TUTORIAL - 8

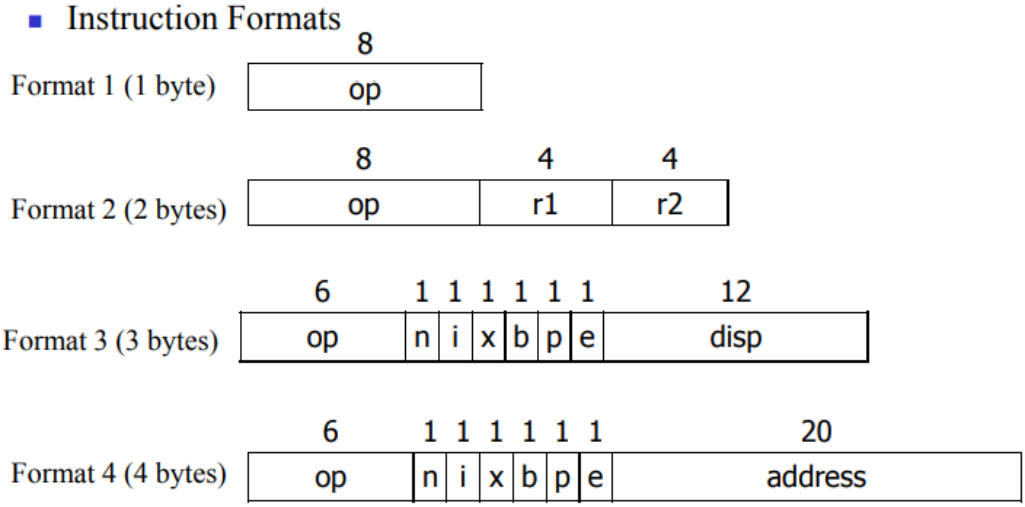
ASAVADI THANMAI SAHITH
20114018

Introduction:

The main objective here is to implement a version of two-pass SIC/XE assembler.

The Assembler includes all the SIC/XE instructions and support all the four addressing modes alongside program relocation.

- Instruction Formats



Formats 1 and 2 are instructions that do not reference memory at all

Addressing modes

- Base relative (n=1, i=1, b=1, p=0)
- Program-counter relative (n=1, i=1, b=0, p=1)
- Direct (n=1, i=1, b=0, p=0)
- Immediate (n=0, i=1, x=0)
- Indirect (n=1, i=0, x=0)
- Indexing (both n & i = 0 or 1, x=1)
- Extended (e=1 for format 4, e=0 for format 3)

It also includes Machine-Independent features like:

1. Literals
2. Expressions
3. Program Blocks
4. Control Sections and Programing Linking
5. Symbol Defining Statements

Input : Assembler source program using the instruction set of SIC/XE.
Output : Output contains:

1. A symbol table generated in Pass 1.
2. Intermediate file generated by Pass 1 used by Pass 2.
3. Listing file generated by Pass 2.
4. Object program file generated by Pass 2.
5. Error file containing errors in the input (if any).

Instructions for execution(README):

SIC-XE-Assembler-

Steps to compile and run the program:-

1. execute

```
chmod u=rwx compile.sh
```

inside the project directory to get execution permissions.

2. run compile.sh script using command

```
./compile.sh
```

3. An object file named assembler.out will be generated in the INPUTS folder.
4. cd to INPUTS folder and run assembler.out file using

```
cd INPUTS
./assembler.out
```

5. Type the input file name to generate the object and listing files. There will be a test_input.txt file in INPUTS folder just for testing purpose.

```
..ember/INPUTS
(base) ➜  SIC-XE-Assembler git:(master) ✗ ls
INPUTS        README.md     assembler.exe compile.sh    pass1.cpp     pass2.cpp     tables.cpp    utility.cpp   ~$Report.docx ~$reprt.docx
(base) ➜  SIC-XE-Assembler git:(master) ✗ ./compile.sh
(base) ➜  SIC-XE-Assembler git:(master) ✗ ls
INPUTS        README.md     assembler.exe compile.sh    pass1.cpp     pass2.cpp     tables.cpp    utility.cpp   ~$Report.docx ~$reprt.docx
(base) ➜  SIC-XE-Assembler git:(master) ✗ cd INPUTS
(base) ➜  INPUTS git:(master) ✗ ls
assembler.out  test_input.txt
(base) ➜  INPUTS git:(master) ✗ ./assembler.out
****Input file and executable(assembler.out) should be in same folder****

Enter name of input file:test_input.txt

Loading OPTAB

Performing PASS1
Writing intermediate file to 'intermediate_test_input.txt'
Writing error file to 'error_test_input.txt'
Writing SYMBOL TABLE
Writing LITERAL TABLE
Writing EXTREF TABLE
Writing EXTDEF TABLE

Performing PASS2
Writing object file to 'object_test_input.txt'
Writing listing file to 'listing_test_input.txt'
(base) ➜  INPUTS git:(master) ✗ ls
assembler.out        error_test_input.txt        intermediate_test_input.txt listing_test_input.txt      object_test_input.txt       tables_test_input.txt       test_input.txt
(base) ➜  INPUTS git:(master) ✗
```

