Software Design Document (SDD)
Assignment 2# DXE Disassembler for XE computer
CS530, Spring 2020

Team:

- Nathaniel Stewart, cssc1901
- Tim Tea, cssc1956
- Francisco Melendez Hernandez, cssc1905
- Thong Le, cssc1902

Overview & Goals:

- Read and store object code and symtab files into internal data structure.
- Translate object code into assembly code.
  - Implement map/table of SIC/XE opcodes in C++
  - Develop an algorithm to parse through object code
  - Able to translate the wide range of XE features such as labels, accurate displacement calculation for addressing, instruction formats, indirect addressing, etc.
  - Able to handle as many types of errors as possible (see *System Specification* for specific errors we expect to encounter)
- Store translated results into generated listing (.lis) and source (.sic) files.
- Create a <dxe> executable via Makefile.
- Program compiles successfully on Edoras (Linux).

Project Description:

- For this project we will be developing a disassembler for SIC/XE machine code. The primary purpose of the disassembler is to parse through XE object code and transliterate that code into assembly language. The main files of this project include an executable file <dxe> (format '%dxe <filename>') that requires an object file <filename>.obj and symbol file <filename>.sym in the same directory to execute. Upon execution two files, the listing file <filename>.lis and source file <filename>.sic, will be generated containing the translated assembly codes.

Plan of Action and Milestones:

- Successfully read object and symbol files.
- Implement the opcode, symbol, and literal maps.
- Implement record classes and data structures (T).
- Transfer record information found in the object and symbol files into their appropriate data structures.

- Implement algorithms to read through T data structures and calculate/extract addresses and translate opcodes by cross-referencing opcode struct array, to store into SourceStatement struct instances.
- Assemble SourceStatement instances to write into new listing and source files.
- Implement error handling for potential edge cases and errors described in System Specifications.
    - Thoroughly test edge cases.
    - Test compiling on edoras/linux.

Requirements:
- C/C++ source file
- Header File
- Compile on edoras ("a2" directory)
- README file
- Makefile
- Software Design Document

System Design/Specification:
- System Specification:
    - File formats involved: .cpp (c++ source file), .h (header file), .sic (source file), .lis (listing file), .obj (object file), .sym (symbol table), .txt (text)
    - As this program is written in C++, the user must have g++ installed or be using an IDE with g++. Unless the executable <dxe> file already exists within the directory, the user must enter <make> to prompt the Makefile to generate the executable file. The execution format is 'dxe <filename>'. <filename> must be an .obj file. This requires .obj and .sym files present in the same directory in order to correctly execute.
    - Upon execution, two files, a .lis listing file and a .sic source file, should be generated containing the translated assembly code.
    - Error Processing: the following errors should be handled internally by the program:
        - Incorrect arguments executed (ex. incorrect number of arguments in command). Anything that does not fit the 'dxe <filename>' format will print out an error message with the correct format to be used and exit.
        - Our program should detect whether the <filename> file is the correct file format (which is .obj). If not, the program will print an error message and exit.

        - Symbol table or object file missing. Our program will detect whether one or both of these files are missing upon compile, and will exit (and print an error message of which is missing) if so.
        - Incorrect addresses. Should any of the addresses fall outside of the boundaries of the address range, or reference anything that does not exist, the program should print an error message of the specific error location and exit.
- System Design:
    - Internal tables. Our program will contain references to SIX/XE instruction sets, opcodes, and records. These tables will provide the scaffolding needed to translate object code into assembly language.
    - Line-by-line representation by class. Each line of the assembly language source code file will be represented by a struct object instance. The class will be called "lisStatement", and each line/statement will be stored in an ordered to be iterated and printed into the source and listing files at the end of translation.
    - Process 1: Store file contents into data structures for utility.
        - Data structures used:
            - vector<string> *symFileVector*: holds all lines of .sym file
            - vector<string> *objFileVector*: holds all lines of .obj file
            - vector<string> *TRecordVector*: holds T record lines of .obj file
            - vector<string> *MRecordVector*: holds M record lines of .obj file
        - Process:
            - The lines of the .sym file are read via stream and stored into *symbolFileVector*. The lines of the .obj file are read via stream and stored into *objFileVector*.
                - If the line is a T record, it is pushed into *TRecordVector*.
                - If the line is an M record, it is pushed into *MRecordVector.*
    - Process 2: Build a symtab and littab.
        - Data structures used:
            - vector<string> *symFileVector*
            - struct *symbol* {string *name,* string *flag,* unsigned int *value*}: represents a symbol, with variable names corresponding to attributes of a symbol. *value* is the address of the symbol.
            - struct *literal* {string *name*, unsigned int *length*, unsigned int *address*, *string* lit}: represents a literal, with variable names corresponding to attributes of a literal
            - vector<symbol> *symtab*: vector of symbol instances
            - unordered_map<int, literal> *littab*: maps literal struct instances (value) to addresses (key)

