**ASSEMBLER**



1) Mohamed Nabil (62).

2) Nesma Emad El din (72).

3) Caroline Adel (49).

4) Amr Fathy Al dafrawy (48).

Systems Programming-1 Final-Project (phase-2)

Requirement Specification:

It is required to implement Phase-2 of a (cross) Assembler for (a subset of) SIC/XE machines.

Phase-2 specification requires the following:

1. Input:

Source file name "src.txt" via a (.exe).

1. Process:

The input source file is parsed in order to produce pass-1, after that the symbol table produced in phase-1 is used while forming phase-2 assembled file and object code.

The assembler for phase-2 should support:

1. EQU and ORG statements.

2. Simple expression evaluation. A simple expression includes simple (A

<op> B) operand arithmetic, where <op> is one of +,-,\*,/ and no spaces

surround the operation, eg. A+B.

1. Output:

The output of the assembler should include (at least):

1. Object-code file whose format is the same as the one described in the text

book in section 2.1.1 and 2.3.5.

2. A report at the end of pass2.

Pass1 and Pass2 errors should be included as

part of the assembler report, exhibiting both the offending line of source code and the error.

Design:

Pass-2 assembler:

* Source lines are read in sequence.
* The lines are passed to a parser method which supports free formatting.
* The parser method, along with other methods, is used as discussed before to output the pass-1 for the assembler and save the symbol table.
* If pass-1 was performed successfully pass-2 is then executed.
* In pass-2 every line entry is checked for producing the op-code.
* Errors in pass-2 are recorded so as to determine whether the object file will be created or not.

Algorithms Description:

The implementation is encapsulated into one class

assemble.cpp (as specified in the description) this class includes

1. Class Entry():

* This class entity represents the lines which are written to the output file.
* Its member variables are (int loc) to hold address of the current entry, and a set of strings (label, op\_code, operand, comment, error, ObjectCode) to describe entry.

1. A set of methods described below:
2. checkByte:

* Parameters:

A string which is the operand to the BYTE directive.

* Return type:

Returns an integer which is the value of the operand.

* Functionality:

Checks for byte declaration syntax, returns its value if correct syntax, -1 otherwise.

1. checkWord:

* Parameters:

A string which is the operand to the WORD directive.

* Return type:

Returns an integer which is the value of the operand.

* Functionality:

Checks for word declaration syntax, returns its value if correct syntax, -1 otherwise.

1. toDecimal:

* Parameters:

A string which is to be converted.

* Return type:

An integer which is the hexadecimal conversion of the string.

* Functionality:

Convert a string into hexadecimal notation.

* Description:

The result is added up via converting every character into its integer notation and then adding the correct weighting according to the decimal place of the digit (16 ^ weight).

1. toInteger:

* Parameters:

A string which is to be converted.

* Return type:

An integer which is the decimal conversion of the string.

* Functionality:

Convert a string into decimal notation.

* Description:

The result is added up via converting every character into its integer notation and then adding the correct weighting according to the decimal place of the digit (10^ weight).

1. toLower:

* Parameters:

String to be converted.

* Return type:

String after conversion.

* Functionality:

Convert all characters of the input string to lower case.

1. checkOperand:

* Parameters:

Two strings, the first is an opcode and the second is its corresponding operand.

* Return type:

A Boolean to denote the result of the check.

* Functionality:

To check whether the operand is a valid match for the opcode or not.

* Description:

It first checks for single operand instructions, as "tixr" and "clear", if their corresponding operand is a register true value is returned otherwise false.

Then it checks the format for opcodes that take 2 operands, it checks both are register names and they are separated by a comma, if so it returns true otherwise false.

1. validateOpcode:

* Parameters:

A string which is the opcode to be validated.

* Return type:

An integer value.

* Functionality:

Checks for the opcode in the opcode map, returns an integer value denoting the byte format for the given opcode if found, if the opcode is invalid it returns -1.

1. isDuplicateLabel:

* Parameters:

The string of which existence is to be checked for duplicity.

* Return type:

A Boolean to denote the check.

* Functionality:

Returns true if the label is a duplicate, false if not.

1. oneWord:

* Parameters:

A vector of strings containing parameters of a source code line of length 1, a vector of strings holding comment line and an integer denoting the format.

* Return type:

Void.

* Functionality:

Creates a new Entry instance and adds it up to the source code entry table.

Updates the current address with the correct value.

