**Word Count: 742**

**TEAM 6:**  
\* Anas Saad (2510059)  
\* Ahmid Omarzada (2527809)  
\* Sayed Hashemi (2508673)  
\* Ahmed Youssef (2507690)  
\* Usama Bin Fakhar (2530869)

Compile command for simulator: g++ simulator.cpp  
Compile command for assembler: g++ assembler.cpp

***Simulator:***

We first approached the simulator by making a simulator class, we used a vector of strings for the store/memory and we initially were going to represent the registers using bitsets from the bitset library, but that wasn’t the best approach because we couldn’t work with the registers in terms of their decimal counterpart, like the CI or the accumulator, so we decided to use strings for any binary number in the program, and the CI as an integer because of the incrementCI() function, and anytime we want to use their other counterpart we would use the btod() and dtob() functions that convert from binary to decimal and vice versa.

There were also a lot of problems with validation, so we implemented 5 functions that validates each of the following: the machine code as a whole, each machine code line, the opcode, the operand, and any individual memory address. These were used throughout the code to ensure nothing goes wrong. As for the displaying the registers, there are 5 functions that display them in binary form for the user to see.

For the main fetch-execute cycle, it consists of incrementing the CI, fetching the opcode and operand from the machine code line, decoding the opcode and the operand address, and ends the program at the STP opcode. The problems we had was that we had a switch statement for the opcodes in the decodeAndExecute() function but it wasn’t working for some reason so we swapped it for if else statements and it started working fine. The last problem we faced was with the fetching process, where we didn’t properly understand the concept of absolute addressing so we were mistaking the first 5 bits of the machine code line as the actual operand, which we later understood and fixed it so that everything works fine.

***Assembler:***

We initially approached the assembler by not using object oriented programming as we wanted it to be simple procedural programming but that proved to be difficult as the code got very complicated and long, so we revamped the whole thing and created a header file containing an assembler class which contains all the functions needed for the assembler to work. At first we had a string variable that stores each line of machine code for every loop, but that clearly wouldn’t and didn’t work so we resorted to using a struct which holds information about each instruction line, then each instruction line is stored in an instruction list vector. This makes access to instruction a lot easier and more organised. The symbol table was quite simple, we just used an unordered map to store the label as the key and the address as it’s value.

The next problem we had was loading in and parsing the assembly code, because when it was read into the vector, it couldn’t be parsed until the spaces and comments were removed, so we created a few helper functions that remove comments and spaces from all lines and print back the filtered assembly instructions that are ready to be parsed.

The third was problematic thing was the method of parsing/extracting the labels, opcodes, operands and variables. We were going through the instruction string and extracting characters from specific indexes, which obviously wouldn’t work for different instruction lines, so then we researched a bit and found find() function that find’s a specific character in a string and returns the position of that character relative to that string. So we used this to find the position of the colon of a label, and extracted the characters from index 0 to that position, and would then erase it from the line. Now the line starts with the opcode, and because the opcode is always 3 characters, it does the same thing and extracts from index 0 to index 2, then erases it, and so on.

Now that we had instruction info for each instruction line stores in the instruction list vector, and the symbol table filled with labels and addresses, we wrote a function that writes the machine code in a vector of strings, referring back to that instruction list for opcodes and operand addresses. After all of this, machine code is produced and written to an output.txt file.

***Conclusion:***

To be honest, we collectively feel that we have benefitted from working on this assignment. Our knowledge of C++ and how memory and hardware work has increased and I personally enjoyed this assignment very much.