Assignment #2 DIS Disassembler for XE computer
CS530, Fall 2020

Team:

Jesus Aviles, cssc3040 , Red ID: 823931557
Naibi Yilidaer, cssc3054, Red ID:820217379
Claire Fiorino, cssc3053, Red ID: 821765822

Overview & Goals:

The goal of this project was to create a working disassembler for the SIC/XE machine.

We approached this project in a way similar to incremental developmental design. We believed this was appropriate because the process of disassembling SIC/XE code happens in stages, so we wanted to make sure we could replicate each stage with successful output before moving on to the next.

Our subgoals were to have working iterations of the project, adding more each time:

Iteration 1: Be able to successfully parse the .sym/.obj files and store data
Iteration 2: Add accurate addressing mode calculations for formats 3 and 4
Iteration 3: Add accurate addressing mode calculations for formats 1 and 2
Iteration 4: Be able to print properly formatted data to .sic/.lis files

Project Description:

Design, develop, test, and deliver a disassembler program for the XE variant of the SIC/XE family of machines.

The goal of this project is to design, develop, test, and deliver a Disasembler program for the SIC/XE machine.

- The project must accept and open an XE object file, alongside its accompanying .sym file.
- Once the program processes all necessary information, it shall generate a source file and .lis file using the same <filename> during input.
- If neither the .obj file or the .sym file exist, the program will exit immediately.

Plan of Action and Milestones:

Week 1:
- Outline Pseudocode

- Brainstorm direction of implementation
- Begin code for reading .obj file and .sym file

Week 2:

- Beginning construction functions and implementing the beginning steps to Disassembling an Object code file.

Week 3:

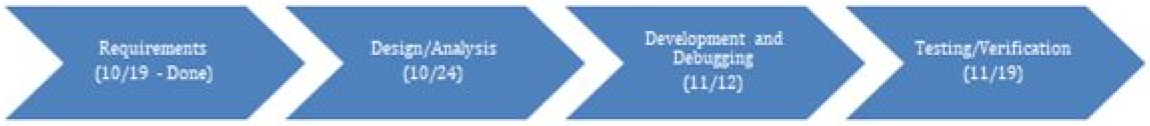
- Begin implementation of logic to calculate various addressing modes and object code.

Week 4:

- Work on successful output of .sic and .lis files to be generated

Week 5:

- Create a valid makefile
- Write a descriptive README
- Revise Final Draft of the Software Design Document
- Final week of testing and debugging before submitting final project.


Requirements (10/19 - Done)
Design/Analysis (10/24)
Development and Debugging (11/12)
Testing/Verification (11/19)

Requirements:

Linux environment or shell emulator required for use of the SIC/XE Disassembler.

Disassembler requirements: Summarized in the description of the project….

- Project must implement a Disassembler for SIC/XE machine .obj file/.sym file
- Project will scan the current working directory for necessary files and begin execution, shutting down immediately if none are found.
- Addresses for each operation, along with the object code, shall be calculated and stored in a data structure for later reference.
- Source file and listing file shall be generated at the end using the information calculated and shall output to their respective files in the same directory.

Required Documents:

- README
- Makefile
- Test files

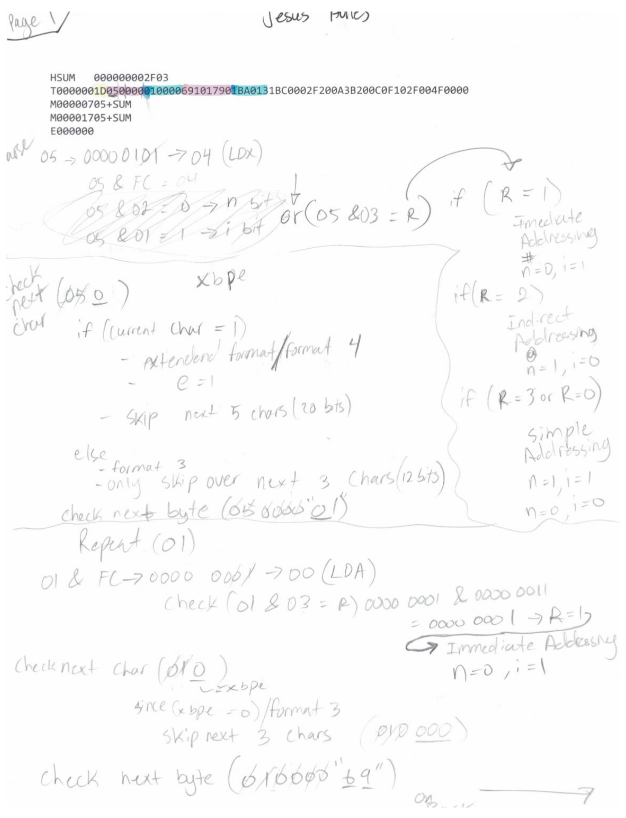
- SDD (Software Design Document) Final Draft

