‘xed’

SIC XE DISASSEMBLER- DESIGN DOCUMENT

AUTHORS:

- Jack Bruce (RedID 822320220, cssc0420)
- Jacob Romio (RedID 822843795, cssc0413)
- All program materials located on **cssc0420@edoras.sdsu.edu:~/a2**

TEAM ORGANIZATION:

- We utilized **git** version control system by using a **feature-branch** workflow style
  - We divided up work based on what was needed chronologically.
  - Each feature was given its own dedicated branch prior to merging it to the master branch.
- We worked extremely well together, our workflow could be improved by each of us being more familiar with **git**
  - Resolving conflicts that occur during merges was a hurdle that we struggled to overcome efficiently

OVERVIEW:

1. Open and store XE object file and its accompanying symbol file
   a. <filename>.obj
   b. <filename>.sym
      i. Contains SYMTAB and LITTAB
2. Disassemble object code
3. Generate XE source file and XE listing file
   a. <filename>.sic & <filename>.lis

SYNOPSIS:

% xed <filename>

/*

STEP 1:

```c
/**********************[ Record Types ]**********************/
typedef struct Header
{
    char name[6]; //[1-6] (ASCII form)
    unsigned int startadr; //[7-12] (numeric form)
    unsigned int prglen; //[13-18] (numeric form)
}Header;

typedef struct Text
{
    unsigned int startadr;
    unsigned int reclength;
    unsigned char inst[30];
}Text;

typedef struct Mod
{
    unsigned int startadr;
    unsigned int modLength;//IN 1/2 Bytes
    //SYMBOL VALUE UNKNOWN(char*?)
}Mod;

typedef struct End
{
    unsigned int firstinst ;//optional?
}End;
/******************[ End of Record Types ]*******************/
```

Given <filename>, open <filename>.obj and <filename>.sym. Then process them accordingly. (We won't need .sym til part 2)
We will store each record in one of these 4 structs ^

a) **READ AND STORE .OBJ FILE** (records.c/h)
   1. Check if <filename>.obj and <filename>.sym exist
      a. **NO?** exit program with appropriate error message.
      b. **YES?** Continue.
   2. Open <filename>.obj and <filename>.sym
   3. For .obj (**see records.h)**
      a. There will be 2 passes
         i. **PASS 1:**
            1. Count how many **T** and **M** records exit in file
               a. **tCount & mCount**
            2. **tCount & mCount** will decide how large **T[ ]** & **M[ ]** are
         ii. **PASS 2:**

1. Look for record type flags
    a. **H, T, M, E**
2. Each time one of these signal characters is encountered
    a. Create appropriate struct in scope of main()
        i. 1 - **Header**
        ii. **tCount - Texts**
            1. Pointers for these will be stored in **T[ ]**
        iii. **mCount - Mods**
            1. Pointers for these will be stored in **M[ ]**
        iv. 1 - **End**

3. Populate struct variables accordingly
    a. Header
    b. Text
    c. Modification
    d. End

b) READ AND STORE .SYM FILE (symbol.c/h)

```c
typedef struct Symbol
{
    char label[7];
    int value; /* made this signed as value is not
                  necessarily an address */
    char type;
}Symbol;

typedef struct Literal
{
    char name[7];
    char literal[9];
    unsigned int length;
    unsigned int address;
}Literal;
```

1. The goal of the this step is to convert the information stored in <filename>.sym into data structures representing **SYMTAB** and **LITTAB**
2. To accomplish this, we used a similar 2 pass approach from the above algorithm
    a. **PASS 1:**
        i. Check <filename>.sym for proper **SYMTAB HEADER**
            1. This is the part that lists the fields followed by a line of '_'s
        ii. Count every line until 2 new line characters are encountered in succession
            1. This is the **symbolcount**
        iii. Check <filename>.sym for proper **LITTAB HEADER**
        iv. Count every line until EOF
            1. This is the **literalcount**
    b. **PASS 2:**
        i. With these counts we make to arrays:
            1. **SYMTAB[]**
                a. Array of **Symbol** pointers
            2. **LITTAB[]**
                a. Array of **Literal** pointers

ii. Parse <filename>.sym for **SYMTAB** section and **LITTAB** sections accordingly
    1. Proper field parsing relies on every field occupying a consistent number of columns
iii. Populate **Symbol** and **Literal** structs and store in arrays