Functionalities and Specifications :

Functions:

pass1() : Use the source files to update intermediate and error files. If the source file is not found, or if the intermediate file does not open, the corresponding error is written to the error file, and if the error file does not open, it is printed to the console. Declare the required variables. Then take the first line as input and check if it is a comment line. It takes them as input, prints them to an intermediate file, and updates the line numbers until the lines are comments. If the line is not a comment, check if the opcode is "START", update the line number, LOCCTR, and start address if found, and initialize the start address and LOCCTR as 0 if not found. .. Then use two nested while ()-. A loop in which the outer loop is repeated until the opcode is equal to "END", and the inner loop is repeated until the opcode is "END" or "CSECT". In the inner loop, check if the row is a comment. If you comment, it will be output to an intermediate file, the line numbers will be updated, and the next input line will be accepted. If it is not a comment, check if the line has a label. If so, check if it is in SYMTAB. If found, print a duplicate symbol error in the error file or assign it a name, address, and other required values. Insert it into a symbol and save it in SYMTAB. Next, check if the opcode exists in OPTAB. Check its format, if it exists, and increment LOCCTR accordingly. If not found in OPTAB, use other opcodes such as `WORD`,` RESW`, `BYTE`,` RESBYTE`, `LTORG`,` ORG, `BASE`,` USE`, `EQU`,` EXTREF. Check it. `Or` EXTDEF`. Therefore, insert the symbols, xrefs, and external definitions you created into the SYMTAB or control section map. For example, for opcodes like USE, add a new BLOCK entry to the BLOCK map as defined in the Utility.cpp file, and for LTORG, call the function handle_LTORG () defined in pass1.cpp. For `ORG`, specify the specified LOCCTR. Operand value. For EQU, check if the operand is an expression and use the evaluate [removed]) function to check if the expression is valid. If valid, pass the symbol to SYMTAB. Also, if the opcode does not match the above opcode, an error message will be thrown in the error file. Then update the data accordingly so that it is written to the intermediate file. After the control section while loop ends, update the CSECT_TAB, label, LOCCTR, start address, and length values and continue to the next control section until the outer loop ends. After the loop ends, save the length of the program and proceed to print the SYMTAB, LITTAB, and other control section tables (if any). Then proceed to pass2 ().

handle_LTORG() : Uses PassbyReference. By taking an argument from the *pass1()* function, the literal pool that existed up to that point is output. Run the iterator to print all the literals that exist in LITTAB, then update the line numbers. For some literals, if the address is not found, the current address is stored in LITTAB and the LOCCTR is incremented based on the existing literal.

evaluateExpression() : It uses PassbyReference. Use a While loop to get symbols from the formula. If you can not find an icon in symhtab, you will keep an error message in an error file. Use the variable PAIRCOUNT that holds the account when this expression is absolute or relative, and the pair count is an unexpected value. Display error message.

getString(): takes character as input and gives string output.

intToStringHex(): int to hex gives the hex as string output.

expandString(): Extend the input string to the specified input size. A string deployed as a parameter and length of the output string takes insert characters for deploying this string.

stringHexToInt(): converts the hexadecimal string to integer and returns the integer value.

stringToHexString(): takes in string as input and then converts the string into its hexadecimal equivalent and then returns the equivalent as string.

checkWhiteSpace(): checks if blanks are present. If present, returns true or else false.

checkCommentLine(): check the comment by looking at the first character of the input string, and then accordingly returns true if comment or else false.

if_all_num(): checks if all the elements of the string of the input string are number digits.

readFirstNonWhiteSpace(): takes in the string and iterates until it gets the first non-spaced character. It is a pass by reference function which updates the index of the input string until the blank space characters end and returns void.

writeToFile(): takes in the name of the file and the string to be written on to the file. Then writes the input string onto the new line of the file.
getRealOpcode(): for opcodes of format 4, for example +JSUB the function will see whether if the opcode contains some additional bit like '+' or some other flag bits, then it returns the opcode leaving the first flag bit.

getFlagFormat(): returns the flag bit if present in the input string or else it returns null string.

