Part 1, Fibonacci Program

This part consisted of writing assembly code to get the Fibonacci number for a given number n. The program had to be written in two different ways, iteratively and recursively.

The iterative method was done by pushing all the Fibonacci numbers to the stack. It starts by pushing 0 and 1 to the stack, then the loop starts. The loop pops the two top numbers from the stack (n-1), (n-2), adds them together, then pushes the new number (n) along with the previous two to the stack. Once the index (number of times the loop has iterated) is equal to the number we were looking for, the loop breaks and the number we were looking for is popped and sent to r0.

The recursive method has a sum register that gets added to every time a Fibonacci number reaches 1 or 0. The program works by linked branching back to the top of the program twice, in order to get the n-1 and the n-2

A problem faced with the iterative method was that after the program had run, the stack was still full of numbers we didn't care about anymore, so a subroutine had to be written that would clear the stack. The creation of this subroutine also meant a special case was needed if n was 0 or 1 so that the stack would be empty, and the correct number was returned.

The recursive program had problems with where to store variables. At first, I tried storing the sum in a variable register, however I was getting errors that the register was getting clobbered

because I was modifying its value in subroutines and not respecting caller callee conventions. Since it was getting returned, I decided to move it into an argument register. I also tried storing the n in the stack, but I was getting clobbering errors because when I tried to change the n in subroutines, I was unable to push the initial values of variable registers because I had to pop the n from the stack, so I simplified my program and the n would just be modified in the main program, doing that also fixed an error I was getting from moving the stack pointer in a subroutine.

The iterative method could be improved by not using the stack at all. Three variable registers could be used, which would reduce the runtime of the program significantly, because reading and writing from memory take an order of magnitude more time than just operating on registers. It would also require less subroutines.

In the demo, the TA pointed out that my program wasn't true recursion, since I was making a tree and adding the leaves to my big sum instead of returning a value at the end of each recursion, so I would change my program to have a couple registers to hold the values of the returned recursions instead.

Part 2 2d Convolution

This program would modify an image (a 2d matrix) using values from another smaller 2d matrix

It was written using 4 for loops, which was done through branching with 4 different indices that were all stored in the stack and popped only when needed to save on registers used. The program must calculate the address of all the needed items in both matrices by multiplying the y value by the width of the matrix and add the x value to find the index, then multiply by four in order to turn it into an address. Since it is impossible to store a 2d matrix in memory, it was just stored linearly as if it was a normal array. The program also must calculate a couple temporary values to use as indices in the fx matrix as well. The program is simple otherwise, it just iterates at the end of each loop and does a normal branch back up to the top.

Not many challenges were faced in the writing of this program, however a significant portion of it had to be rewritten due to misreading the problem and the way it worked had to be rethought. Other simple issues popped up, including slight errors in instructions that led to incorrect values being calculated.

There might be another solution that does not require using the stack, just holding the values in variable registers, as well as the fact that a decent amount of code could be simplified. I have several parts that are three lines that could be reduced to one.

Part 3, Bubble sort

The program sorts a list of integers, using the bubble sort algorithm.

The program is two simple loops using branching with two indices that count the number of times it has looped. The bubble sort algorithm works by comparing all the values to the index right next to them and swapping them if needed. The program finds the two addresses of the side-by-side elements in the list, loads their values, and compares them, swapping if needed.

No real challenges were encountered in writing the solution, but the code could be made a bit more efficient by rewriting some code and condensing parts that were split up in three lines into just one line. Also there are many much faster sorting algorithms that could be implemented, since the bubble sort algorithm is known to be one of the slowest sorting algorithms, since it operates in n^2 time