/*

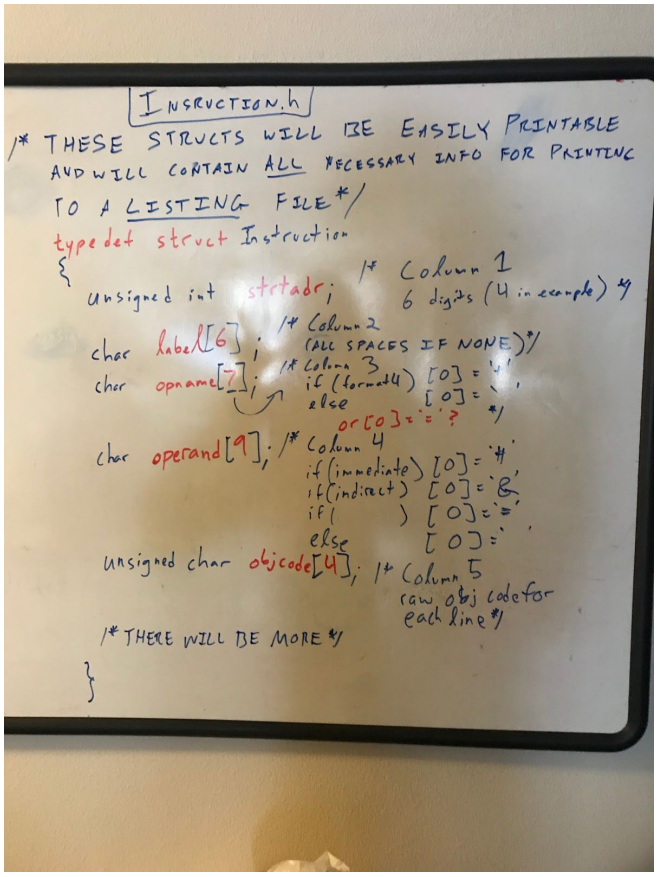
STEP 2: (BIG STEP)

With all of our populated structs we have to process each one accordingly. Easy enough!

a) From **Text** record to **Instruction**
b) Opcode storage and retrieval (**OPTAB**)
c) Determining format and addressing modes for each instruction
d) Recognizing and processing assemble directives
  i) RESB/RESW
  ii) BASE/NO BASE
  iii) WORD/BYTE

a) From **Text** record to **Instruction:**

I think this design will be best approached if we work backwards. See this example of an **Instruction** struct. If we can fully populate this struct. Formatting and outputting will result in a completed listing file.

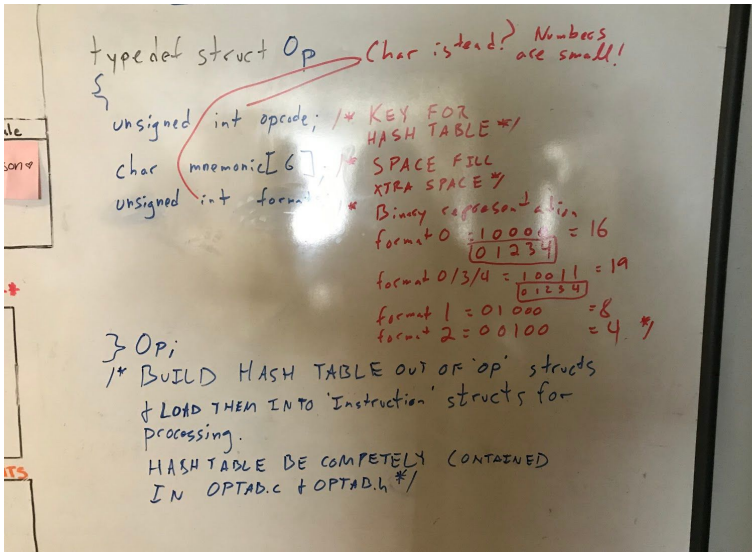


This turns the complex job of disassembling into 5-ish *slightly* less complex jobs. ***Forget the big picture. Just populate one field at a time.***

b) Opcode storage and retrieval (**OPTAB**)

Operations will be packaged in these **Op** structs. **OPTAB** will simply be a table of **Op** structs.

