

SIC-XE Assembler

20114047

Problem Statement:

Implement a two-pass assembler as presented in the book by Leland L. Beck.

Features Implemented:

The designed assembler includes all the SIC/XE instructions and supports all four formats of instructions in SIC/XE and addressing modes.

It also supports machine-independent features:

- Literals
- Symbol Defining Statements
- Expressions
- Program Blocks

Directory Structure

```
SIC-XE_assembler
| .gitignore
| assembler.java
| convert.java
| error.txt
| intermediate.txt
| listing.txt
| littab.txt
| object.txt
| pass1.java
| pass2.java
| program_blocks.txt
| README.md
| symtab.txt
| tables.java
| tc.txt
| tc1.txt
| tc2.txt
| tc_final.txt
|
└─ build
    .gitkeep
    assembler.class
    convert.class
    pass1.class
    pass2.class
    tables.class
```

Usage

- compile the code using:

```
javac -d build *.java
```

- cd into the build directory where the `.class` files are

```
java assembler ../path_to_the_tc_file
```

- to run these test cases use the command:

```
java assembler ../tc_name.txt
```

Working and Design the Assembler

Data Members

```
static int error_flag = 0;
// an integer value that represents the kind of error encountered in the code

/**
 * error flags:
 * 0 => no error
 * 1 => duplicate symbol
 * 2 => invalid opcode
 * 3 => invalid instruction format
 * 4 => invalid expression
 */

static String line = "";
// line read from the test case file

static String starting_address = "0";
// starting address of the code given by the start instruction

static Hashtable<String, String> LOCCTR = new Hashtable<String, String>();
// a hashtable of location counters one for each program block

static Hashtable<String, String> LOCCTR_next = new Hashtable<String, String>();
/** a hashtable of location counters pointing to the next instruction,
used to update LOCCTR one for each program block */

static String current_LOCCTR = "0";
// LOCCTR being used for the current program block

static Hashtable<String, ArrayList<String>> OPTAB = tables.OPTAB();
/** OPTAB stored as a hashtable with the <mnemonic OP CODE> as the key
and value an arraylist of OP CODE and format */

static ArrayList<String> MODRECORDS = new ArrayList<String>();
// stores the modification records to be written to the object file

static Hashtable<String, String> BLKDISP = new Hashtable<String, String>();
/** Stores the displacement for each block to be added to the address for
different program blocks while getting the object codes */

static Hashtable<String, String> ASSEMBDIR = tables.ASSEMBDIR();
// A hashtable that stores the list of Assemble Directives and a value "y" indicating its presence.

static LinkedHashMap<String, ArrayList<String>> LITTAB = new LinkedHashMap<String, ArrayList<String>>();
```

```
// A Literal Table hashmap with key as a literal and value <literal_value, size, location, program_block>

static LinkedHashMap<String, ArrayList<String>> BLOCKTABLE = new LinkedHashMap<String, ArrayList<String>>();
/** used to store info about the program block
    <block_name, block_idx, starting_location, length_in_bytes>    */

static Hashtable<String, ArrayList<String>> REGISTER = tables.REGISTER();
// contains information about the registers in SIC/XE, the id, size in bytes, and the value stored

static LinkedHashMap<String, ArrayList<String>> SYMTAB = new LinkedHashMap<String, ArrayList<String>>();
/** The symbol table which stores information about all the symbols encountered in the assembly of the code.
    <LABEL, expression_type(absolute/relative), address, program_block> */
```

