**Project 1 Summary**

Getting Started: To test the project, please change to the source directory and issue ‘make’ to use our makefile which will compile memory.c and cpu.c using gcc. The project can then be run by issuing ‘./cpu program.code interrupt.code’.

The purpose of this project is to explore the interaction that takes place between the processor and the main memory of a computer. We saw many helpful metaphors along the way – variables act as registers in the CPU, and the pipes made in C helped us to simulate the processor’s fetch cycle with the memory and cpu communicating via load and store operations. These are simplified concepts but it is important to understand how they work in order to understand computer architecture and the operating systems that run on top. By implementing an easy-to-understand model of processor-memory interaction, we can see some of the design issues that faced the first builders of Von-Neumann machines. In particular, because the instructions we used have simple representations (1, 2, 3, etc.), it is easy to observe how a CPU actually operates on a series of binary data held in memory, where user data looks extremely similar to program instructions. It is important to understand that the processor gets all of its information from the same place, whether that be instructions or user data, and that we must give context to this data in order to make it useful.

In order to effectively model process-memory interaction, we created a simple virtual machine that is able to read a small number of different instructions, and we implemented the instruction logic using a backend written in C. The cpu.c code creates two pipes (for input and output), forks the memory process, and then begins executing the instructions from location 0 onward. The memory.c code loads a program file containing code written in our assembly language, as well as an additional interrupt handler if provided as an argument. After reading the commands into our main memory data structure, the memory code listens to requests from the cpu.c code which will read and write to the memory. The programs supplied for us to run are small and so we did not encounter any issue that required memory protection, although it should be noted that certain conditions could give rise to instability. This is due to the way that the stack grows from position 999 towards position 0, where the program code resides in memory. If the program code were long enough, the stack could eventually overwrite some of the program itself.

Our team was able to collaborate effectively through the use of git version control, with a central repository located on GitHub (<https://github.com/dvanallen/CE4348/tree/master/assignment1>). We met after class to talk about how to best divide the work, and split it into three main parts: the memory code, including read and write access to the data structure; the processor instruction set; and the main code that would first load the input program file into memory and then set up the fetch and execute cycle within the processor. Ben got us started early by setting up the process of reading a file into our memory data structure. He used C++ to read in the supplied program file and check for errors at runtime. Daniel then added the basic design for the CPU code and began implementing instructions. He also produced this documentation and hosted the project on his GitHub account. Desmond finished up by rewriting some of the input/output in C and finishing off the process communication and instruction implementation. Altogether we were a very effective team because we started the project early and were able to define clear responsibilities so that the work was shared with minimal overhead when putting it back together. This was the first project that we have done together using GitHub, and getting used to collaborating using a version control system was a valuable part of this assignment, even though it was not necessarily a requirement.