System Design/Specification:

Data structures utilized: 1D arrays of structs

Implementation: 2 pass instructions as outlined in the textbook
- Pass 1 - Defining symbols
- Pass 2 - Assembling the instructions and generating the object program files

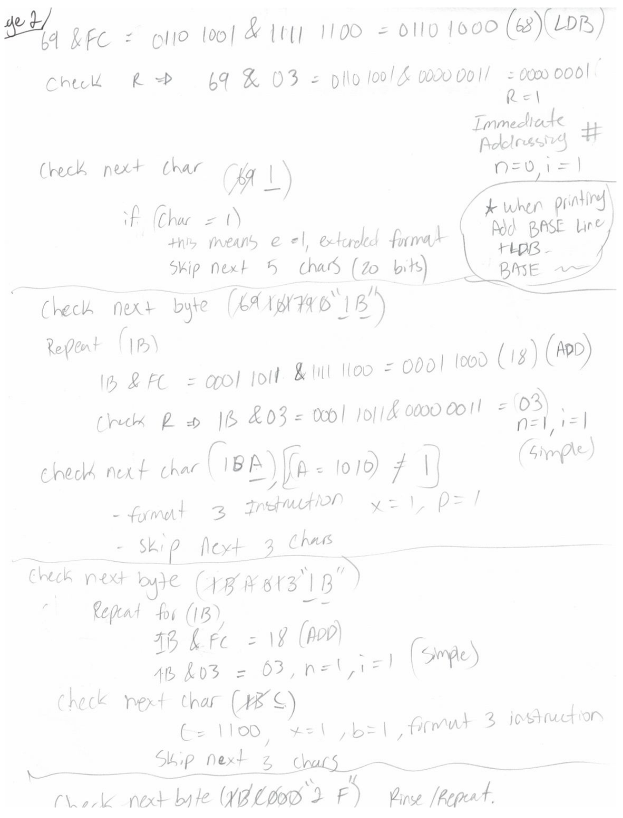
Initial Psuedocode/Brainstorming:



Page 1 Jesus [illegible]