Functions

```
##### tables.java #####
public static Hashtable<String, ArrayList<String>> OPTAB()
// returns the OPTAB as a hashtable

public static Hashtable<String, ArrayList<String>> REGISTER()
// returns register information as a hashtable

public static Hashtable<String, String> ASSEMDIR()
// returns assembler directive information as a hashtable

##### convert.java #####
public static int toDecDig(char a)
// returns the decimal equivalent of a hexadecimal digit

public static char toHexDig(int a)
// returns a hexadecimal digit for a decimal value

public static String toHalfBytes(String a)
// converts an ASCII string input to a hexadecimal string

public static String extendTo(int n_dig, String a)
// extends a string to the specified number of digits by padding with 0s

public static int HextoDec(String val)
// converts a hexadecimal string to its decimal value for arithmetic operations

public static String DectoHex(int val)
// converts an input decimal value to a hexadecimal string.

public static int BintoDec(String val)
// converts a binary string to a decimal value

public static String DectoBin(int val)
// converts a decimal value to a binary string

##### pass1.java #####
public static void appendError(String x) throws IOException
// append errors to errors.txt file

public static String get_label(String line)
// function that returns a label if present in the line

public static String get_opcode(String line)
// function that returns the opcode if present in the line

public static String get_operand(String line)
// function that returns an operand if present in the line
```

```

public static boolean is_comment(String line)
// boolean function to check if a line is a comment or not

public static void insert_symbol(String label, String locctr, String type)
// inserts a symbol into the symbol table

public static int getSize(String OPERAND)
// returns the size of a byte encoded string or a hexadecimal string

public static boolean is_end(String line)
// checks if the END assembler directive is encountered in the line

public static String remove_comment(String line)
// removes a comment from a line if present

public static boolean isConstant(String x)
// checks if an operand is a constant value

public static String getExpressionValue(String x)
// returns the value of an expression

public static String getExpressionType(String x) throws IOException
// returns the type of an expression if it is valid

public static boolean isValidExpression(String x) throws IOException
// checks if an expression is valid

public static String getValueOperand(String x)
// returns the operand value for literals

##### pass2.java #####
public static String[] getHeaderData(String first, String programLength) throws FileNotFoundException, IOException
// returns the data values necessary for writing the header record to the object file

public static ArrayList<String> getTokens(String l)
// tokenizes the lines read from the file

public static int charcount(String input, char ch)
// returns the count of a particular character found in a string

public static void addObjectCode(BufferedWriter bw_object) throws IOException
/** adds object code to a text record and checks if the text record needs to be written
to the object file. If necessary the text record is written and reinitialized to empty record */

public static void writeTextRecord(BufferedWriter bw_object) throws IOException
// the text record is written to the object file and is initialized to empty

public static boolean checkPCrel()
// checks if PC relative addressing is applicable

public static boolean checkBASERel()
// checks if base relative addressing can be applied

public static boolean isConstant()
// checks if the input operand is a constant value

public static ArrayList<String> getMultipleOperands(ArrayList<String> tokens)
// returns operands for instructions that require 2 operands

public static void setBLOCKDISP()
// sets the values for program block displacements

```

Pass 1

Pass 1 of the assembler takes as input the test case file to be assembled and output is an intermediate file that is used as an input in pass 2 of the assembler.

The input file is read line by line and is parsed to get the **LABEL**, **OPCODE**, and **OPERAND** based on if the line contains a label or not or if the instruction contains an operand or not.

The line is first checked for the **START** assembler directive and the starting address is set to the operand value, the location counter is set to the starting address and the first line is written to the intermediate file.

The program enters in a do-while loop which checks for the **END** assembler directive which when encountered the loop ends.

Each line is first checked for a comment which is written to the intermediate file and the execution continues. If the input line contains a label, the label is added to the symbol table.

Then the line is checked to contain an assembler directive which if true the assembler directive is processed and the line is written to the intermediate file and the execution continues. The **EQU** assembler directive requires checking if the expression is a valid expression or not and the type of expression ("**R**" / "**A**") is also added to the symbol table based on the rules. If an operand encountered in an expression is not a constant and is not found in the **SYMTAB**, a forward reference error is written to the error file.

Space is reserved for instructions **RESB**, **RESW**, **WORD**, **BYTE**, and the **LOCCTR** is incremented by the required number of bytes and for the **WORD**, **BYTE** instructions, the object code is also written to the intermediate file. When the assembler directive encountered is **LTORG**, the literals encountered which have not been written into the intermediate file are listed in a literal pool in the intermediate file.

If the instruction is found to contain a valid **OPCODE**, the instruction is written into the intermediate file along with the location counter (**LOCCTR**), the error flag, and the object code for the instruction.

The literal table, the block table, the literal table, and the symbol table are written into their corresponding files to be read from in Pass 2 of the assembler.

The length of the program given by the **LOCCTR** at the end of Pass 1 is returned to Pass 2 which is used for writing the header record.