1. twoWord:

* Parameters:

A vector of strings containing parmeters of a source code line of length 2, a vector of strings holding comment line and an integer denoting the format.

* Return type:

Void.

* Functionality:

Creates a new Entry instance and adds it up to the source code entry table.

Updates the current address with the correct value.

1. threeWord:

* Parameters:

A vector of strings containing parmeters of a source code line of length 3, a vector of strings holding comment line and an integer denoting the format.

* Return type:

Void.

* Functionality:

Creates a new Entry instance and adds it up to the source code entry table.

If a possible error is encountered it is added.

Updates the current address with the correct value.

1. parse\_sic:

* Parameters:

String (line of the source code) to be parsed.

* Return type:

Void.

* Functionality:

Check line syntax, detect errors and convert the input into a form valid for later purpose (writing the list file).

Handles fixed format parsing.

* Description:

The input line is first checked for its type if It is a comment it is added as a comment, otherwise the parsing process both checks for syntax and starts separating line entries, converting them to lower case strings, validating opcodes and whether they match their corresponding operands in order to add the line as a new Entry.

It also checks that appropriate types of code are in their correct positions according to sic machine fixed formatting rules.

1. parse:

* Parameters:

String (line of the source code) to be parsed.

* Return type:

Void.

* Functionality:

Check line syntax, detect errors and convert the input into a form valid for later purpose (writing the list file).

Handles the free format parsing.

* Description:

The input line is first checked for its type if It is a comment it is added as a comment, otherwise the parsing process both checks for syntax and starts separating line entries, converting them to lower case strings, validating opcodes and whether they match their corresponding operands in order to add the line as a new Entry.

Only separates strings upon spaces since no kind of formatting specification is imposed in free formatting.

1. fillingMap:

* Parameters:

Void.

* Return type:

Void.

* Functionality:

Filling up the map holding the opcodes and their formats from an external text file.

1. checkEndLine:

* Parameters:

String which is the last line in the source code.

* Return type:

Void.

* Functionality:

Checks that the end line of the program is valid one.

1. checkSpace:

* Parameters:

String to be checked.

* Return type:

Boolean denoting check result.

* Functionality:

Checks for spaces in a string, returns true if at least one character was found, false if not (the line is all spaces).

1. checkOrg:

* Parameters:

String which is the operand for 'org' directive.

* Return type:

Int which is the address to bs assigned for the current line.

* Functionality:

1. checkEqu:

* Parameters:

String which is the operand for the 'equ' directive, and a string which is the label for the 'equ' directive.

* Return type:

Int which corresponds to the address to be assigned to this line.

* Functionality:

1. formate1:

* Parameters:

String which is an instruction of format 1.

* Return type:

String holding the corresponding opcode.

* Functionality:

A member of a set of functions designed to return an opcode for a certain instruction line according to its format used, format -1 is used in this case.

1. formate2:

* Parameters:

String which is the instruction, and another string which is the operand to an instruction of format 2.

* Return type:

A string holding the corresponding opcode.

* Functionality:

A member of a set of functions designed to return an opcode for a certain instruction line according to its format used, format -2 is used in this case, it reviews the registers -operands- and accordingly calculates the opcode.

1. formate3\_4:

* Parameters:

Integer which holds the index of the current line in the entries table,

String which is the instruction, and another string which is the operand to an instruction of format 2, and an integer holding the current address of the line.

* Return type:

A string holding the corresponding opcode.

* Functionality:

A member of a set of functions designed to return an opcode for a certain instruction line according to its format used, format -3 || 4 are used in this case, it reviews the instruction opcode, calculates the corresponding "nixbpe" values and evaluates the object code.

1. eval\_address:

* Parameters:

String which is the operand for a certain instruction.

* Return type:

Returns integer which is address for corresponding operand.

* Functionality:

Evaluates the address for all formats, it Is used to get the TA for a certain instruction whether from the symbol table or otherwise.

1. ObjectCode:

* Parameters:

Void.

* Return type:

Void .

* Functionality:

Method responsible for assigning objectcodes for all source code lines in order to be printed out In the main method in pass-2. This is only when pass-1 is error free.

1. buildObjectFile:

* Parameters:

Void.

* Return type:

Void.

* Functionality:

Method responsible for building up the object file if and only if pass-2 is error free.