Class EvaluateString – contains the functions :

-peek()- returns the value at the present index.

-get()- returns the value at the given index and then increments the index by one.
-number()- returns the value of the input string in integer format.

TABLES :

It contains all the data structures required for our assembler to run. It contains the structs for labels, opcode, literal, blocks, extdef, extref, and control sections. The CSECT_Tab contains Maps are defined for various tables with their indices as strings with the names of the labels or opcodes as required.

UTILITY :

Contains useful functions that will be required by other files.

PASS2

pass2(): We take in the intermediate file as input using the *readIntermediateFile()* function and generate the listing file and the object program. Similar to pass1, if the intermediate file is unable to open, we will print the error message in the error file. Same with the object file if unable to open. We then read the first line of the intermediate file. Until the lines are comments, we take them as input and print them to our intermediate file and update our line number. If we get opcode as 'START', we initialize out start address as the LOCCTR, and write the line into the listing file. Then we check that whether the number of sections in our intermediate file was greater than one, if so, then we update our program length as the length of the first control section or else we keep the program length unchanged. We then write the first header record in the object program. Then until the opcode comes as 'END' or 'CSECT' if the control sections are present, we take in the input lines from the intermediate file and then update the listing file and then write the object program in the text record using the *textrecord()* function. We will write the object code on the basis of the types of formats used in the instruction. Based on different types of opcodes such as 'BYTE','WORD','BASE','NOBASE','EXTDEF','EXTREF','CSECT', we will generate different types of object codes. For the format 3 and format 4 instruction format, we will use the *createObjectCodeFormat34()* function in the pass2.cpp. For writing the end record, we use the *writeEndRecord()* function. If control sections are present, we will use the *writeRRecord()* and *writeDRecord()* to write the external references and the external definitions. For the instructions with immediate addressing, we will write the modification record. When the inner loop for the control section finishes, we will again loop to print the next section until the last opcode for 'END' occurs.

readTillTab(): takes in the string as input and reads the string until tab('\t') occurs.

readIntermediateFile(): takes in line number, LOCCTR, opcode, operand, label and input output files. If the line is comment returns true and takes in the next input line. Then using the *readTillTab()* function, it reads the label, opcode, operand and the

comment. Based on the different types of opcodes, it will count in the necessary conditions to take in the operand.

createObjectCodeFormat34(): When we get our format for the opcode as 3 or 4, we call this function. It checks the various situations in which the opcode can be and then taking into consideration the operand and the number of half bytes calculates the object code for the instruction. It also modifies the modification record when there is a need to do so.

writeDRecord(): It writes in the D record after the H record is written if the control sections are present.

writeRRecord(): It writes in the R record for the control section.

writeEndRecord(): It will write the end record for the program.
After the execution of the pass1.cpp, we will print the Tables like SYTAB, LITTAB, etc., in a separate file and then execute the pass2.cpp.

Data Structures used

1. Map

2. Struct

   Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. Structure(struct) is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.

   Map is used to store the SYMBOL TABLE, OPCODE TABLE, REGISTER TABLE, LITERAL TABLE, BLOCK TABLE, CONTROL SECTIONS.

   Each map of these tables contains a key in the form of string(data type) which represent an element of the table and the mapped value is a struct which stores the information of that element.

   Structures of each are as follows-

   SYMTAB

   The struct contains information of labels like name, address, block number, a character representing whether the label exits in the symbol table or not, an integer representing whether label is relative or not.

   OPTAB

   The struct contains information of opcode like name, format, a character representing whether the opcode is valid or not.

   LITTAB

   The struct contains information of literals like its value, address, block number, a character representing whether the literal exits in the literal table or not.

   REGTAB

   The struct contains information of registers like its numeric equivalent, a character representing whether the registers exits or not.