        - Process: *symbolFileVector* has two of the first lines (the headers and the horizontal divider) removed. It then iterates through its elements until it finds the header for the literals, and then removes three lines (the empty line before the header, the header, and the following horizontal divider). Two more iterations are then made through *symbolFileVector*, the first uses a *stringstream* to enter columns of the table in each line (represented as words) into *symbol* structs. Each *symbol* struct instance represents a row of the table, and each is pushed into *symtab*. The same is done in the second iteration, this time using *literal* struct instances to represent rows, which are then mapped into *littab* with their addresses as keys.
    - Process 3.1: Read H record.
        - Data structure used: vector<string> *objFileVector*.
        - Process: the first element of *objFileVector* (the first line of .obj) is extracted for the program name, the start address, and the program length.
    - Process 3.2: Read T records.
        - Data structures used:
            - vector<string> *objFileVector*
            - vector<string> *TRecordVector*
            - struct *instruction*{string *objectCode*, unsigned int *loc*, unsigned int *nixbpe*, unsigned int *opcode*}: represents extracted instructions from T records
            - vector<instruction> *instructionList*: holds *instruction* struct instances
        - Algorithm:

```
for each record in TRecordVector, read nibbles
     get T record starting address
     get T record length
     set locctr = T record starting address
     if locctr coincides with address in littab
          if this is the first literal
               mark as address of LTORG
          endif
          create literal declaration statement
          add literal length to locctr
     endif
     get current opcode
     get current nixbpe
     create new instruction
```

```
      if locctr coincides with symbol from symtab
            set label of instruction to symbol
      endif
      if instruction is format 1
            set nixbpe to 0
            set instruction object code to next 2 nibbles
            push instruction into instruction list
            move locctr forward 1 byte
      else if instruction is format 2
            set nixbpe to 0
            set instruction object code to next 4 nibbles
            push instruction into instruction list
            move locctr forward 2 bytes
      else if instruction is format 4 (nixbpe & 0x01 == 1)
            set instruction object code to next 8 nibbles
            push instruction into instruction list
            move locctr forward 4 bytes
      else
            set instruction object code to next 6 nibbles
            push instruction into instruction list
            move locctr forward 3 bytes
      endif
end for
```

- Process 4: Build RESB statements
  - Data structures used:
    - int reserveMarker: holds the position of the latest label
    - vector<symbol> symtab
  - Note: RESW and RESB do not have any object code representation. The only information we are given are their positions in *symtab*. This poses a great challenge in constructing an accurate algorithm, and so our algorithm relies on two premises in order to correctly execute:
    - 1. each RESW or RESB statement is declared one after the other in source code
    - 2. RESW and RESB are declared at the end of the program.

  The violations of these two premises will result in an incorrect or inaccurate placement of the statements in source code.

