Jacqueline Gronotte, cssc041
Cesar Lopez, cssc0465
Samantha Quiroz, cssc0416
CS 530, Assignment 2
SIC/XE Disassembler

Software Design Document

<u>System Specification</u>

System Inputs:

<filename>.obj File contains the Header Record, Text Records, Modification Records, and End Records

<filename>.sym File contains the SYMTAB

System Outputs:

<filename>.sic Source File

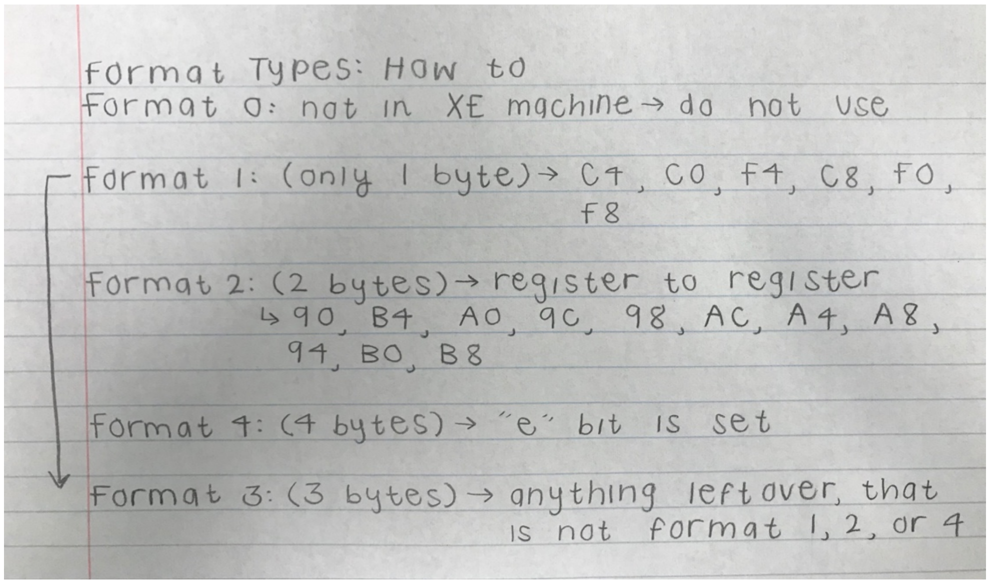
Performance Requirements: Completely disassembles any SIC/XE Machine code.

Design: Code was written in C++.

System Software Design:

Upon beginning this assignment, we discovered the great deal of planning and efficient team work it would take to complete all aspects of our disassembler. Our meetings included writing down numerous “pseudo” codes for determining each format type from the object file, how to deal with NIXBPE bits, and understanding the SYMTAB and LITTAB. Included in this document are diagrams that we used to aid in our thought process and planning for each step of the disassembler. Our team primarily met twice a week throughout the duration of the project, using one day to plan and the second meeting to code.

In our first meeting, we decided we would write our disassembler using C++ and would post our combined code on a shared GitLab project containing all necessary files. We began our collaborative work by deciding how our program would recognize the four format types of an object file and differentiate them from one another. Below is a photograph of our written work to display this discussion.



Format Types: How to
Format 0: not in XE machine → do not use

Format 1: (only 1 byte) → C4, C0, F4, C8, F0, F8

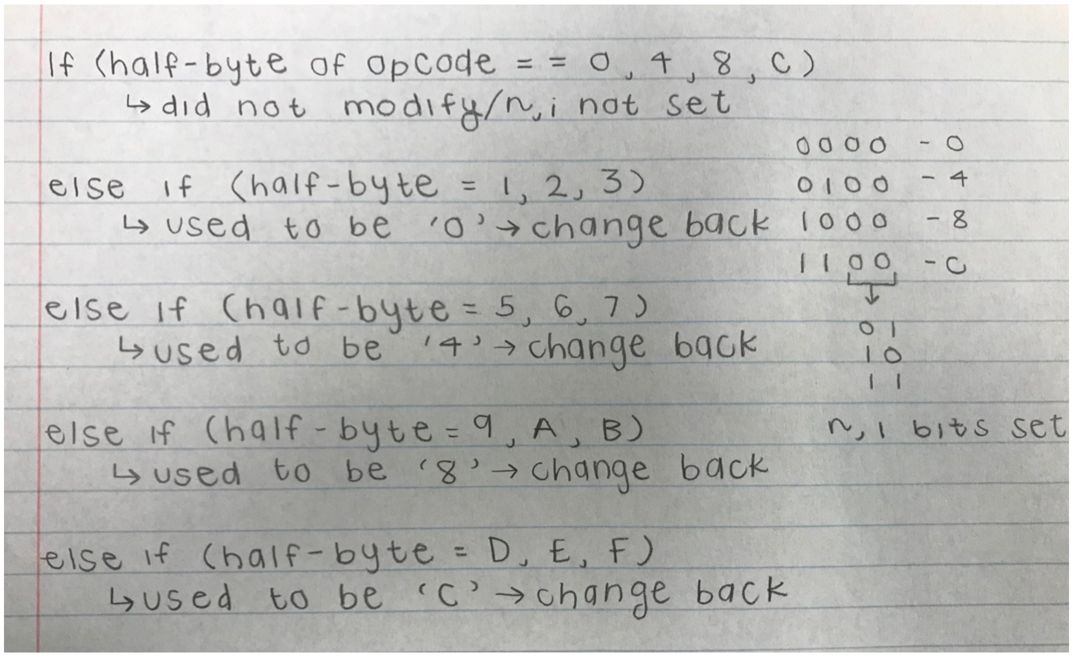
Format 2: (2 bytes) → register to register
↳ 90, B4, A0, 9C, 98, AC, A4, A8, 94, B0, B8

