Team Members:

Sierra Morehardt, Stefani Bekker, Ryan Aguiar

Team IDs:

Cssc0419, Cssc0424, Cssc0480

Assignment #2, SIC/XE Disassembler

CS530, Professor Leonard

***Software Design Document***

**1. System Specification**

**Inputs:**

A file, *<filename>.obj*, containing Header, Text, Modification, and End Records

A file, *<filename>.sym* containing the SYMTAB and LITTAB to reference records labels and addresses

**Outputs:**

A file, *<filename>.sic*, containing the source code of the collected inputs

A file, *<filename>.lis,* containing the address listing of the source code

**Error Handling:**

Program disallows inappropriate inputs and syntax

**Design:**

Language:

Program was written in the C++ language

Processing Files:

* OPTAB File: contains and retrieves machine code instructions
* SYMTAB File: contains and retrieves machine code labels
* Main File: file that combines all others in order to populate the .sic and .lis files

**2. Software Design**

In addition to the requirements requested through the prompt, tentative planning and design was implemented in order to complete our program. Our team ran meetings, along with independent work and virtual collaboration through the duration of our assignment. A majority of our collaborative effort focused on how to appropriately deal with addressing formats, the calculation of the opcode, nixbpe status, collecting and receiving appropriate labels and information from the SYM, LIT, and OP tables, error handling, special use cases, for example: program and location counter, handling assembler directives (ex: EQU), and appropriately formatting and populating our output files. As we progress through this document you will see the steps taken by our team over the duration of this project. Included will be screenshots of the program files and execution, as well as diagrams and models used in our creative process.

At the start of our project, our first goal was to anticipate the sum of work that would be sent our way. As college students and not peer employees, it was evident to us that we would be unlikely to work collectively as much as we would prefer. To tackle this dilemma, we learned to use Microsoft Visual Studio in a collaborative way. The program offers a live virtual session where each team member was able to edit and run the program on one window of Microsoft VS.

Another effort taken to ensure success on our project was language selection : C++. It was to our knowledge that the included libraries in C++ would provide benefits while completing our project. Some of these libraries include: <algorithm>, <cstring>, <exception>, <fstream>, <list>, <stdlib>, and many others. In addition to the many useful libraries, we believed that the C++ language was one that each team member could excel in due to prior experience.

In regard to the implementation of our program, we will discuss the steps that got us to our resulting source code. First, we will discuss the process taken to handle erroneous input. Our team took the approach of first checking to make sure the correct file format was inputted. Per the README, the correct command line input was the format:

‘./xed <filename>.obj’

To ensure correct entry, we ran several checks:

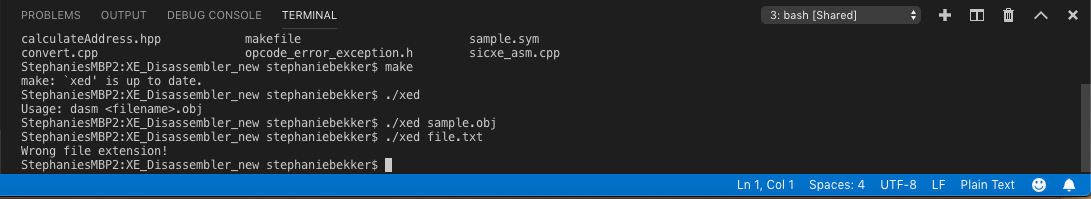
/\* Error Checking \*/

if(inFile.substr(extPosition+1) != "obj") {

cout << "Wrong file extension!" << endl;

exit(1); }

This process along with checking arguments allowed us to ensure a ‘.obj’ file was submitted for processing. In the event that the wrong file was entered, we see the result as follows:



In the above example, when the user entered ‘./xed file.txt’ the program handled the error by indicating that the file extension was incorrect. Now that we have acknowledged that an object file has been submitted, we must ensure that the file indeed contains the records we are looking for. Each record begins with an identifying letter: ‘M’ for modification, ‘T’ for text, ‘H’ for header, and ‘E’ for end. Throughout our code these checks are made, and if the file does not follow format the program terminate without result. These checks are formatted:

‘if(modLine[0] != 'M')

continue;’

Each letter is checked per different record.