- Algorithm:

```
for every element sym in symtab from reserveMarker to symtab size
      create RESB declaration with sym name
      set RESB declaration address to current sym address
      if sym is last element in symtab
            set bytes reserved = (program length - sym address)
      else
            set bytes reserved = (next sym address - sym address)
      endif
      push RESB declaration into list of source code statements
endfor
```

- Process 5: Calculate target addresses and operands

If opcode is 0x68 (LDB)
    Set the Base register value equal to the value of the displacement
If format 4
    Concatenate "+" to the opcode
If format 2
    if opcode is 0xB4 (CLEAR) or 0xB8 (TIXR) or 0xB0 (XVC)
        Set operand variable to "register1"
    Else (every other format 2 instruction)
        Set operand variable to "register1,register2"
Else format 3
    If simple
        If indexed
            If base relative
            Else if pc relative
            Else direct
        Else non-indexed
            If base relative
            Else if pc relative
            Else direct
    Else if indirect
        If base relative
        Else if pc relative
        Else indirect
    Else if immediate
        If base relative
        Else if pc relative
        Else indirect
Else if format 4
    If simple
        If indexed

```
                (Target address) = (X) + Address
                        If the value of the target address is a symbol
                                Target = symbol name
                        Operand = Target + “,X”

        If non-indexed
                (Target address) = Address
                        If the value of the target address is a symbol
                                Target = symbol name
                        Operand = Target
    Else if indirect
                ((Target address)) = Address
                        If the value of the target address is a symbol
                                Target = symbol name
                        Operand = “@” + Target
    Else immediate
                Target address = Address
                        If the value of the target address is a symbol
                                Target = symbol name
                        Operand = “#” + Target
If format 3
    Check the flag bits
    If Simple Addressing
        If Indexed
            If Base relative
                (Target address) = (B) + (X) + Displacement
                If the value of the target address is a symbol
                        Target = symbol name
                Operand = Target + “,X”
            If PC
                (Target address) = (PC) + (X) + Disp
                If the value of the target address is a symbol
                        Target = symbol name
                Operand = target + “,X”
            Else
                (TA) = Disp + (X)
                If the value of the target address is a symbol
                        Target = symbol name
                Operand = target + “,X”
        Else
            If Base
                (Target address) = (B) + Displacement
                If the value of the target address is a symbol
                        Target = symbol name
                Operand = Target
```

```
                If PC
                        (Target address) = (PC) + Disp
                        If the value of the target address is a symbol
                                Target = symbol name
                        Operand = target
                Else
                        (TA) = Disp
                        If the value of the target address is a symbol
                                Target = symbol name
                        Operand = target
If Indirect
        If Base
                ((TA)) = Disp +  (B)
                If the value of the target address is a symbol
                        Target = symbol name
                        Operand = "@" + target
        If PC
                ((TA)) = Disp +  (PC)
                If the value of the target address is a symbol
                        Target = symbol name
                        Operand = "@" + target
        Else
                ((TA)) = Disp
                If the value of the target address is a symbol
                        Target = symbol name
                        Operand = "@" + target

Else If Immediate
        If Base
                TA = Disp +  (B)
                If the value of the target address is a symbol
                        Target = symbol name
                        Operand = "#" + target
        If PC
                TA = Disp +  (PC)
                If the value of the target address is a symbol
                        Target = symbol name
                        Operand = "#" + target
        Else
                TA = Disp
                If the value of the target address is a symbol
                        Target = symbol name
                        Operand = "#" + target
```

Insert calculated variables to lisStatement struct, with 5 fields holding output column accordingly

- Process 6: Convert instructions into lisStatement structs to be added to an ordered map of the source code.
    - Primary Data Structures used:
        - map srcmap<int, lisStatement>: key is address, value is lisStatement
        - struct lisStatement {col1, col2, col3, col4, col5}: to hold the five columns of information in source code file
        - unordered_map<int, string>: key is register number, value is register name, to be used for format 2 instruction translation
    - Process: Goes through each instruction and extracts information to enter into lisStatement struct. Calculates the operand (col4) for each instruction through a series of logic branches, taking into account 'nixbpe' information.
- Process 7: Write information from srcmap into produced .lis and .sic files.

Development Environment:

- The program will be written in C++. The IDE's involved will be Visual Studio and VIM.
- Workflow: Each feature (or bullet under the *Plan of Action* heading) will be developed by a pair of programmers. Both pairs will work on their separate features, however each pair must test the other pair's code at the end of each development cycle.
- Debugging: We will be using Visual Studio's debugging features to help us solve obstacles we encounter in our code.
- Code Sharing: Since our program will not have enough independent parts to necessitate git, code will be shared on a single Google Document.
- Communication: Established over Discord.