Format 4: (4 bytes) → "e" bit is set

Format 3: (3 bytes) → anything left over, that is not format 1, 2, or 4

Next meeting, we determined the code to check that we are in fact reading in an ".obj" file and include proper error handling if the input file extension is incorrect or not found entirely. We then worked on the Header Record, starting at its first character "H" and were able to easily read in the program label, starting address and the program length; then output the resulting information into one string. Reading in the Text Record from its starting character "T" next, we read in the starting address, length (number of bytes) of object code in hexadecimal format, followed by each of the object codes.

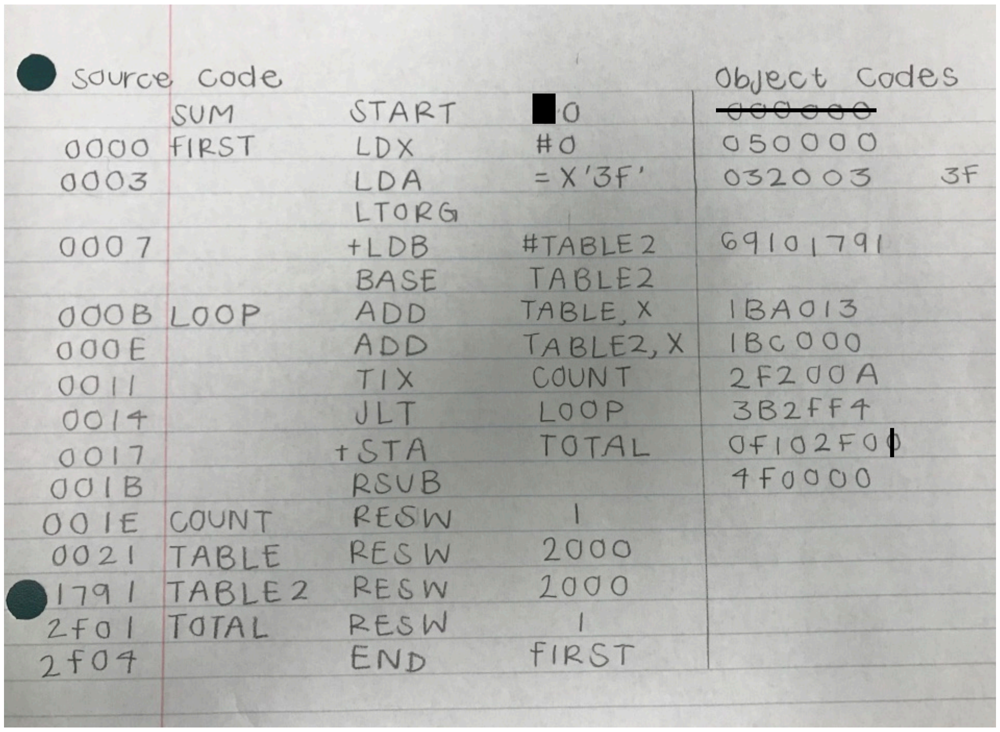
When reading in each individual object code of the Text Record, we had to follow numerous steps to ensure the correct format and retrieve the proper output, including the op value. We checked the first half-byte of the object code to see if its "n" and "i" bits had been modified from "00" to "11" to follow the XE machine formatting. In the photograph below, is how we established if this modification occurred and how we obtained its original opcode corresponding machine instruction.



```
If (half-byte of opcode == 0, 4, 8, C)
    ↳ did not modify/n,i not set
else if (half-byte = 1, 2, 3)
    ↳ used to be '0' → change back
else if (half-byte = 5, 6, 7)
    ↳ used to be '4' → change back
else if (half-byte = 9, A, B)
    ↳ used to be '8' → change back
else if (half-byte = D, E, F)
    ↳ used to be 'C' → change back
```

```
0000  - 0
0100  - 4
1000  - 8
1100  - C
  ↓
 01
 10
 11
n, i bits set
```