[Due to time constraints, we went with a more direct approach and did not follow the Op struct design.]



OPTAB.c (This will *hopefully* just be a hash table containing)

1. BuildOpTab()

   Op OpTab[150];

   OpTab[18] = malloc(sizeof(Op);
   OpTab[18]->opcode = 18;
   OpTab[18]->mnemonic = “ADD “
   OpTab[18]->format = 19;
2. Op GetOp(int opcode) {

   Return OpTab[opcode];

   }

c) Determining format and addressing modes for each instruction

Finding Format and Addressing modes will be in another class.

Format.c (This will do addressing modes too)

- **Text** struct will be input
- This will deal with populating the **Instruction** structs
  - Create Instruction array in main (maybe?)
- Addressing modes (example)

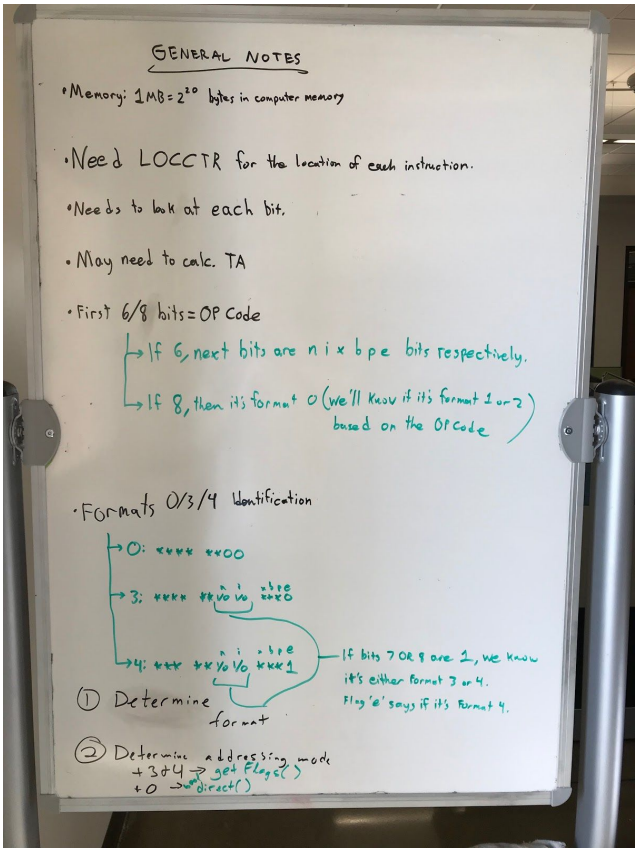
  For immediate addressing:

Instruction->operand[0] = ‘#’;
Priority is given to symbols, then data.
If it’s something (aka no special symbol)
Instruction->operand[0] = ‘ ‘;

- GetAddrMode(Instruction inst)
  inst->operand[0] = ‘#’ || ‘&’ || ‘ ‘;

Format Specification Procedure(Byte by byte):

Check first byte
>If Format 1 Op Code, DONE
>else if format 2 op code, grab 2nd byte and DONE
>else: Must be format 0/3/4
>Get 2nd byte & check * * 1/0 1/0 bits of 2nd byte
if(bit3==0 && bit4==0)
format 0 & grab 3rd byte DONE
else
format 3 or 4
>Grab next 3rd byte
>if(e==0)
Format 3
Else
Format 4 and grab 4th byte



