Nick Krisa
Ayessa Medrano
Oreoluwa Ogundipe
Elliott Watson
Assignment #2
CS 530

Software Design Document

System Specification:

System Inputs:

<*filename*>*.obj* File contains the Header record, Text records, Modification records, and End records

<*filename*>*.sym* File contains the SYMTAB and LITTAB that references labels by addresses

System Outputs:

<*filename*>*.sic* Source File

Level of Error of Processing Required: Program throws an error if instruction is invalid.

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

Design:

Code was be written in C++.

There will be multiple files to handle the processing of the object program:

- Op code file (retrieval of machine code)
- Symbol table file (retrieval of labels from SYMTAB or LITTAB)
- Main file – combines all files to process object program

System Software Design:

This assignment required a great deal of planning to create a SIC/XE disassembler. As a team, we had meetings, almost weekly, to determine the many elements that are necessary to make our disassembler work properly. Our meetings included topics such as determining the handling of different formats, how to calculate opcode, nixbpe bits, the extraction and retrieval of information from the SYMTAB/LITTAB and OPTAB, specific instructions that would need special treatment, the inclusion of certain variables to help our code work (program counter and location counter), assembler directives, and distinguishing an object program file from a random text file. Included in this document are diagrams, we as a team, developed during our meetings that will show how our thinking process went. Note, that these diagrams are from our brainstorming sessions and that not all ideas were correct at the time. However, the ideas continually changed to be correct in the meetings after that. Also note, the diagrams were rewritten for better clarity.

In our first meeting as a group, we first strategized how to handle the .obj file that would be passed into our code. We also decided that we would be writing this disassembler in C++ as we thought some of the programming language's libraries would benefit our code.

To determine that the .obj file is not just a random file with random letters and numbers, we would check the first character of each line. To be a valid object program file, the first character of the first line would be ‘H’ for Header record, the following lines’ first character would be ‘T’ for Text record, ‘M’ would be an acceptable first character in the lines after the Text records as it would imply it was a Modification record, and the last line’s first character would have to be an ‘E’ to signify the End record. If the first characters of each line do not follow this convention, then the code would stop from there.

READING THE LINE AND DETERMINING
TYPE OF RECORD

```
file = ______ .obj
For( no-lines )
    line = getline (file);
    if (line [0] = 'H') {
        :
    }
    else if (line [0] = 'T') {
        :
    }
    else if (line [0] = 'M') {
        :
    }
    else if (line [0] = 'E') {
        :
    }
    else
    // not an object file (random file)
```

After determining that the file was an object program, the symbol table (.sym file) is read also.

In the following diagram, we demonstrate, simply, how to work with the Header record and Text records of the object program.

FLOW FOR CONVERTING RECORDS

(1) READ IN OBJECT FILE
(2) READ IN SYMBOL FILE

IF first letter = 'H'
pick next 6 and store in a variable 'label' ⇒ use forloop
→ 1 2 3 4 5 6

7 8 9 10 11 12 → STARTING ADDRESS
13 14 15 16 17 18 → program length

IF first letter is T
load next 6 → starting address
load next 2 → record length

---

pick first 2 → convert to binary → save as variable
05 → 0000 01(01) → save to variable NI
→ FIND ~~[illegible]~~ OBCODE BASED ON NI

save as variable (opcode) ← 04 ←
→ check OPTAB for what machine instruction variable corresponds to
→ write to file

On the next meeting, we processed the handling of the different formatting (0, 1, 2, 3, 4) that one would encounter in SIC/XE disassembler. We all agreed that format 3 and 4 would be the trickier component we would have to focus on as we had to determine the handling of nixbpe bits.

FORMAT 0 INSTRUCTION (N AND I ARE 0)

if ni = 0
- (1) Take next 2 bytes
- (2) Convert to binary
- (3) Take leftmost bit
  - If it is 1
    - store x-flag = True
  - else
    - store x-flag = False
  - → make MSB (Most significant bit) = 0
    - → call this variable → "address"

if x-flag = True
- (1) Get what is in X register
- (2) ~~Convert~~ Convert that to binary
- (3) Add that to "address"
- (4) Convert back to hex
- (5) Check the SYMTAB for what it is equivalent to
- (6) Print that and ", X"
- (7) On to the next instruction

if x-flag = False
- (1) Convert address to hex
- (2) Check what it is equivalent to
- (3) Print that
- (4) On to the next instruction

One of the ideas we considered including in our program was the use of classes. This class would handle the nixbpe bits of each opcode in the object program and determine the format of the opcode. In the end, we decided not to do classes for our program as we concluded it was not necessarily needed.