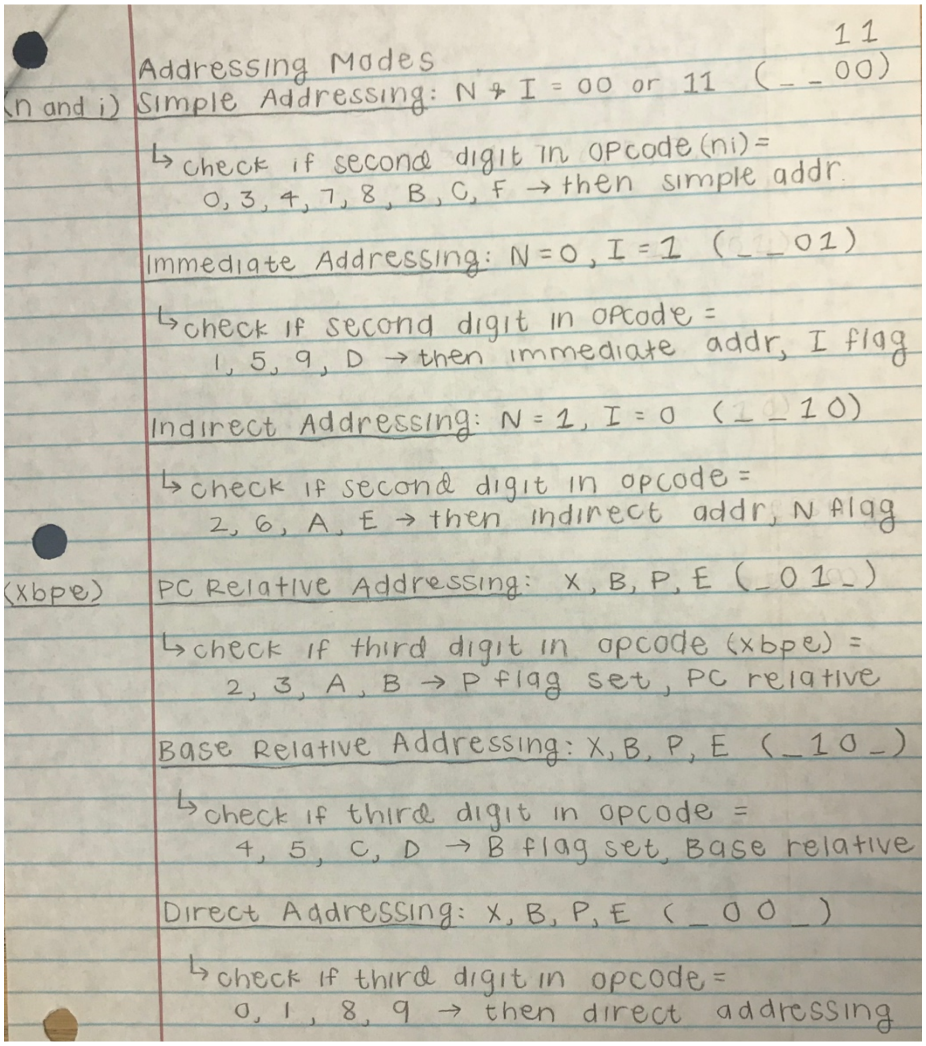
For numerous meetings, we worked on understanding all aspects of the Text Record, including format types, addressing modes, target addresses, as well as maintaining our location counter. We found it would be simplest if we hand-wrote and solved for our object codes from the sample program provided by Professor Leonard, to ensure we were receiving the proper outputs/addresses throughout our code. After understanding the sample source code and object code, we created our own test files to test specific format type outputs not included in the sample (format 1 and format 2). We used this as an aid in creating our ".lis" file to ensure that we had accounted for all machine instruction types and were displaying the correct output accordingly.



Source Code				Object Codes	
	SUM	START	[redacted] 0	~~000000~~	
0000	FIRST	LDX	#0	050000	
0003		LDA	=X'3F'	032003	3F
		LTORG			
0007		+LDB	#TABLE2	69101791	
		BASE	TABLE2		
000B	LOOP	ADD	TABLE, X	1BA013	
000E		ADD	TABLE2, X	1BC000	
0011		TIX	COUNT	2F200A	
0014		JLT	LOOP	3B2FF4	
0017		+STA	TOTAL	0F102F00	
001B		RSUB		4F0000	
001E	COUNT	RESW	1		
0021	TABLE	RESW	2000		
1791	TABLE2	RESW	2000		
2f01	TOTAL	RESW	1		
2f04		END	FIRST		

For our Assembler Directives, we knew that "START" was placed at beginning of the program and "END" at the bottom of the program. In order to output "BASE", we checked if "LDB" was found and print "BASE" on the following line, along with correct formatting provided with it. For "RESW" and "RESB", we checked if the symbol was in the Symbol Table and if it was not used as a label, then we reserved the proper amount of bytes at the end of our program. Towards the last week of our project we focused on reading from the Symbol Table and the Literal Table, as this was the most challenging concept for all of us. For literals, we print "LTORG" to form our literal pool after the last literal is used within the program, where we did not need to use our location counter.

