree with prefix notation
- Downloads Ti-84 High Level documentation to pull ideas to implement extra functionality
- Converting Expression Trees to assembly code
- Cute online C++ to assembly code.(Might make syntax translations easier)
- Handling tree in pure assembly