The assembler is a c code file that takes in files filled with an assembly language program and converts it into machine code. We first loaded in a file of assembly language, and began reading it into a buffer. We split it up into a string based on where the whitespace was and used that to determine exactly what step to do for a specific operation that was being read. This step also covered error checking by making sure it was able to split each string regardless of what kind of white space was being used to separate it. This means the assembly language could have used only spaces, only tabs, or a combination of the two, and it would still be able to figure out each string. It also read through the file to find if any comments existed and removed them in order to only deal with assembly language that contains instructions.

We used a series of if statements to compare the operations being read in, to known operations that are allowed to be conducted. If the assembler was passed in an unknown operation it would result in an error stating that it was an unknown opcode being read. The assembler also keeps track of where labels occur and stores where the locations of jumps in order to traverse through the assembly language correctly. Once the opcode was determined we were left with only registers, labels, offsets or immediate values in each line. We used bitshifting to get each opcode, register, label, offset, or immediate value into the correct location of the 32 bit word that is output as machine code.

We had other error checking throughout the process that included reading labels to verify they were correctly used in the assembly language program. For example, if the label was in an invalid format such as including characters that are not letters in the ascii table. It also checked if there were duplicate labels being used in the assembly language.

Some difficulties we had throughout the process of creating our assembler. When writing the code for the and operation to convert it to machine code, our output was always off by a few digits. For example the first output was supposed to be 8912903 but ours would output 8912906. We were not able to solve this issue before the time the assignment was due.

Another issue we had was we tried to separate each line with the original new line by only getting rid of the spaces and tabs when reading the file. At first we thought this was a good idea for keeping track of label jumps and when to read an opcode after reading registers or a label. We believe this did not work because there were a lot of segmentation faults that had something to do with the Strtok() function but were never able to solve that issue. We fixed it by restarting the process and including the new lines in the whitespace removal.