Evan Cropper – Class Project Report

The class project involves writing an assembler from scratch in a language of my choosing (Python) for the hypothetical language and architecture SIC/XE. The project will ensure that I understand the theory behind low-level computer language and will help me to develop extensive programming and development skills. Since my team is just me, I will be working on the project every evening Monday, Wednesday, and Friday from 6:00 to 8:00. This will give me enough time to use efficiently; more than 8 weeks and 48 hours of work time. I will be using the book System Software: An Introduction to Systems Programming by Leland L. Beck alongside the given PowerPoint to help me in making sure that my assembler works correctly.

I will be building the class project in Python. Python provides a high-level abstraction which makes it easier to focus on the logic of assembling instructions without getting bogged down in low-level details. Its syntax and nature allow for rapid prototyping and development compared to lower-level languages like C. It also offers extensive libraries and tools that can be utilized for parsing and lexical analysis. Python is incredibly good for string operations. It allows for easy integration with other languages and systems if needed, so if I ever decide to go back to this project, I can easily use it for an actual piece of hardware. Python code is also one of the more readable languages, and it is easy to understand what is going on just by looking at the code, improving long term sustainability of the project. The only possible tradeoff I can think of by choosing Python is that the language is not the lightest nor performant of languages, so I am trading off performance for readability and ease of programming.

The program will be developed in a straightforward manner; there will be several classes involved which will not only represent the assembler itself, but aspects which are critical for the operation of the assembler. To get anything running, I will have to initialize the critical tables. `OPCODE\_TABLE` will be defined as a constant map of instruction mnemonics to opcodes, `SYMBOL\_TABLE` will be filled in during runtime to be a map of symbol names to addresses. `LITERAL\_TABLE` will also be filled in during runtime to store literals. `BLOCK\_TABLE` will be evaluated to a map of block names to block numbers and lengths. The `instruction\_stream` will be a vector of instruction objects resembling the assembled program. This is effectively the raw output of the program and should be able to be transferred to a linker/loader for further processing.

I will give an overview of some key functions in the program. `assemble(filename)` will effectively be the main function of this program which will then trickle down to trigger all the other functions in this program. It takes in a .txt file as input. `pass\_one()` and `pass\_two()` represent the two passes of the assembler which will accomplish the tasks appropriate for the assembler. The assembler class includes a `generate\_object\_code()` function which will make heavy use of the encode\_instruction(instruction)` function, which will convert an instruction to binary object code and the `resolve\_symbol(symbol\_name)` function, which will resolve a symbol to its address. `output\_results(object\_code, listing)` will generate output files which will be essential for sending the output to a hypothetical linker/loader and for debugging.

Unit tests will be created for several key functions, especially `encode\_instruction()` and `resolve\_symbol()`, since they will be ran several times throughout the program and are especially important for the program to run correctly. Writing tests for these functions will make my life easier when debugging issues in the program.

The first functions which should be developed are these key functions. They should work independently from the rest of the program, and they should work flawlessly each time no matter the circumstances. I’ll start by defining the opcode table, defining some instructions manually, then working on the `evaluate\_expression()` functionality for the first step of development.