1. modifyLocation:

* Parameters:
* Return type:
* Functionality:

Main Data Structures:

1. Maps:

A map data structure is used:

* For the symbol table:

It is used to insert labels in the symbol table and to easily be able to retrieve them or check for their duplicity in constant time.

* For saving up SIC/XE machine appendix:

A two dimensional map is used to save up SIC/XE machine instruction set and their corresponding format.

* For saving up SIC/XE machine instructions op-codes.

Assumptions:

1. Errors for pass-1 are produced as follows:

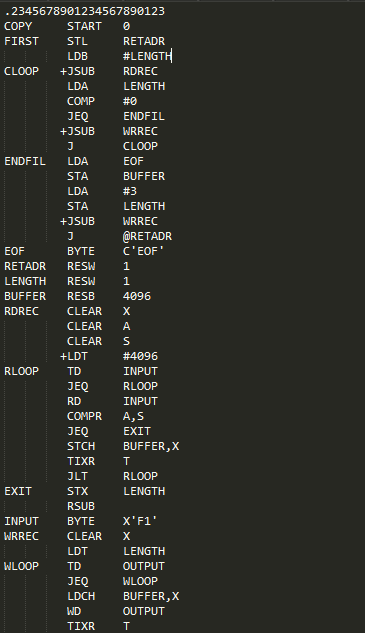
* In case of an invalid operand
* "\*\*\*\*Error: Invalid Operand".
* If line length is exceeded above limit
* "\*\*\*\*Error: Invalid length of the line".
* Invalid line spaces
* "\*\*\*\*Error: invalid spaces in this line".
* Invalid beginning
* "\*\*\*\*Error: invalid start of the program".
* Invalid op code
* "\*\*\*\*Error: Invalid OpCode".
* Duplicate symbols
* "\*\*\*\*Error: Duplicate Symbol".
* Invalid entry
* "\*\*\*\*Error: Invalid Entry".
* Invalid end program
* "\*\*\*\*Error: invalid end of the program".

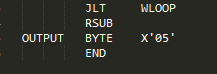
1. Errors for pass-2 are produces as follows:
2. Free format is used , not fixed format.
3. Operands and Labels cannot include space characters.

Sample Runs:

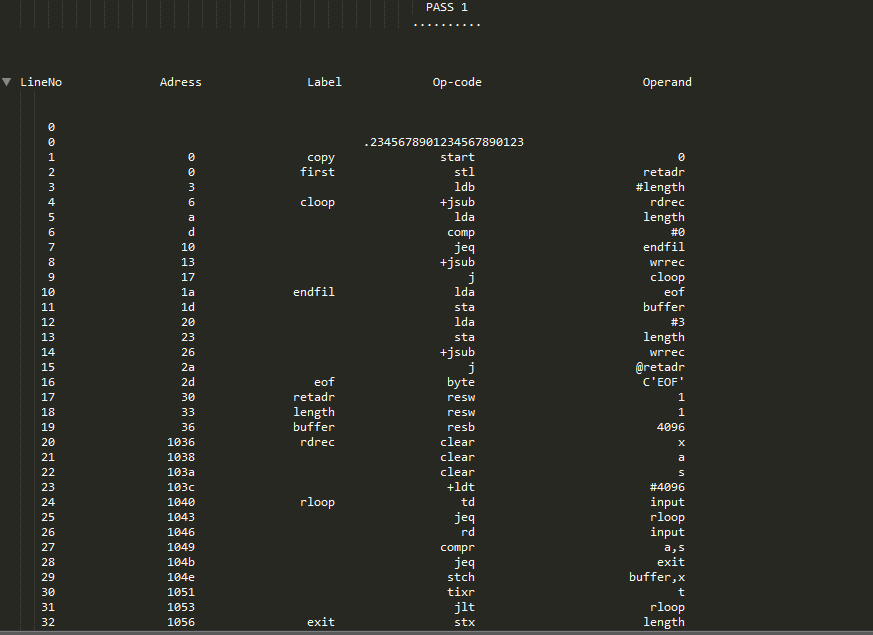
**First sample:**

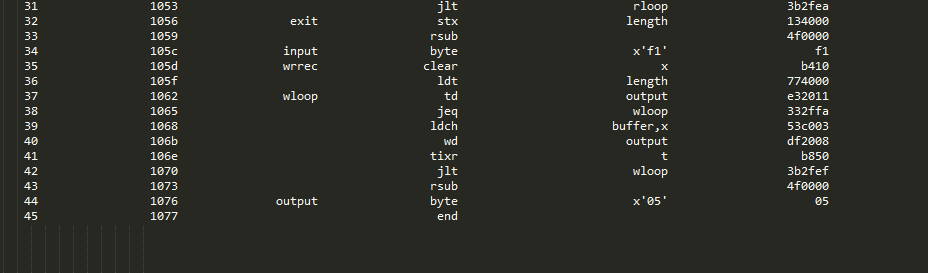
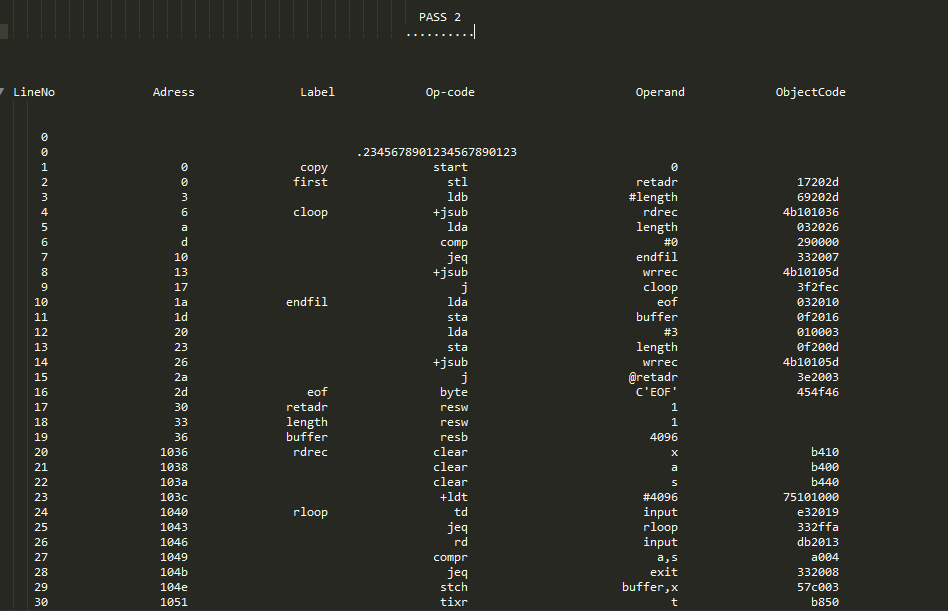
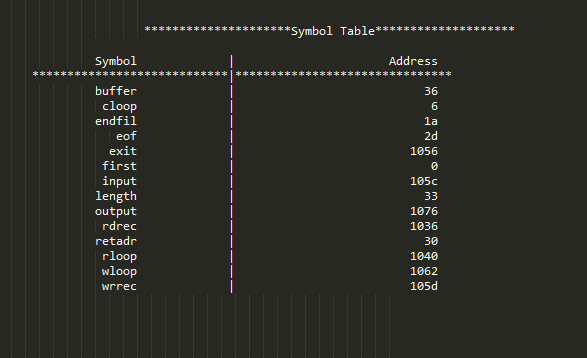
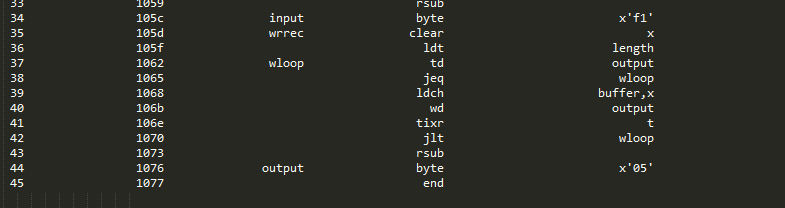
Input:



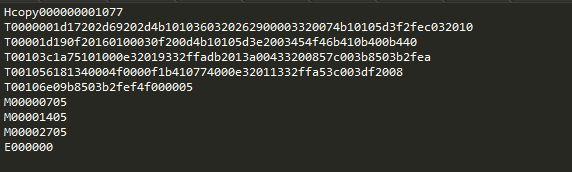


Output:



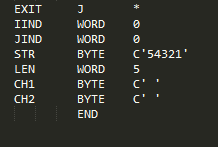
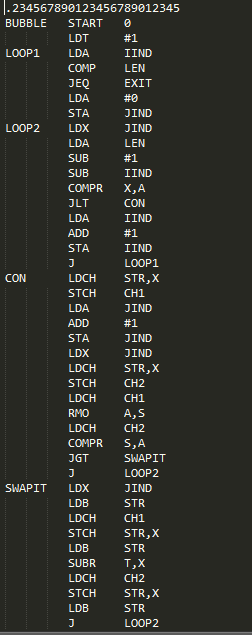


Object file:

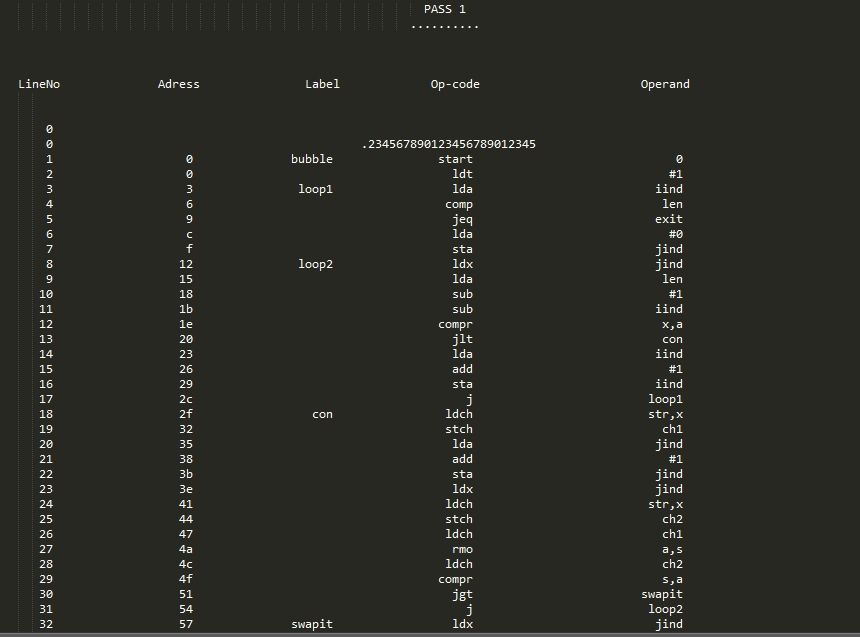


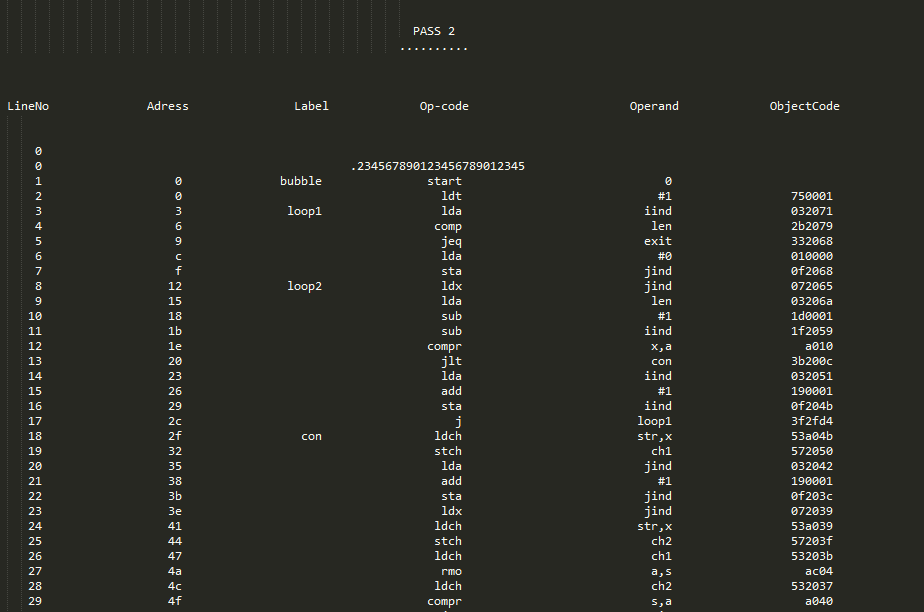
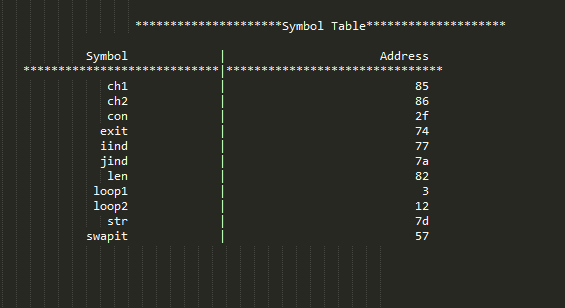
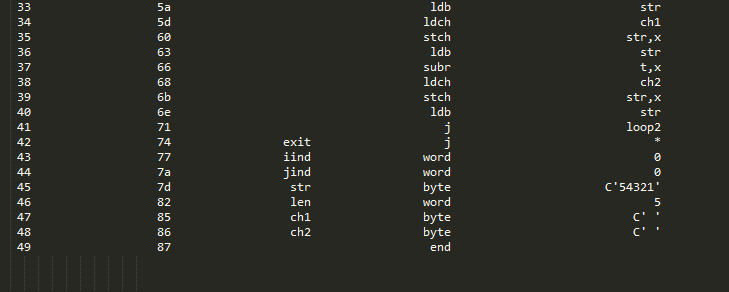
**Second sample:**