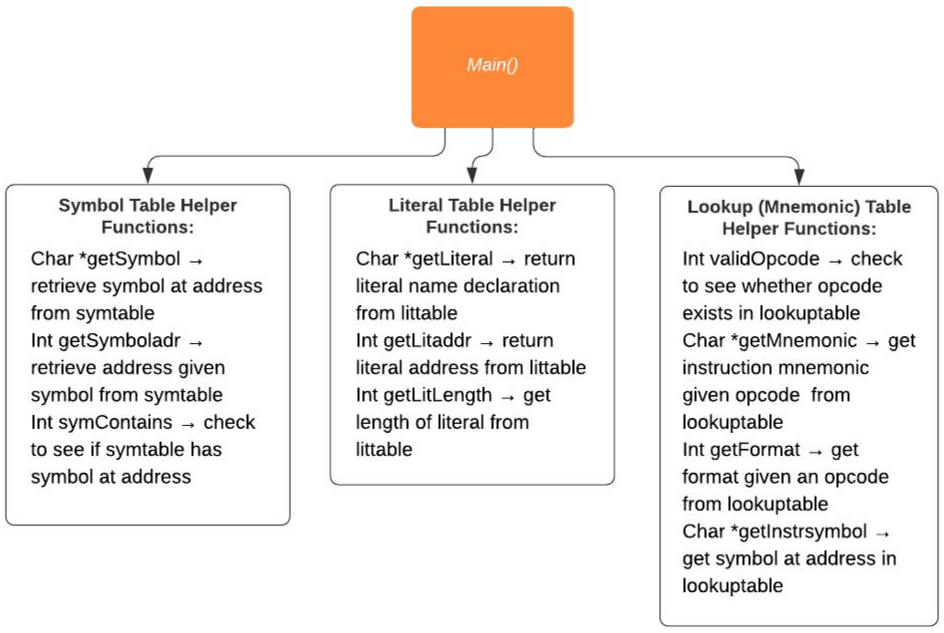
```
HSUM   000000002F03
T0000001D050000010000691017901BA0131BC0002F200A3B200C0F102F004F0000
M00000705+SUM
M00001705+SUM
E000000
```

```
[illegible] 05 -> 0000 0101 -> 04 (LDX)
    05 & FC = 04
    ~~05 & 02 = 0 -> n bit~~
    ~~05 & 01 = 1 -> i bit~~   or (05 & 03 = R)   if (R = 1)
                                                      Immediate Addressing
                                                      #
                                                      n=0, i=1
                                                  if (R = 2)
                                                      Indirect Addressing
                                                      @
                                                      n=1, i=0
                                                  if (R = 3 or R = 0)
                                                      Simple Addressing
                                                      n=1, i=1
                                                      n=0, i=0
check next char (~~05~~ 0)      xbpe
    if (current char = 1)
        - extendend format/format 4
        - e = 1
        - Skip next 5 chars (20 bits)
    else
        - format 3
        - only skip over next 3 Chars (12 bits)
check next byte (~~05 0000~~ "01")
Repeat (01)
01 & FC -> 0000 0001 -> 00 (LDA)
    Check (01 & 03 = R) 0000 0001 & 0000 0011
        = 0000 0001 -> R = 1
        -> Immediate Addressing
           n=0, i=1
Check next char (~~01~~ 0)
                    <- xbpe
    since (xbpe = 0) / format 3
        Skip next 3 chars (~~010~~ 000)
Check next byte (~~010000~~ "69")
On...
```