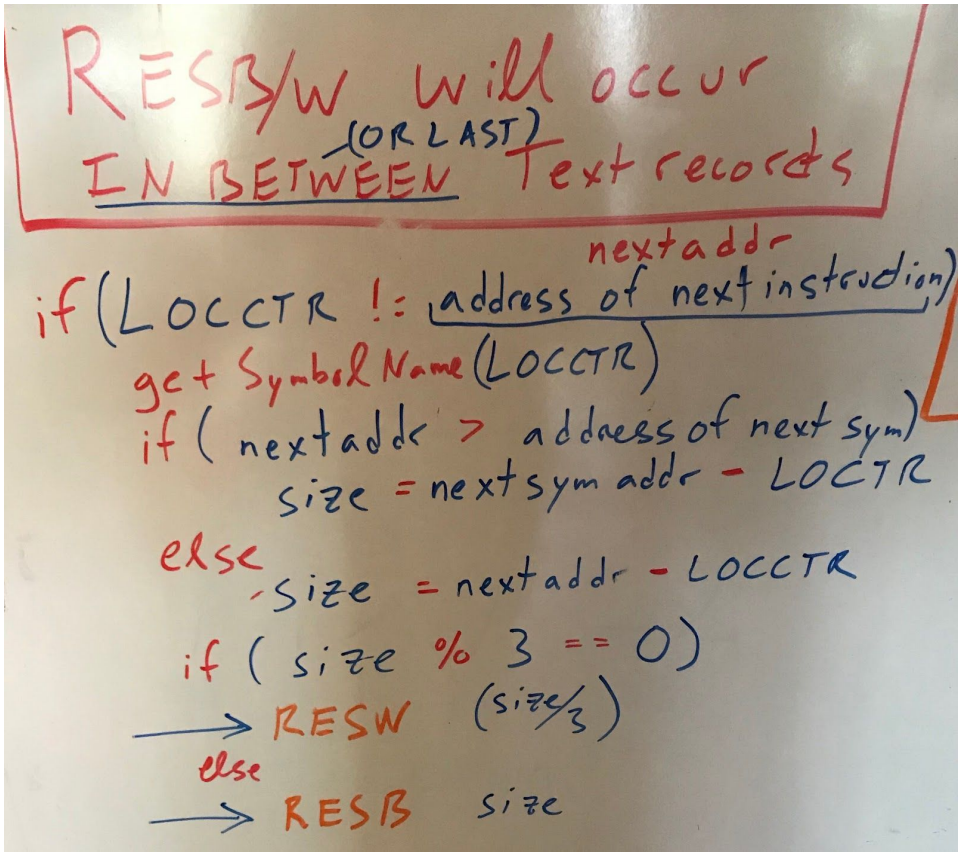
General Notes brainstorming how to deal with reading different formats.

d) Recognizing and processing **assembler directives**

iv) RESB/RESW

We made this algorithm to determine address and value of **RESB/RESW** directives. This algorithm is based on the fact (assumption) that these directives will only occur in between **Text** records, or after the last **Text** record has been processed.

Technically these are directives (not instructions) but we will store these *lines* in instruction structs to relieve special cases in printing the listing file



v) BASE/NO BASE

The BASE directive should occur when a base relative instruction is read in. After such, the next PC relative instruction will display a NO BASE directive.

Due to time constraints, the BASE directive appears after a LDB instruction occurs to match source code in the book.

vi) Literals

1) I literal declaration occurs when **LOCCTR** equals an address of a symbol in **LITTAB**
2) It is preceded by and **LTORG** statement if more instructions follow it.
    - **(a) LTORG** does not occur when a default literal pool appears at the bottom of the source code

vii) **WORD/BYTE (**experimental)

1) It is possible for data values to match a valid instruction format so the accuracy of these statements will vary
    - If the last byte of a Text record is not a format 1 instruction, we assume it is a BYTE directive containing data.
2) If a symbol is recognized to appear at the current address (**LOCCTR**) then the following would trigger a **WORD/BYTE** directive:
    (a) No Symbol recognized for an operand of recognized instruction
        (i) This is less likely for immediate addressing modes
    (b) Invalid addressing mode combination
    (c) Format 0 in a SIC/XE file, vice versa

/*

STEP 3:

After filling the Instruction structs, writing files <filename>.lis and <filename>.sic was a matter of iterating through our linked list of Instruction structs and writing each corresponding file using each Instruction struct in order.

This was made a simple task due to our design choice of making Instruction structs to properly match the source code format and by putting each Instruction struct into a linked list ready to print after the file was finished being read.

Style:

We did our best to follow Leonard's C example files as closely as possible

- There will be a header at the top of each file
- There will be a footer notating EOF of each file
- Follow Leonard's method template for every method you make
    - I have included one for every method I make

Variables: (no camel case)

- int eachwordlowercase;

Methods:

- EachWordUpperCase();

Comments:

- /* Practically all comments should be
    embedded like this */
    - Be careful to keep ALL code from passing the width limit.