We focused on understanding the addressing modes to appropriately figure out exactly which operands and format we would use in our output. We meticulously worked to understand the addressing modes for each input as this would easily define our displacement and ensure that our operand column had the correct display based on its addressing mode. We found that the Text Record took up the majority of our programming time, as we accounted for each and every case that could appear in a given object code needed for disassembling. Although it was challenging and time consuming, we obtained a stronger understanding of the SIC/XE Machine and how exactly each addressing mode works/does not work with one another. Pictured below, is our "pseudo" code of how we determined the various addressing modes based on the first three digits of a given opcode.



11

Addressing Modes

(n and i) Simple Addressing: N & I = 00 or 11 (_ _ 00)

↳ check if second digit in opcode (ni) =
0, 3, 4, 7, 8, B, C, F → then simple addr.

Immediate Addressing: N = 0, I = 1 (_ _ 01)

↳ check if second digit in opcode =
1, 5, 9, D → then immediate addr, I flag

Indirect Addressing: N = 1, I = 0 (_ _ 10)

↳ check if second digit in opcode =
2, 6, A, E → then indirect addr, N flag

(xbpe) PC Relative Addressing: X, B, P, E (_ 01 _)

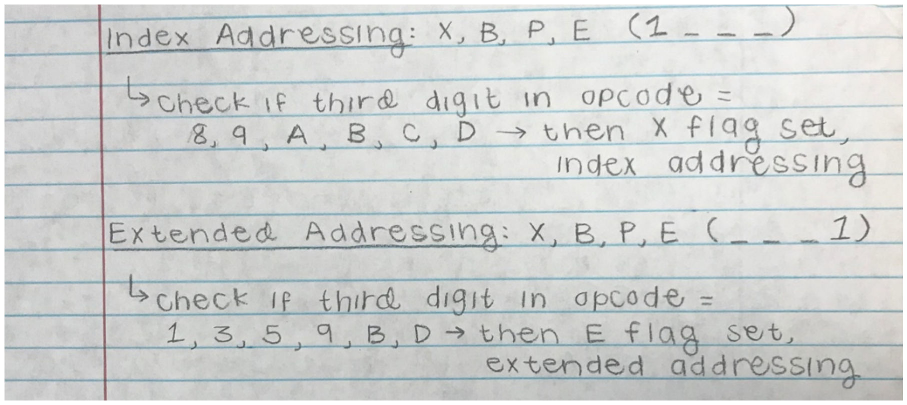
↳ check if third digit in opcode (xbpe) =
2, 3, A, B → P flag set, PC relative

Base Relative Addressing: X, B, P, E (_ 10 _)

↳ check if third digit in opcode =
4, 5, C, D → B flag set, Base relative

Direct Addressing: X, B, P, E (_ 0 0 _)

↳ check if third digit in opcode =
0, 1, 8, 9 → then direct addressing



Index Addressing: X, B, P, E (1 _ _ _)

↳ check if third digit in opcode =
8, 9, A, B, C, D → then X flag set,
index addressing

Extended Addressing: X, B, P, E (_ _ _ 1)

↳ check if third digit in opcode =
1, 3, 5, 9, B, D → then E flag set,
extended addressing

In our Modification Record, "M", we took in the address of where the modification needed to be made, but did not actually modify the file as we programmed a dissembler and did not truly need to. Lastly, our End Record, beginning with the character "E" and simply read in the address of the first executable line of code, which is commonly the starting address.

After each step, we used print statements to ensure that we were retrieving the correct information, including original opcode/modified opcode, format types and addresses. Throughout this project, we realized the importance of setting goals for each meeting/each week leading up to the deadline. We organized our project into "chunks" dedicating meetings to work on a specific aspect of the program (format type algorithms, addressing modes, etc.) and once completed, we moved onto the next. We found simplicity in writing “pseudo” code together and agreeing upon that before placing it into our program. Although we struggled at times with understanding the SYMTAB and using Literal Pools, we referred to the textbook and one another in order to solve each problem we faced. Through this project, we gained stronger work habits and learned how to better prepare for large programming assignments in the future.