Final Project Report

Assembler

4/25/2018

**Group Members:**

**Name Role/Tasks**

Trenity Bray (Coder)

Tyler Manifold (Coder/ Group Leader)

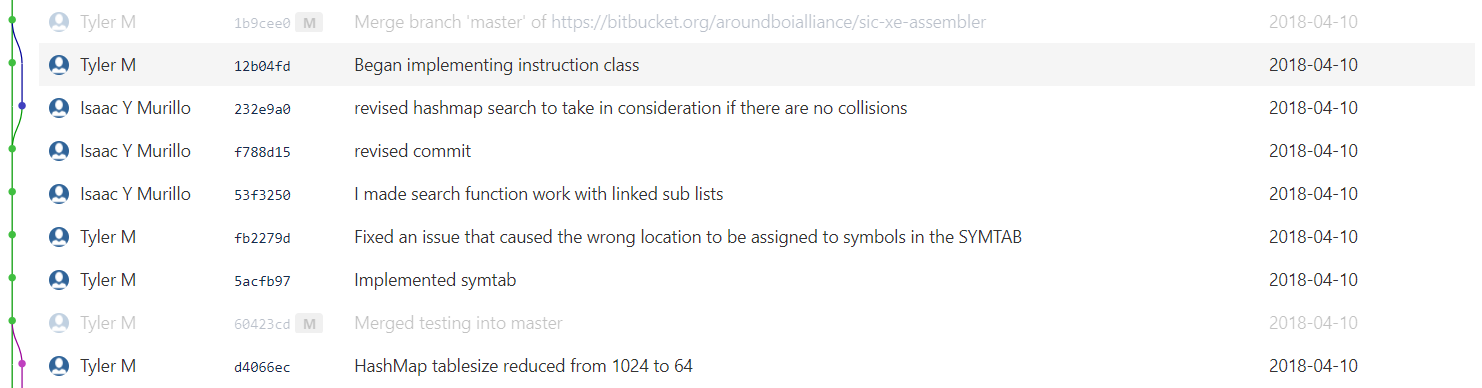
Isaac Murillo (Coder/ Reports)

**Group Organization:**

**Bit Bucket:** We organized ourselves quickly to begin the project. Tyler Manifold became group leader as he had a starting idea on how to initiate the project. We used bit bucket to organize and structure our code, as well as using it to structure our team. With bit bucket our group lead could assign tasks needed to be done or issues that pop up to a team mate and assign the priority state of the given assignment. For example, if a certain class or new file that needs be created would appear on bit bucket with corresponding priority, and who it was assigned to.

 Example:

Bit bucket also allowed the whole group to see who committed code and on what day. Helping us keep track of work ethic and correlate our attention if someone needs help with their assigned part (we also had a testing branch for testing code with the rest of the program).

Example: 

**Scheduling:** Our group schedule consisted mostly of small meetings after system programing class. With small updates on different parts of the project and any Q&A that needed to be done. Even though we implemented it late into the project we did end up taking advantage of hangouts (“Group text app”) for ease of communication between the group.

**Project/Program Layout:**

**Project Vision:** The current and final project layout implementation follows the original vision which we planned in the beginning of this project. Which was following simplicity and using object-oriented programing with different file format. Which meant separating major concept operations into its own respective class (i.e. tokenizer, HashMap, etc.), and would be its own file. This made splitting up the work easier and testing simpler as we can debug them separately and review them individually. If anything ends up going wrong in the code, we wouldn’t have to scroll though lines of code every time. We also kept in mind that these classes must work with each other. Making us think on how this would affect the program when producing this program.

**Classes:** When creating classes, we always made them their own respective files (i.e. Header and .cpp files). The advantage of making our project object oriented, and simply makes the summary/explanation simple and easy as well.

**Tokenizer:** Handles the parsing of the content in the file which it later turns into tokens.

**Assembler:** The assembler (2 pass assembler) controls everything. Pretty much everything is all traced back to the assembler. It would create OPTAB, SYMTAB, etc. Pretty much everything it needed to function with the inclusion of also reading in the file while allowing the tokenizer to take care of making tokens based on the inputs of the file contents. The assembler was also in charge beginning calculations on address instructions, and relative address.

**Pass 1:** First pass will insert unknown symbols and their memory locations into the SYMTAB for use in pass 2

**Pass 2:** The second pass of the assembler uses the now populated SYMTAB to begin generating object code

**HashMap:** HashMap was a just a general storage medium in which we would use for creating a SYMTAB, OPTAB, and DTAB. We knew were going to need a data structure in which we could be flexible and simple enough to use inheritance on (e.g. OPTAB).

**Instruction:** This was an important class as the instruction class oversaw building the final instruction.

**Issues:**

**HashMap (Isaac Murillo):** HashMap was a tricky one as it was one of our building block objects. How the HashMap functioned would define how we implemented our program. Thus, with that came many problems, in the beginning (Isaac got it done quickly), but it followed a classic hashing function in which it used an integer for a key, and a string for a value. We needed those two to be flipped, so the problem ended up being how would we hash a string.

**Solution:** The solution that ended up being used were using ASCII values and units of powers to give weight to the characters. As it would make hashing easier and dealing with collisions a lot simpler to solve.

**Tokenizer (Tyler Manifold):**

**Problem 1:** Tyler ran into problems with properly dealing with comments and white space characters (space, tab, newline, carriage return) when reading in the file. This would have been an issue for any comments that would slip through could have destroyed our project.

**Solution:** His answer to this problem was to write a function to strip them out on a given string,

**Problem 2:** Knowing which column a token is in.

**Solution:** The solution to this problem was by redesigning tokenizer to add each token from a line to an array and return it. If # of tokens < 3, then first token is mnemonic opcode. Otherwise first token is label.

**Instructions (Tyler Manifold):**

**Problem 1:** One of the issues in this regard was determining PC/Base relativity.

**Solution:** Mostly bugs and calculating errors for figuring out if it was PC or Base relative instruction and using switch statements to for each case.

**Problem 2:** Additionally, keeping correct count of locations based on number of bytes used for an instruction, indicated by various factors such as instruction format and assembler directives (RESW, RESB, etc.).