Run/Test Environment:

- The program will be run and tested on edoras, which already contains the g++ compiler.
- Our testing will use the sample files found on Blackboard. Since the object and source files are already there, we can simply cross-reference to verify our program works as intended.
- Debugging: As mentioned in our *Development Environment* section, we will be using Visual Studio's debugging features to help us solve problems we encounter in our code. Visual Studio will already be equipped to notify us of almost all of our syntax errors, but should we encounter logical, branching, or looping errors we will resolve that in our second or third round of testing.

REFERENCE

SIC/XE Instruction Set Table

SIC Instructions are in blue

Mnemonic	Format	Opcode	Effect	Notes
ADD m	3/4	18	A <-- (A) + (m..m+2)	
ADDF m	3/4	58	F <-- (F) + (m..m+5)	X F
ADDR r1,r2	2	90	r2 <-- (r2) + (r1)	X
AND m	3/4	40	A <-- (A) & (m..m+2)	
CLEAR r1	2	B4	r1 <-- 0	X
COMP m	3/4	28	A : (m..m+2)	
COMPF m	3/4	88	F : (m..m+5)	X F C
COMPR r1,r2	2	A0	(r1) : (r2)	X F C
DIV m	3/4	24	A : (A) / (m..m+2)	
DIVF m	3/4	64	F : (F) / (m..m+5)	X F
DIVR r1,r2	2	9C	(r2) <-- (r2) / (r1)	X
FIX	1	C4	A <-- (F) [convert to integer]	X F
FLOAT	1	C0	F <-- (A) [convert to floating]	X F
HIO	1	F4	Halt I/O channel number (A)	P X
J m	3/4	3C	PC <-- m	
JEQ m	3/4	30	PC <-- m if CC set to =	
JGT m	3/4	34	PC <-- m if CC set to >	
JLT m	3/4	38	PC <-- m if CC set to <	
JSUB m	3/4	48	L <-- (PC); PC <-- m	
LDA m	3/4	00	A <-- (m..m+2)	
LDB m	3/4	68	B <-- (m..m+2)	X
LDCH m	3/4	50	A [rightmost byte] <-- (m)	
LDF m	3/4	70	F <-- (m..m+5)	X F
LDL m	3/4	08	L <-- (m..m+2)	
LDS m	3/4	6C	S <-- (m..m+2)	X
LDT m	3/4	74	T <-- (m..m+2)	X
LDX m	3/4	04	X <-- (m..m+2)	
LPS m	3/4	D0	Load processor status from information beginning at address m (see Section 6.2.1)	P X
MUL m	3/4	20	A <-- (A) * (m..m+2)	
MULF m	3/4	60	F <-- (F) * (m..m+5)	X F
MULR r1,r2	2	98	r2 <-- (r2) * (r1)	X
NORM	1	C8	F <-- (F) [normalized]	X F
OR m	3/4	44	A <-- (A) | (m..m+2)	
RD m	3/4	D8	A [rightmost byte] <-- data from device specified by (m)	P
RMO r1,r2	2	AC	r2 <-- (r1)	X
RSUB	3/4	4C	PC <-- (L)	
SHIFTL r1,n	2	A4	r1 <-- (r1); left circular shift n bits. {In assembled instruction, r2=n-1}	X
SHIFTR r1,n	2	A8	r1 <-- (r1); right shift n bits with vacated bit positions set equal to leftmost bit of (r1). {In assembled instruction, r2=n-1}	X
SIO	1	F0	Start I/O channel number (A); address of channel program is given by (S)	P X
SSK m	3/4	EC	Protection key for address m <-- (A) (see Section 6.2.4)	P X
STA m	3/4	0C	m..m+2 <-- (A)	
STB m	3/4	78	m..m+2 <-- (B)	X
STCH m	3/4	54	m <-- (A) [rightmost byte]	
STF m	3/4	80	m..m+5 <-- (F)	X
STI m	3/4	D4	Interval timer value <-- (m..m+2) (see Section 6.2.1)	P X
STL m	3/4	14	m..m+2 <-- (L)	
STS m	3/4	7C	m..m+2 <-- (S)	X
STSW m	3/4	E8	m..m+2 <-- (SW)	P
STT m	3/4	84	m..m+2 <-- (T)	X
STX m	3/4	10	m..m+2 <-- (X)	
SUB m	3/4	1C	A <-- (A) - (m..m+2)	
SUBF m	3/4	5C	F <-- (F) - (m..m+5)	X F
SUBR r1,r2	2	94	r2 <-- (r2) - (r1)	X
SVC n	2	B0	Generate SVC interrupt. {In assembled instruction, r1=n}	X
TD m	3/4	E0	Test device specified by (m)	P C
TIO	1	F8	Test I/O channel number (A)	P X C
TIX m	3/4	2C	X <-- (X) + 1; (X) : (m..m+2)	C
TIXR r1	2	B8	X <-- (X) + 1; (X) : (r1)	X C
WD m	3/4	DC	Device specified by (m) <-- (A) [rightmost byte]	P