FIGURING OUT FORMAT PER INSTRUCTION

NI → 01 – (binary)   0F0000

n flag   i flag

OPCODE → 04 – (hex)

- do more operations to find out how many more half-bytes to fetch

MNEMONIC → LDX – (char)

IF FORMAT 1 → ON TO THE NEXT INSTRUCTION

IF FORMAT 2 → TAKE NEXT BYTE ①

CONVERT BYTE TO BINARY ②

ex. $\underbrace{0000}_{r1}\ \underbrace{0110}_{r2}$

PRINT r1 AND r2 ③

ON TO THE NEXT INSTRUCTION ④

0F0000

→ convert to binary → save as xbpe $\boxed{0000}$

Class

INSTRUCTION TYPE (ni, xbpe)
n-flag
i-flag
x - ✓
b - ✓
p - ✓
e - ✓
int Format (e-flag) returns 3/4

The following meetings included an in-depth look on how opcodes of format 3 and 4 instruction would be handled. We knew that the nixbpe bits would be significant in how our program would write the source file and focused on the different combinations of the bits that could happen. We worked through example object code to determine the many possibilities that occur that would how the source file's contents.

^050000^ 032003 LDX #0

convert to binary

0101 0000

ni xbpe

→ immediate → #

if (b = 0 and P = 1)

if (x = 0) TA = disp + PC

else { TA = disp + PC + X

read in the next 3 characters

if (n = 0 and i = 0)

go to symtab check if the TA is in there. If it is, print '#' and the label

else print

"#" and the decimal value of the TA

05 - 01 = 04 → OPCODE

0000 0100

hex

n	i	x	b	p	e
@	#	add ,x	TA=disp+(B)	TA=disp+(PC)	TA=addr

```
if (b=1, p=0, e=0)
    disp = last 3
    TA = disp + (B)
else if (b=0, p=1, e=0)
    TA = disp + (PC)
else if (b=0, p=0, e=1)
    address = read next 4
    TA = address
else (b=0, p=0, e=0)
    address : read next 3
    TA = address
    string s = "   "
if (n)
    s = "@" + TA                    CHECK SYMTAB
                                      FOR TA
else if (i)
    if TA not in SYMTAB
    convert TA to decimal
    s = "#" + decimal value of TA
    if (x)
        s = TA + ",x"
```

After determining how to handle nixbpe bits of Format 3 and 4 instructions, we focused on the calculation of the target address of each opcode. We spent one long meeting discussing how to calculate the address of a certain label as we were getting confused on if we had to add or subtract the program counter from the display to get the target address. When we finally understood how to calculate the target address needed to find a label, we concentrated on how to process the information of each label in the .sym file, assembler directives, and the creation of an optab for operands. This work was done individually and is also the time that we started to finally code. We had another meeting to talk about problems we encountered while coding and the addition of certain variables or components that we thought were missing from our program.

(8)

WALKING THROUGH THE TEXT RECORD

sample.sym

<SYMTAB>

FIRST	000000	R
LOOP	00000B	R
COUNT	00001E	R
TABLE	000021	R
TABLE2	001791	R
TOTAL	002F01	R

<LITTAB>

(in bytes)

Lit	Length	Address
=X'3F'	2	000003

X = 'XX' = 1 bytes

C = 'CC' = 2 bytes

sample.obj

(length in bytes)

