Evan Wright

emw200004

CS4348.006  
  
 The purpose of the project is to model how the CPU works with main memory in order to execute instructions from a provided program. It simulates how two separate and simple hardware components, or in this case processes, can communicate with each other in order to achieve something complex. While this is an abstraction and built at a far higher level than ordinary, it is helpful to understand the intricacies of each piece of a computer and the difficult task of managing them in the form of the operating system.

This project was implemented in Java using the Runtime exec method to create separate processes. We start by running the CPU and providing it with command line arguments in the form of the file name and the timer in the form of an integer representing how many instructions to wait before interrupting. We first initialize the registers that will be used along with some other helpful variables such as an instruction counter used with the timer, and a mode represented through 0 and 1. After checking to make sure the passed-in arguments are acceptable, we create a separate process for the Memory file by using the Runtime exec method. Afterward, we quickly set up an output stream for writing in the form of a print writer and an input stream for reading in the form of a scanner. This will allow us to communicate with the memory on the CPU side, so we print the file name to the memory process. Meanwhile, the memory process has begun running and will immediately initialize an array of size 2000 to correspond to the storage space in true main memory. First, the memory sets up a reader to communicate with the CPU from its own side and uses it to read the file name. Then, the memory attempts to open the file, and upon success, it will parse through the file to load instructions into the array. If it finds an integer, then it corresponds to a value to be put into memory at the next available address. If it finds a dot, then it means the program is trying to tell it to put the following instructions at the given address, so the dot is removed and the remainder is parsed to set our address to start at that index. Everything else is ignored, so that comments don’t cause any problems.

After the file contents have been loaded into the memory array, the memory will enter a loop over the CPU reader’s output where it will continuously read lines from the CPU. These lines will be parsed into various commands, such as the first index being either a 1 or 2 corresponding to read or write. The second part will be the address specified for the command to be executed. If we are writing data to memory, then there will also be a third part that corresponds to this data. If we are reading data from memory, then we must print the result back to the CPU for it to read. However, in order for the memory to do any commands, it must receive them from the CPU. So back in the CPU, we will have entered a loop to fetch and execute indefinitely. First, we must check if it’s time for an interrupt by checking our instruction counter and comparing it to the given timer, and if so then we must execute the interrupt handler. In the interrupt handler, we switch our mode to kernel mode i.e. 1, and push our necessary registers onto the stack so we can utilize system code without losing any data integral to the normal program. Ultimately, the interrupt handler will run through the normal code execution process, except with a few caveats: we check if we’re already in an interrupt before entering the interrupt handler to prevent an infinite loop, and we must be in kernel mode to execute the instructions past address 1000 (the system memory). Following that, we send a request to memory to read at the current PC, which is initially 0. Then the memory processes will read that request, split the commands into “read” and “address 0”, and then get memory[0] and print it back to the CPU to read. That value is then passed in the CPU to the IR register, where it is determined how to process the instruction and thereby execute whatever action is needed. We achieve this by using a very large switch statement with a corresponding number for each possible instruction. Whatever value is received into the IR, the execute method will process correctly with the switch statement. Some instructions require incrementing X, pushing a value onto the stack (which requires an additional call to write to memory and then adjusting the SP), loading values into registers (which requires an additional call to read from memory and putting that following number into the register), etc. The PC is adjusted to correspond to the instructions read so that the next instruction will be read appropriately, except in the case of a jump instruction where the next instruction will be at a preset PC.

Personally, I had a lot of trouble with getting the Java Runtime methods to work. I was given the warning that it had been depreciated, but the instructions said that we had to use that particular method so I didn’t want to try the new ProcessBuilder version. I tried reconfiguring my IDE to as far back as Java 1.8 and it still wasn’t creating my Memory process correctly, and running it in the terminal also gave just an Exit code 0. It took several days to fix the problem as well as the advice of the Professor, but thankfully we were able to fix it by passing a different argument as the string version is now depreciated but we can still make an array of strings. Other than that, this project was very engaging and it feels very rewarding to understand how the components of a computer work together to execute instructions, even if it is at a high level of abstraction such as this. Ultimately, I can only hope that I don’t have extraneous issues bogging down my time in Project 2, but we live and we learn.