RECORDS

Object Code Format

- **Header** record

Col 1	**H**
Col 2-7	**program name**
Col 8-13	**starting address (hex)**
Col 14-19	**length of object program in bytes (hex)**

- **Text** record

Col 1	**T**
Col 2-7	**starting address in this record (hex)**
Col 8-9	**length of object code in this record in bytes (hex)**
Col 10-69	**object code (69-10+1)/6=10 instructions**

- **End** record

Col 1	**E**
Col 2-7	**address of first executable instruction (hex)**

Relocatable Program

- **Modification record**

Col 1	**M**
Col 2-7	**Starting location of the address field to be modified, relative to the beginning of the program**
Col 8-9	**length of the address field to be modified, in half-bytes**

Define record:

Col. 1	D
Col. 2–7	Name of external symbol defined in this control section
Col. 8–13	Relative address of symbol within this control section (hexadecimal)
Col. 14–73	Repeat information in Col. 2–13 for other external symbols

Refer record:

Col. 1	R
Col. 2–7	Name of external symbol referred to in this control section
Col. 8–73	Names of other external reference symbols

SYMTAB FILE (INPUT)

```
Symbol  Value   Flags:
-----------------------
FIRST   000000  R
LOOP    00000B  R
COUNT   00001E  R
TABLE   000021  R
TABLE2  001791  R
TOTAL   002F01  R

Name    Literal  Length Address:
------------------------------
        =X'3F'     2    000003
```

OBJECT FILE (INPUT)

```
HSUM   000000002F04
T0000001E0500000320033F691017911BA0131BC0002F200A3B2FF40F102F014F0000
M00000805
M00001805
E000000
```

SIC FILE (OUTPUT)

```
.  SOURCE CODE FOR THE XE VERSION OF THE SIC FAMILY OF COMPUTER
SUM      START   0                    SIMPLE SAMPLE PROGRAM
FIRST    LDX    #0
         LDA    =X'3F'
         LTORG
        +LDB    #TABLE2
         BASE    TABLE2
LOOP     ADD     TABLE,X
         ADD     TABLE2,X
         TIX     COUNT
         JLT     LOOP
        +STA     TOTAL
         RSUB
COUNT    RESW    1
TABLE    RESW    2000
TABLE2   RESW    2000
TOTAL    RESW    1
         END     FIRST
```

LIS FILE (OUTPUT)

```
0000  .  SOURCE CODE FOR EXAM #2
0000  EXAM2    START   0
0000  FIRST    CLEAR   X        050000
0003           LDA    #0
0003          +LDB    #TABLE2   69101791
000B           BASE    TABLE2
000B  LOOP     ADD     TABLE,X  1BA013
000E           ADD     TABLE2,X 1BC000
0011           TIX     COUNT    2F200A
0014           JLT     LOOP     3B2FF4
0017          +STA     TOTAL    0F102F01
001B           RSUB             4F0000
001E  COUNT    RESW    1
0021  TABLE    RESW    100
1791  TABLE2   RESW    100
2F01  TOTAL    RESW    1
               END     FIRST
```