Myson Burch

John Carillo

Dr. Xukai Zou

CS 300

2 May 2017

Final Project Report

The purpose of this project was to create an assembler for the hypothetical computer system using the SIC/XE architecture which is seen in *System Software: An Introduction to Systems Programming*, by Leland Beck. The team for the project was made up of two members: Myson Burch and John Carillo.

At the Mid-Term Report, we were still in the abstract designing phase. We had originally been leaning towards C++ with Python as close second. We had decided to go with a modular design and believed C++ would help us with the modular design as it a language that both of us were fairly comfortable and familiar with. Python was a close second because of its simplified syntax seemed beneficial to a fast pace of development. After discussion and debate over the programming language we decided that it would be in our best interest to go with Java. We realized that with Java we would be able to keep familiarity as we both have worked with it before, while also helping us make our program more dynamic. The ability to easily pass arrays and array lists through function calls was an appealing capability that Java haves. In addition, the object-oriented approach that Java takes seemed to be beneficial to the modular approach we were taking. Overall, we decided after much discussion and debate to go with Java over C++ and Python.

When we started this project, we decided to use a modular design was the best way to go about building the assembler. In this way, we were able to take each action and split them as their own modular. We used the file examples provided at the beginning of the semester to break the project into each modular. The first milestone to reach was getting the basic.txt file to work properly. After discussion, we decided to go with a two-pass assembler as outlined in chapter two of the book previously mentioned. Each pass was considered a modular where the responsibilities were split between Myson and John. John was responsible for the first pass and Myson was responsible for the second pass.

In the first pass, John split the modular into multiple ones where each one built off itself. The first was to be able to open and parse the basic.txt file. John was struggling with determining how regularly define each column in the text file. John discovered that each column was delineated by a ‘\t’ character. Utilizing the object String’s split() method, John was able put the columns in an array that was easily parsed to determine each lines label, operand, operator, and comment. Next John created an op object where the mnemonic, format length, and opcode for each operand would be stored. Once the op object was created, John created an opTable object where the operands could be stored and called upon whenever an SIC/XE Operand is called upon in the code. Once the opTable was established, John created a location object made to act as the locator pointer in an assembler. At each line in the code the location object is updated to the proper spot. The last thing in the first past John needed to create was a symTab object, which was a table that consisted of the Symbol object created. The symTab was updated every time an instance of a symbol occurred during the first pass. Once all the different components of the first pass were created, John created the parse object which put together all the parts into a functional first pass for the basic.txt. After completing the first pass, John turned the code over to Myson to work on the second pass. While Myson began the task of object code generation and the second pass, John worked on macro expansion as outlined in macros.txt. The logic behind the macro expansion is to run it through a two-pass system similar to the two-pass assembler. The first pass parsed each line of code, looking for the creation of macros and making sure that every called macro is established. When the creation of a macro is discovered, it is stored as a macro object and then placed in the macTab object. The second pass expands the macro at each macro call. The expansion happens before the two-pass assembler begins its run. This was designed so as the assembler, which was still being constructed, could operate independently from the macros. The assembler accepts the expanded file and not the original file.

\*\*\*\*\*\***MYSON’S STUFF GO HERE\*\*\*\*\***

Overall, we were able to create an assembler that correctly creates the text record for the basic.txt file given at the beginning of the semester. The assembler is also able to correctly expand macros, however the text record for the program is not accurate as the macros.txt file contained literals as well. We utilized github to post our code and record our progress. The git hub link is: <https://github.com/jcar195/assembler> You can see the branch from where we finished the two-pass assembler for basic.txt and where the macros expansion. Between all the files, github stated that there were 1872 lines of code. The amount of lines were attributed:

* charTab – 102
* character – 12
* firstpassoutput – 73
* location – 51
* macro – 39
* macroProcess – 193
* macroTab – 48
* main – 40
* objcodegen – 610
* op – 14
* opTab – 100
* parse – 100
* secondparse – 428
* symTab – 50
* symbol – 12

While we were unable to complete functionality for the other files, we were pleased with our work and glad we were able to get the two pass assembler working for basic.txt.