When referencing use cases, there are few special cases. One of which is the use of assembler directives, specifically the directive ‘EQU’. To handle this case, each text record was checked to see if the directive appeared. This use case was checked as follows:

‘/\* Load Text Record for Finding EQU \*/

while(getline(objFile,equLine)) {

if(equLine[0] != 'T')

continue;

testEQU = testEQU + equLine.substr(9);}’

This directive assigns absolute or relocatable values to symbols changing the handling of these records.

Now we will discuss functionality, and the direction our team took in completing the necessary requirements. One of the necessary functions when disassembling the object file is determining the addressing format of each operation. Each operation or ‘Opcode’ uses a format specific to its instruction. These formats are labeled as: 0, 1, 2, 3, and 4.When determining formats we must check the opcode as well as the nixbpe flag bits. Opcodes determine the format per the Optab. However, some opcodes have multiple possible formats. If an opcode has multiple formats, we must consult the ‘e’ flag bit to determine which format(3 or 4) is necessary. Our code handled each case per the following snippet:

‘if((opData.first).compare("") != 0) {

isInstruction = true; //Parse instruction

operandMnemonic = opData.first;

format = opData.second;

/\* Format One \*/

if(format.compare("1") == 0)

operand = "";

/\* Format Two \*/

else if(format.compare("2") == 0) {

string firstRegNum = textRecord.substr(counter + 2, 1);

if(secondRegNum.compare("0") == 0)

operand = thisSymtab.getReg(firstRegNum);

else

operand = thisSymtab.getReg(firstRegNum) + "," + thisSymtab.getReg(secondRegNum);

if(operand.compare("") == 0 || operand.compare(",") == 0)

isInstruction = false;}

else {

//Format 3

string niFlags = hexToBin(opCode.substr(1,1));

string xbpeFlags = hexToBin(textRecord.substr(counter+2,1));

pair<string,string> symName("","");

if(niFlags[2] == '1' && niFlags[3] == '1') { //Simple

if(xbpeFlags[0] == '1')

xFlag = true;}

else if(niFlags[2] == '0' && niFlags[3] == '1') //Immediate

iFlag = true;

else if(niFlags[2] == '1' && niFlags[3] == '0') //Indirect

nFlag = true;

if(xbpeFlags[1] == '0' && xbpeFlags[2] == '0' && xbpeFlags[3] == '1') { //e=1

tempAdr = textRecord.substr(counter+3,5);

eFlag = true;}

else if(xbpeFlags[1] == '0' && xbpeFlags[2] == '1' && xbpeFlags[3] == '0') { //p=1

tempAdr = textRecord.substr(counter+3,3);

tempAdr = incrementAddress(loc,tempAdr,3);}

else if(xbpeFlags[1] == '1' && xbpeFlags[2] == '0' && xbpeFlags[3] == '0') { empAdr = textRecord.substr(counter+3,3);}

else if(xbpeFlags[1] == '0' && xbpeFlags[2] == '0' && xbpeFlags[3] == '0'){ //direct

tempAdr = textRecord.substr(counter+3,3);}

else

isInstruction = false;

//Look for symbol name

symName = thisSymtab.getLit(tempAdr); //operand = literal

if((symName.first).compare("") == 0) { //operand = symbol

symName.first = thisSymtab.getRelSym(tempAdr);

if((symName.first).compare("") == 0) //operand = constant

symName.first = tempAdr;}

.

.

.

.

.

if(format.size() != 1 && eFlag)

//format 3/4, then 4

format = format.substr(2,1);

else

format = format.substr(0,1); //format 1,2

//Increment location for next parsing

loc = incrementAddress(loc,format,0);

counter += stringToInt(format)\*2;

if(operandMnemonic.compare("RSUB") == 0)

operand = "";

operand = remove\_spaces(operand);