   BLOCKS

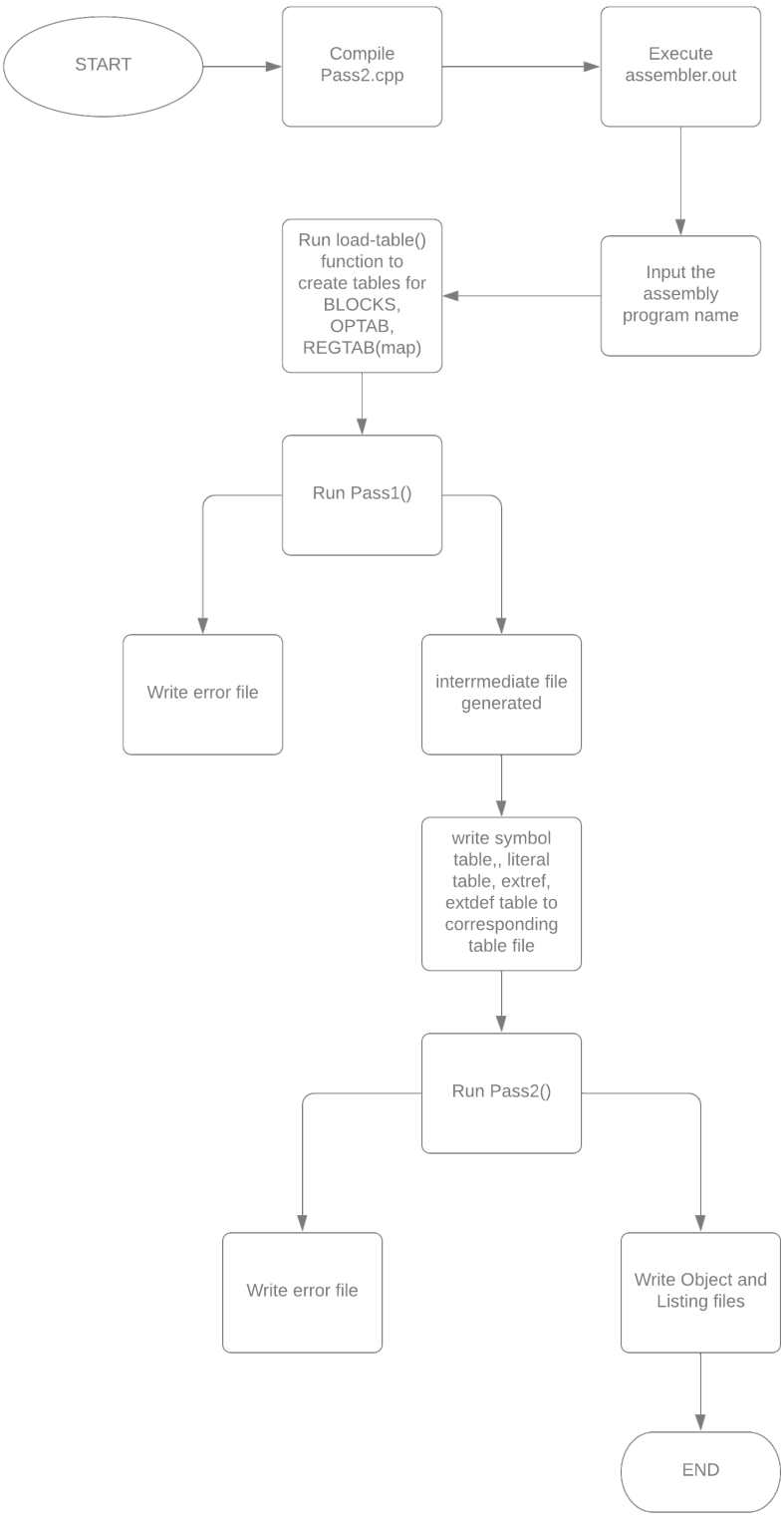
   The struct contains information of blocks like its name, start address, block number, location counter value for end address of block, a character representing whether the block exits or not.

   CSECT

   The struct contains information of different control section like its name, start address,section number,length, location counter value for end address of section.It also contains two maps for extref and extdef of particular section.

Design:

Control Flow:



Conclusion:

- All the requirements have been met succesfully.
- Verified the output object and listing files for the given test input.