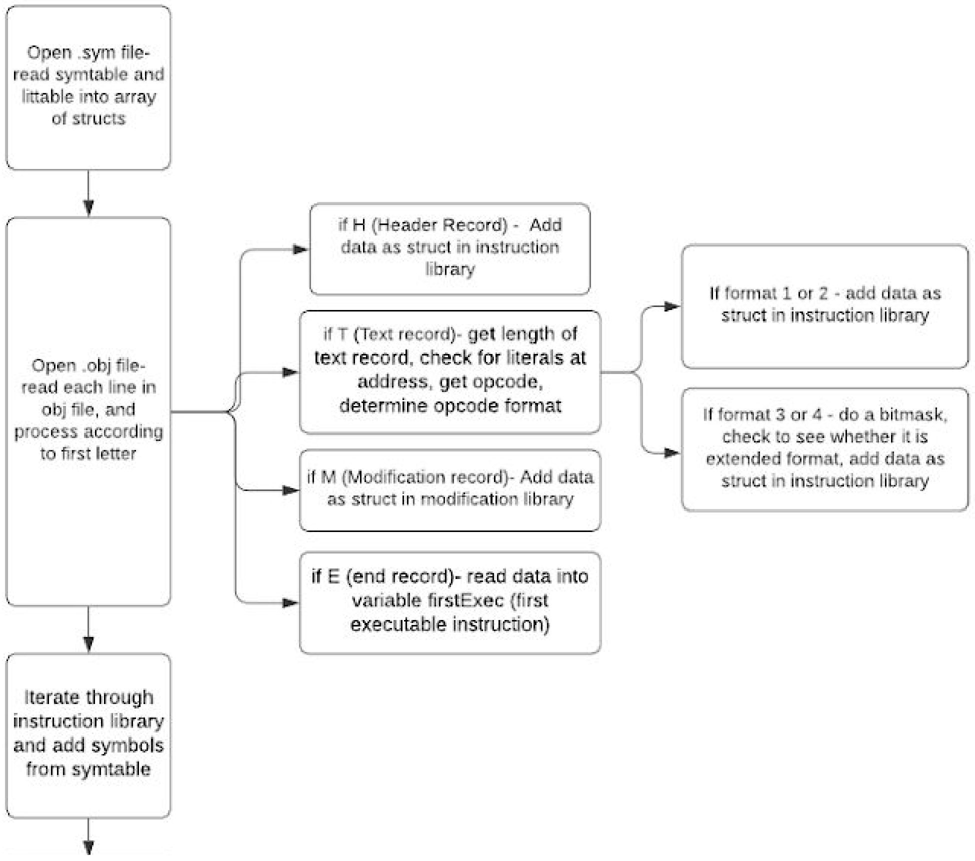


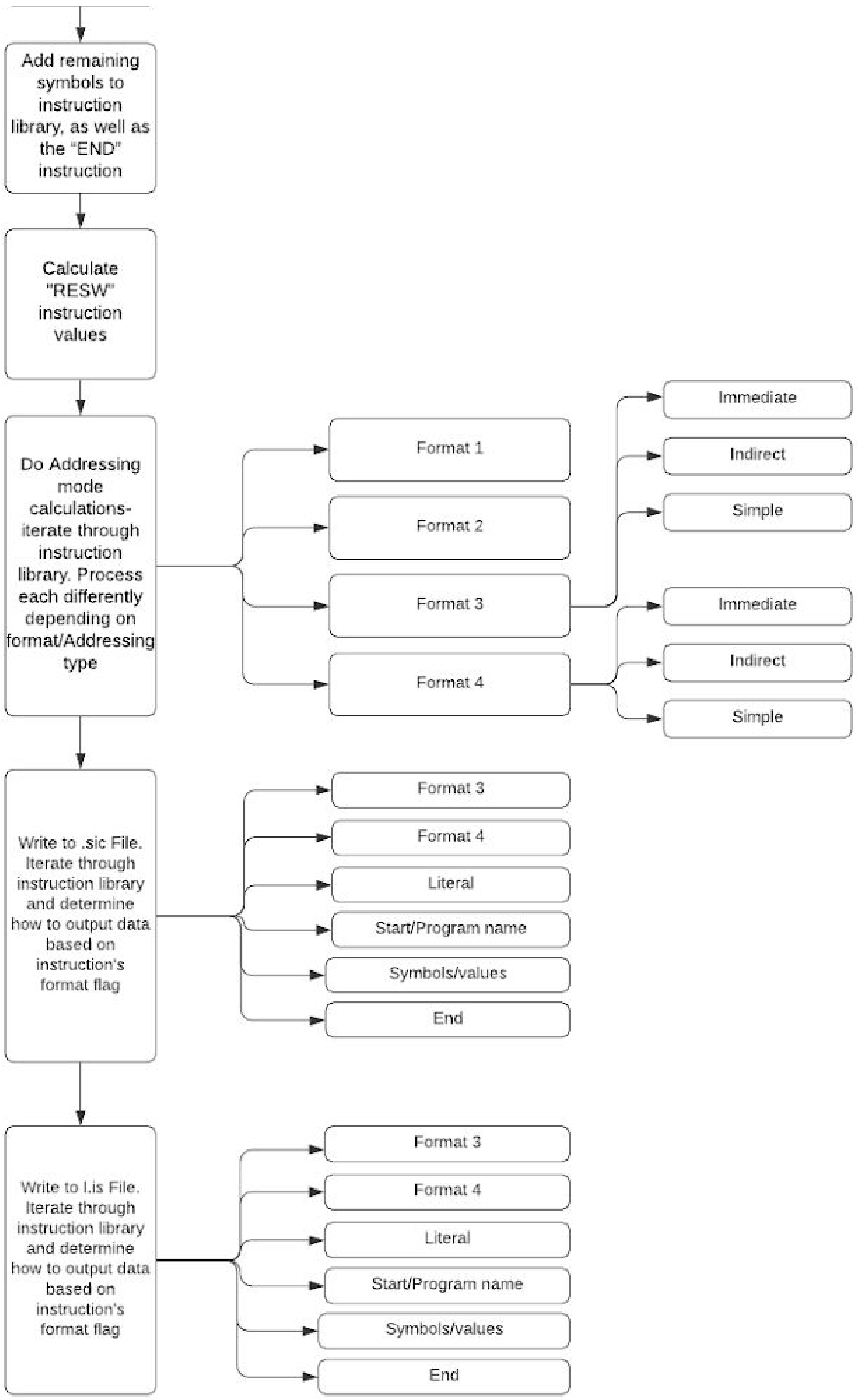
```
Page 2
69 & FC = 0110 1001 & 1111 1100 = 0110 1000 (68) (LDB)
Check R => 69 & 03 = 0110 1001 & 0000 0011 = 0000 0001
                                             R = 1
                                             Immediate Addressing #
                                             n=0, i=1
Check next char (~~69~~ 1)
    if (Char = 1)
        this means e = 1, extended format
        Skip next 5 chars (20 bits)
                                             * when printing
                                             Add BASE Line
                                             +LDB
                                             BASE
Check next byte (~~6910179 0~~ "1B")
Repeat (1B)
    1B & FC = 0001 1011 & 1111 1100 = 0001 1000 (18) (ADD)
    Check R => 1B & 03 = 0001 1011 & 0000 0011 = 03
                                                 n=1, i=1
                                                 (simple)
check next char (1BA), [(A = 1010) ≠ 1]
    - format 3 Instruction  x=1, p=1
    - skip next 3 chars
check next byte (~~1BA013~~ "1B")
    Repeat for (1B)
        1B & FC = 18 (ADD)
        1B & 03 = 03, n=1, i=1 (Simple)
    check next char (~~1B~~ C)
        C = 1100, x=1, b=1, format 3 instruction
        Skip next 3 chars
Check next byte (~~1BC000~~ "2F")   Rinse/Repeat.
```