if(literal != ""){

operand = literal;}

To clarify the snippet above, the instructions have already had their format possibilities determined by consulting the Optab. Format 1 contains only the op code, so no additional steps are needed to disassemble its data. Format 2 is determined by opcodes that are followed by the letter ‘r’. When encountering a format two instruction, the next steps taken are converting and storing the bytes following the opcode. There are two operands ‘r1’ and r2’ that must be accounted for. These operands will later be converted into binary and their resulting addresses calculated and printed onto the resulting files. Format 3 and 4 are handled simultaneous as all format 3 instructions can be used as a format 4 instruction when instructed by the opcode. The opcode is determined to be format 4 when it is preceded with ‘+’. This enables the machine to determine that the ‘e’ bit must be flagged, and that we will be using format 4 instead of format 3. This change in format determines how the program will calculate the address. In Format 4, the bits following the nixbpe flag bits are the complete target address, so the program reads the aforementioned bits and the information is stored. In the case of format 3, the displacement of the target address must be calculated while considering the addressing mode, which itself is determined by the nixbpe bits. The above snippet for format 3 checks these bits and in turn stores new data declaring the addressing mode. From there, our program uses this newly stored data when calculating the addresses it will use to populate our resulting files.

From this point we have disassembled and stored all our data in the form of arrays and vectors. Our team felt it essential that we format our data using these structures. With these structures, we are able to manipulate and format our gathered data in a fluid an efficient manner. We created functions using ‘asm’ declarations as we want our resulting files to be formatted as assembly language source code. Example:

‘void sicxe\_asm::print\_list\_file()‘

To start this process the function has a function ‘get\_to\_start()’ that ensures that the collected data is accurate. If that is so, the first value we will find is the START assembler directive. This is handled by:

‘void sicxe\_asm::get\_to\_start(file\_parser parser) {

while(to\_uppercase(parser.get\_token(current\_line - 1, 1)) != "START") {

if (parser.get\_token(current\_line - 1, 1) != " ")

throw symtab\_exception("First Opcode must be start");’

Once the START assembler directive is confirmed, the program begins to enter its final stages of production. Once in the correct format our files are populated with the following results:

.

.

.

.

.

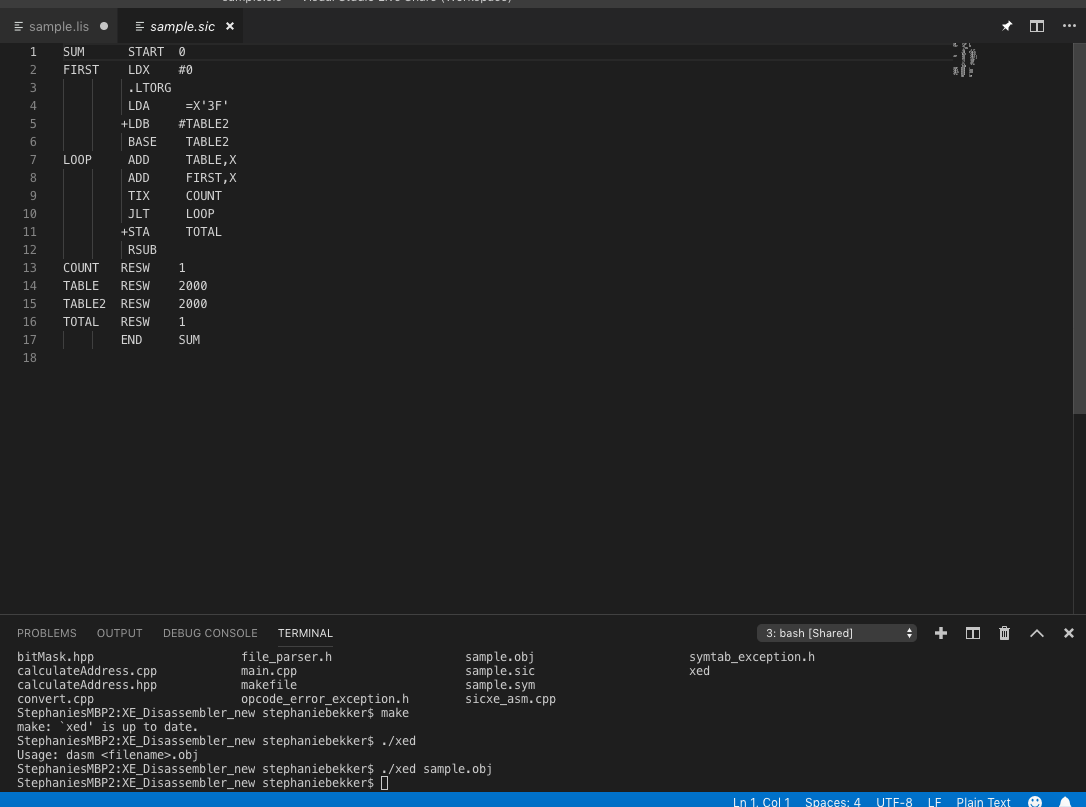
.