Pass 2

The **SYMTAB**, **BLOCKTABLE**, and **LITTAB** are read from their files written in the first pass. The Block displacements are set in the **BLKDISP** table for lookup when changing the addresses for different program blocks symbols.

The first line is read from the intermediate file and the header record is written to the object file and the line is written to the listing file.

Each line read is first checked for a comment which if true is written to the listing file and the execution continues. If not the comments present in the line are removed and the line is tokenized.

The line is checked for the presence of an assembler directive which if found, the corresponding functions of the assembler directives are performed and object codes are generated for **WORD** and **BYTE** directives, and the object code is added to the current text record. If the directive is **RESW** or **RESB** then the current text record needs to be written to the object file and a new text record is initialized.

If a literal pool is encountered the information for the literal is fetched from the LITTAB and is written into the object and listing file.

The line is checked for a valid instruction format which if found the opcode, the format, and the operand if applies to the instruction (RSUB, FIX, and FLOAT do not have an operand). If none of these cases for instruction formats apply, an error is appended to the error file

```
no format exists for the given instruction in the OPTAB
```

The FIX, FLOAT, and RSUB instructions are handled separately and the execution continues. If not, then n, i, x, b, p, e flags are set.

Format 2 instructions are then handled where the **REGISTER** table is queried for the register number to add to the instruction. The object code is added to the text record and the line is written to the listing file. The location counter is set to the program counter and the execution continues.

If the operand is a constant value, the instruction is assembled according to the format. If format 3, then the constant is checked if it has a value in the range (-2048, 2047) and can be assembled as a 12-bit displacement value. If not then the assembly is done using a 20-bit operand format which if not applicable an error is appended.

```
the constant operand is out of bounds
```

The object code is written to the object file and the line is written to the listing file along with the object code.

Else for a format 3 instruction, the PC relative assembly is first tried if possible else the instruction is assembled using base relative addressing if base relative has been enabled and PC relative addressing is not possible for the given instruction and the displacement does not fit in 12 bits.

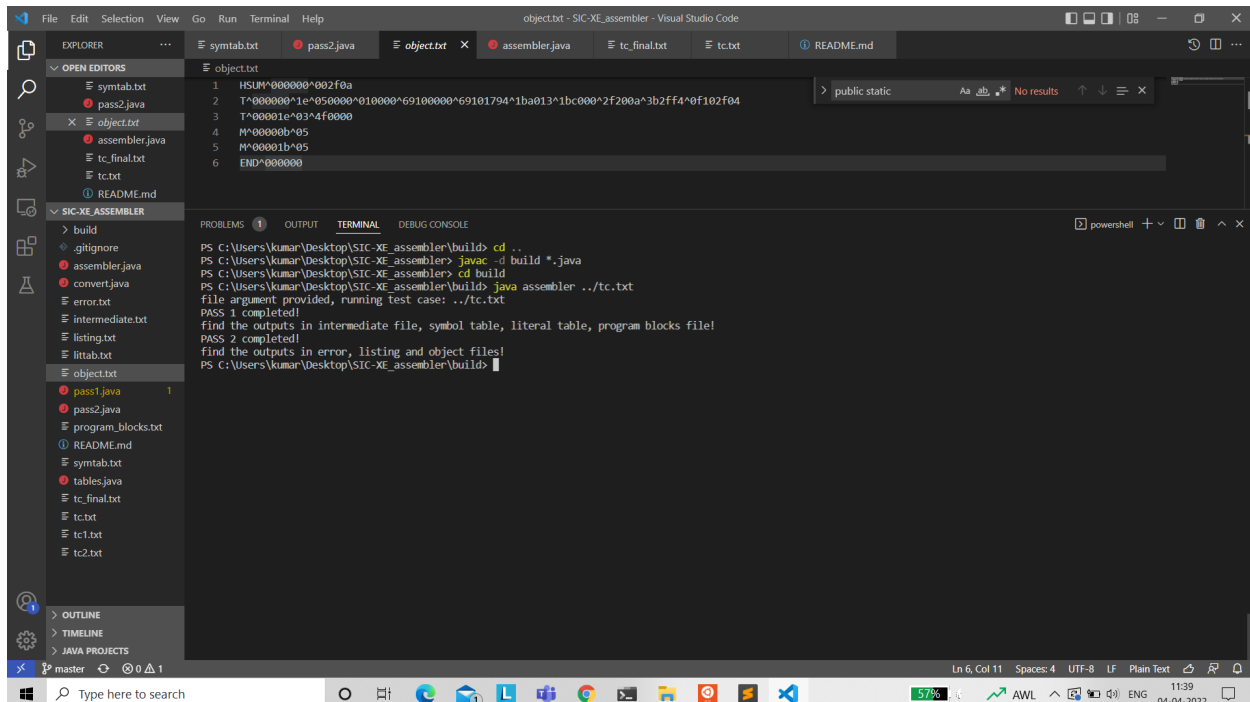
If both these formats are not applicable and format 4 is not explicitly mentioned then the assembly for the given instruction is not possible and an error is appended to the errors file.

```
unable to assemble using PC/BASE relative, specify extended mode explicitly
```