Function diagram:


Main()
Symbol Table Helper Functions:
Char *getSymbol → retrieve symbol at address from symtable
Int getSymboladr → retrieve address given symbol from symtable
Int symContains → check to see if symtable has symbol at address
Literal Table Helper Functions:
Char *getLiteral → return literal name declaration from littable
Int getLitaddr → return literal address from littable
Int getLitLength → get length of literal from littable
Lookup (Mnemonic) Table Helper Functions:
Int validOpcode → check to see whether opcode exists in lookuptable
Char *getMnemonic → get instruction mnemonic given opcode from lookuptable
Int getFormat → get format given an opcode from lookuptable
Char *getInstrsymbol → get symbol at address in lookuptable

Procedural Diagram for main() (with conditional branches):


Open .sym file- read symtable and littable into array of structs
Open .obj file- read each line in obj file, and process according to first letter
if H (Header Record) - Add data as struct in instruction library
if T (Text record)- get length of text record, check for literals at address, get opcode, determine opcode format
If format 1 or 2 - add data as struct in instruction library
If format 3 or 4 - do a bitmask, check to see whether it is extended format, add data as struct in instruction library
if M (Modification record)- Add data as struct in modification library
if E (end record)- read data into variable firstExec (first executable instruction)
Iterate through instruction library and add symbols from symtable


Add remaining symbols to instruction library, as well as the "END" instruction
Calculate "RESW" instruction values
Do Addressing mode calculations- iterate through instruction library. Process each differently depending on format/Addressing type
Format 1
Format 2
Format 3
Format 4
Immediate
Indirect
Simple
Immediate
Indirect
Simple
Write to .sic File. Iterate through instruction library and determine how to output data based on instruction's format flag
Format 3
Format 4
Literal
Start/Program name
Symbols/values
End
Write to l.is File. Iterate through instruction library and determine how to output data based on instruction's format flag
Format 3
Format 4
Literal
Start/Program name
Symbols/values
End

Development Environment:

Code written in C using the following IDE/environments:

- Visual Studio Code using GCC Compiler (via MinGW)
    - Debugging/Run extensions: C/C++ Intellisense, C/C++ CompileRun, Coderunner
- Edoras- Vi editor

Possible implementation of pair programming and cooperative design.

Run/Test Environment:

- Vi editor
- Edoras - Linux environment
- GCC Compiler

Unit testing involved :

- Testing for edge cases and input validation
    - File input
    - Data structure memory allocation

- Verifying out logic via test cases such as Fixed input w/known outcome

Debugging:

- Debugging in stages will allow us to polish and ensure that our functions work as intended before proceeding which may also help reduce future bugs

- Main Debugging method: Print statements