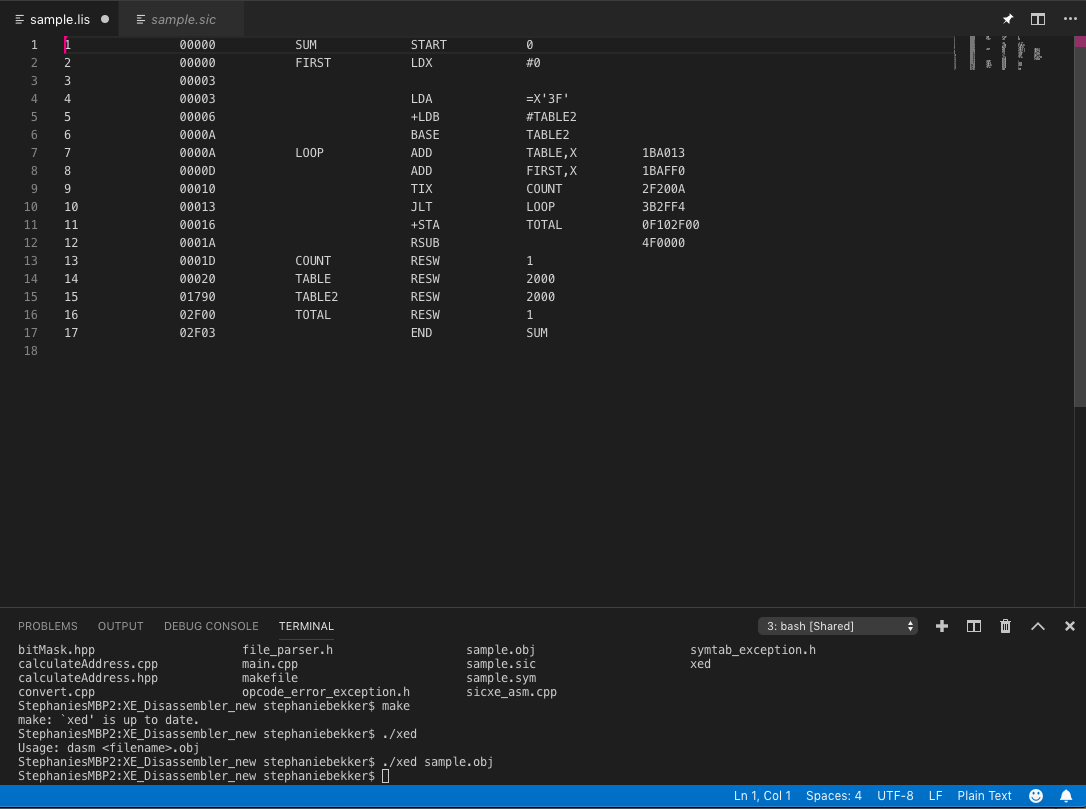
.

.

.

.





**3. Conclusions**

Our team learned a lot from our project in regards to the program as well as how to work in a team environment. Overall our team would consider our results a success, and we believe we could progress as a team if given another task. We also believe that there are areas in which we could use some refining. One of those being project organization; on occasion we had some repeated or superfluous code due to different techniques taken by different members of the team. Another area that could use some work is pre-coding preparedness. Although we all knew the subject at an intermediate level, there are extra steps we believe we could have taken before putting any code down. We managed to divy the workload well, however we could have had each member more prepared. Lastly, we believed the most difficult problem we faced as a ‘coding’ team is debugging unfinished code. It took us a lot of effort to get to the final stages of debugging as our pieces did not all fall together right away. After many sessions of collaboration we were able to resolve our program back to a team effort and from there we were able to debug with ease. We are proud with our results and are eager to find the team we will build our future career with.