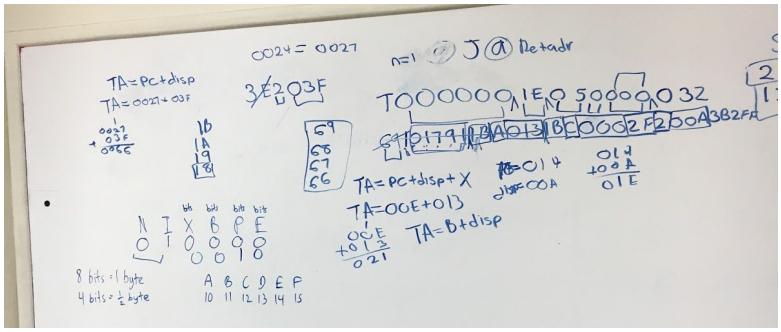
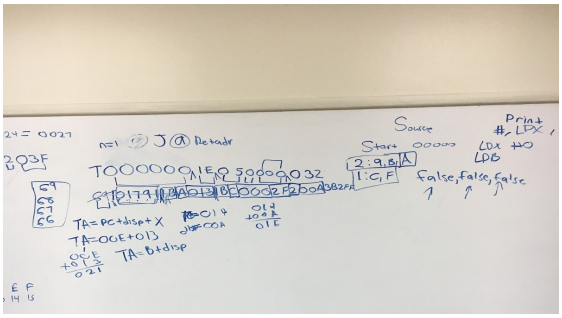
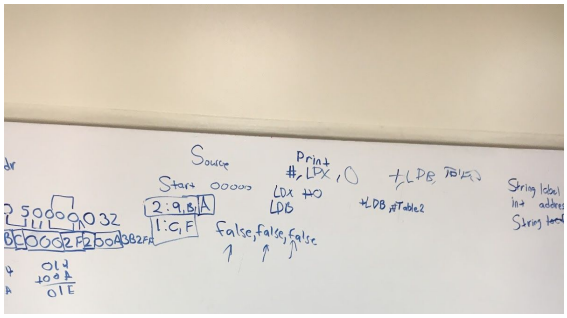
Class: CS 530, Spring 2019
Project: Assignment #2
Disassembler Program for XE variant of the SIC/XE family
Names: Andrew Botros, 819505044, cssc0457
Jan Domingo, 820092657, cssc0463
Anthony Huynh, 819367270, cssc0458
Filename: Software Design Document
/**/

Program/Project Plan:

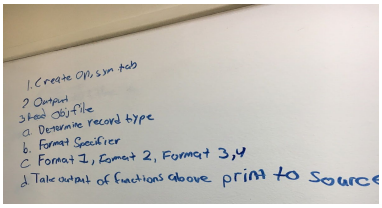
The first time we met up we discussed all the provided files given to us on Blackboard. We took a look at the "sample.obj" file and reviewed how to read a text record. The text record is broken up by columns and we discussed which columns did what. We made sure we knew how to find the opcode using the given instructions. The first three pictures displayed below shows our work going over the text record and the output after the disassembly:







We decided to implement this program in C using CLion and Code::Blocks while keeping in mind that it is utmost important that it runs on Edoras. The next picture shows how we decided to break the assignment up into sub-parts in order to stay organize. It is a big assignment, but is doable by breaking up the assignment into smaller tasks and goals.



System Specification/Design:
Before starting the program we agreed to try to make main() as small as possible so we would have to use multiple functions and call them. We planned on having the location counter inside of main and it will get updated inside the functions. The Program Counter will also get updated in the beginning of each function. Each function, depending on what format it is, will give us the instruction and operand. These functions include:

- Void MainFileParser()
    1. This function controls the bulk of the program, it stores the symTab and LitTab values into a struct and it controls the parsing of the object file which then calls the format functions when appropriate.
- void format2()
    1. We read in the OpCode that is passed into the function and then we read the next two bytes and those bytes contain the Register Mnemonics which we compare and print the corresponding register such as 1 represents Register X and 2 Register L, so on and so forth.
- void format3()
    1. We grab the opCode and the remaining 4 characters. The first of the 4 remaining characters contains the xbpe bits. When we receive the xbpe bits we have several procedures for each scenario such as indexed and PC relative, or extended formatting with indexing.
    2. We then take the displacement, the remaining 3 of the 4 characters and we calculate the displacement via PC or Base depending on the value of the xbpe bit.
    3. The displacement plus PC, Base, or with the X register provides the Target Address which is found in the SymTab struct. When it finds a match with the address field of a symTab struct Array member it prints the corresponding label.
- void format4()
    1. We take the direct address provided in the last 5 half bytes, or in our code 4 characters and we look this up in the SymTab and print the corresponding label

In order to use these functions we would first need to read in the files sample.sym and sample.obj and store them in a struct so we can use them later. We also needed to create a constant struct of all the instructions and their associated opcode. These structs are as follows:

- struct symTab
    1. Since the SymTab has strict formatting such as 8 character long label names and with a specific number of spaces in between that and the address field we can grab the characters at the same exact spot every line
    2. For every entry of the .sym file we grabbed the label and put into the label field of the symTab and the address on the line into the member address field of our SymTab struct.

3. Putting the values into a struct allows for easy lookup, by knowing the label we know the address and vice versa.

- struct opTab
    1. The opTab gives us the instruction in hex as it appears in Appendix A and has the instruction itself and the format
    2. All these member fields allow us to find out the appropriate
- const struct opTab opCodeTable

Code Review:

- After we got everything working we went back and renamed some variables. We cleaned up the program a bit to make it easier to read and follow. We also added more comments to functions and loops explaining how they work.

Final Testing:

- We moved our test files onto Edoras and used make to compile our program. We made sure it ran the way it ran on our IDE. We created a Makefile for our project and named the executable "xed."

Debrief:

- Our group had three members. We worked best when we all met up and tackled problems together instead of doing each task separately and then merging them. Working towards a common goal helped out a lot because if one member of the team did not understand a concept the other members can help them get caught up before moving forward. This also helped with being able to read and understand the program. Since we wrote the code when we were all together we all knew what was being put down and understood which section of the program did what.

Team Issues:

- We worked on it very early, but as exams and other deadlines came up for other classes it was hard to meet up. Over Spring Break we all moved back home, so there wasn't a way for us to physically meet up. Trying to work on the code solo and then merging our ideas was extremely difficult and we were also studying for the second exam. That led to us pushing back the date as there was no immediate deadline in sight. That is not to say, we only worked on it last minute. Had we been able to devote even more time earlier we should have. However when it came down to it, we worked on it consistently and with intensity.

- Another complication is just the overall issue of trying to juggle multiple ideas. Then deciding whose train of thought we should follow. In some instances we thought one of

our ideas was the best one. So that adds a lot of time, you have to walk your group members through your idea and then decide which one to pursue. Initially we thought we could work on the project solo and merge our ideas. It quickly became evident that would not work. If we weren't sitting by side coding, we would've never completed anything.

Team Highlights:

- Andrew was the one who had the full plan set up. Andrew took charge on how to divide the program up into smaller sections and goals. Andrew understood early on how we could parse an object file and passed this knowledge onto the rest of the group.
- Jan is a master with the debugger. In C, since there is very little room for errors especially when working with character arrays. Jan could set up breakpoints and look at the run time values as they updated throughout the code. Whether it be a stray null terminator, or a pointer error we could figure it out.
- Anthony was involved in the various tests, and testing files we made. He made sure we covered all of our bases and checked for edge cases in our code that we didn't account for or errors that we plainly just missed.