If the instruction is a format 4 instruction then a modification record is added to **MODRECORDS** apart from writing to the listing file and object file.

The last text record is written to the object file, the modification records are written to the object file and the **END** record is written to the object file.

Results



```
1 HSUM^000000^002f0a
2 T^000000^1e^050000^010000^69100000^69101794^1ba013^1bc000^2f200a^3b2ff4^0f102f04
3 T^00001e^03^4f0000
4 M^00000b^05
5 M^00001b^05
6 END^000000
```

```
PS C:\Users\kumar\Desktop\SIC-XE_assembler\build> cd ..
PS C:\Users\kumar\Desktop\SIC-XE_assembler> javac -d build *.java
PS C:\Users\kumar\Desktop\SIC-XE_assembler> cd build
PS C:\Users\kumar\Desktop\SIC-XE_assembler\build> java assembler ../tc.txt
file argument provided, running test case: ../tc.txt
PASS 1 completed!
find the outputs in intermediate file, symbol table, literal table, program blocks file!
PASS 2 completed!
find the outputs in error, listing and object files!
PS C:\Users\kumar\Desktop\SIC-XE_assembler\build>
```

test case file:

```
SUM START 0
FIRST LDX #0
  LDA #0
  +LDB #0
  +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 2
  END FIRST
```

intermediate file:

```
SUM START 0
0/0 0 FIRST LDX #0
3/0 0 LDA #0
6/0 0 +LDB #0
```

```

a/0  0      +LDB #TABLE2
      BASE TABLE2
e/0  0  LOOP ADD TABLE,X
11/0  0      ADD TABLE2,X
14/0  0      TIX COUNT
17/0  0      JLT LOOP
1a/0  0      +STA TOTAL
1e/0  0      RSUB
21/0  0  COUNT RESW 1
24/0  0  TABLE RESW 2000
1794/0  0  TABLE2 RESW 2000
2f04/0  0  TOTAL RESW 2
      END FIRST
. Program Length: 2f0a

```

assembly listing file:

```

SUM START 0
0/0  0  FIRST LDX #0          050000
3/0  0      LDA #0          010000
6/0  0      +LDB #0        69100000
a/0  0      +LDB #TABLE2   69101794
      BASE TABLE2
e/0  0  LOOP ADD TABLE,X   1ba013
11/0  0      ADD TABLE2,X 1bc000
14/0  0      TIX COUNT     2f200a
17/0  0      JLT LOOP      3b2ff4
1a/0  0      +STA TOTAL     0f102f04
1e/0  0      RSUB          4f0000
21/0  0  COUNT RESW 1
24/0  0  TABLE RESW 2000
1794/0  0  TABLE2 RESW 2000
2f04/0  0  TOTAL RESW 2
      END FIRST
. Program Length: 2f0a

```

object code:

```

HSUM^000000^002f0a
T^000000^1e^050000^010000^69100000^69101794^1ba013^1bc000^2f200a^3b2ff4^0f102f04
T^00001e^03^4f0000
M^000000b^05
M^00001b^05
END^000000

```

program blocks:

```

default  0  0  2f0a

```


Symbol Table:

SUM	R	0	0
FIRST	R	0	0
LOOP	R	e	0
COUNT	R	21	0
TABLE	R	24	0
TABLE2	R	1794	0
TOTAL	R	2f04	0