H^SUM___^000000^002F04 ↵

1 byte

T^000000^1E^050000^032003^3F^69101791^1BA013^

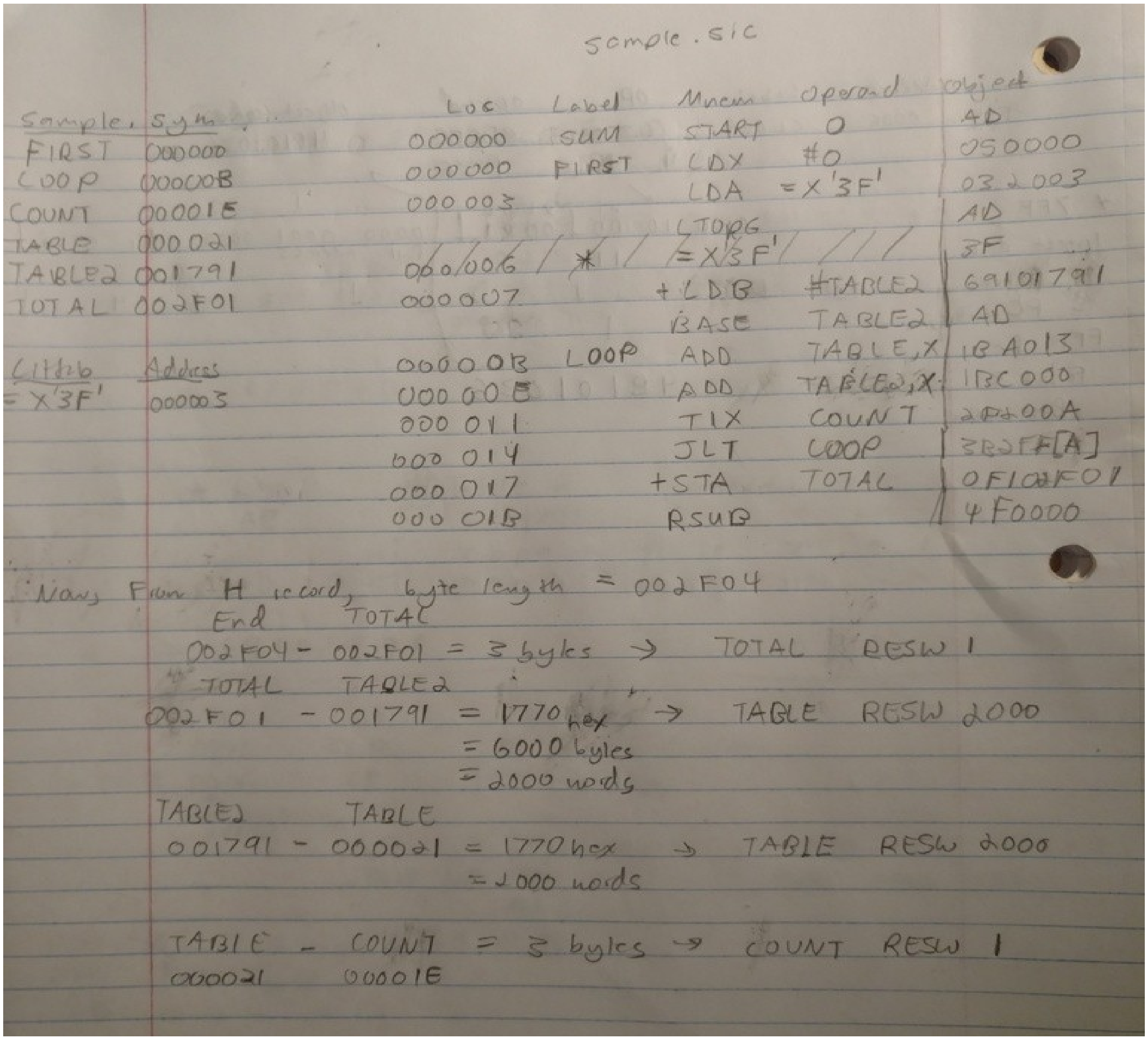
starting address	length of record in bytes	LOCCTR = 000000 PC = 03	LC = 000003 PC = 06	LC = 000006 PC = 07	LC = 000007 PC = 11 = B	LC = 000011 = B PC = 14 = E

1E	050000	032003
16 + 14 = 30 byte record	↓ 0101 0000 ni xbpe = 04 = LDX i = immediate (#)	disp = 003 0011 0010 ni xbpe = 00 = LDA 11 = simple (XE) PC relative PC = 06 TA = PC + disp TA = 06 + 003 TA = 09 ? LITTAB ?

69101791	1BA013
↓ 0101 1001 0001 ni xbpe op = 68 = LDB e = + i = immediate (#) address = 01791 = TABLE2	0001 1011 1010 → 013 ni xbpe op = 18 = ADD ni = 11 = simple (XE) x = indexed p = 1 = pc relative disp = 013 TA = PC + disp E + 013 = 21 = TABLE

The last week before we had to turn in the assignment, we worked on perfecting our coding tasks. We worked on completing our code so that they returned the correct information. We consistently checked for errors in our code by testing our code with the object file and symbol table file given to us.

For example, we had trouble printing out the assembler directives such as RESW and RESB at the end of the opcodes. We continually debugged our code to try to find a solution for it and also tried to solve the problem by hand first.



We checked with each other to clarify any mistakes in the code and misunderstandings of what our tasks would be. Because each team member worked on their own separate codes, we always tried to provide feedback and solutions when the program did not work the way it was supposed to. An example is SymLinkedList.cpp. This file was created to help retrieve the information pertaining to symbols extracted from the .sym file. We initially created findLabel.cpp for finding a symbol's information from the SYMTAB by organizing its name, value, and flag in separate arrays. After much deliberation, we found that the linked list data structure would be better for storing the data of the symbols. Overall, the SymLinkedList.cpp made it easier to get the information needed to print out the values at their locations.

The creation of this disassembler helped us, as a team, realize the importance of planning in such a complicated assignment. We realized that it was better to first consult with each other before actually coding and that our meetings essentially made the coding easier because everything became planned out. Although we did run into problems from time-to-time, we always tried to ask for help when needed and got the job done.