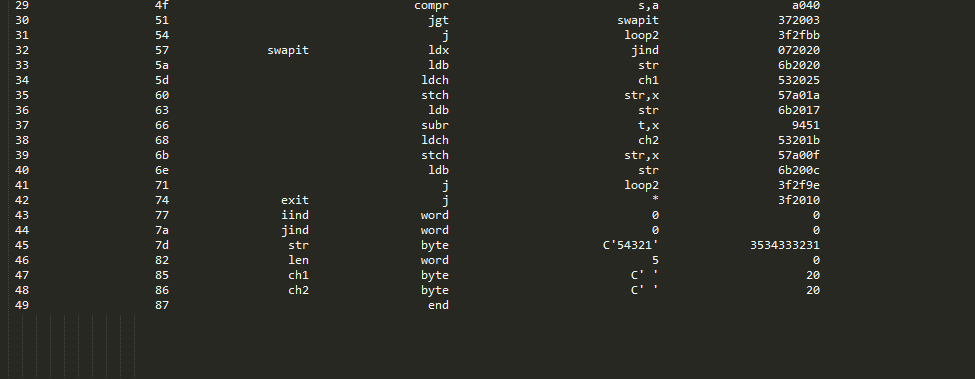
Input:



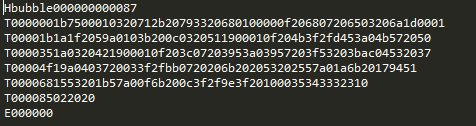
Output:





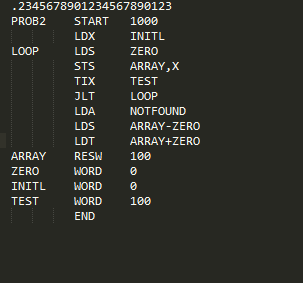


Object file:

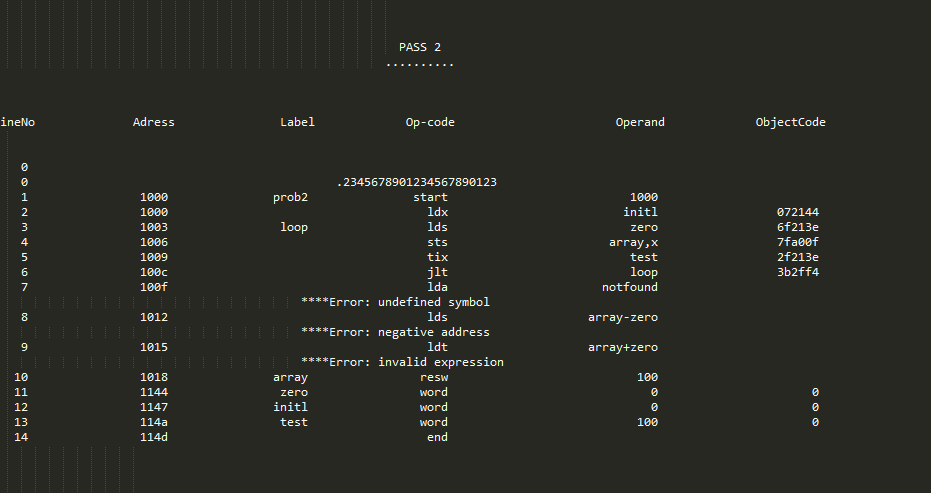
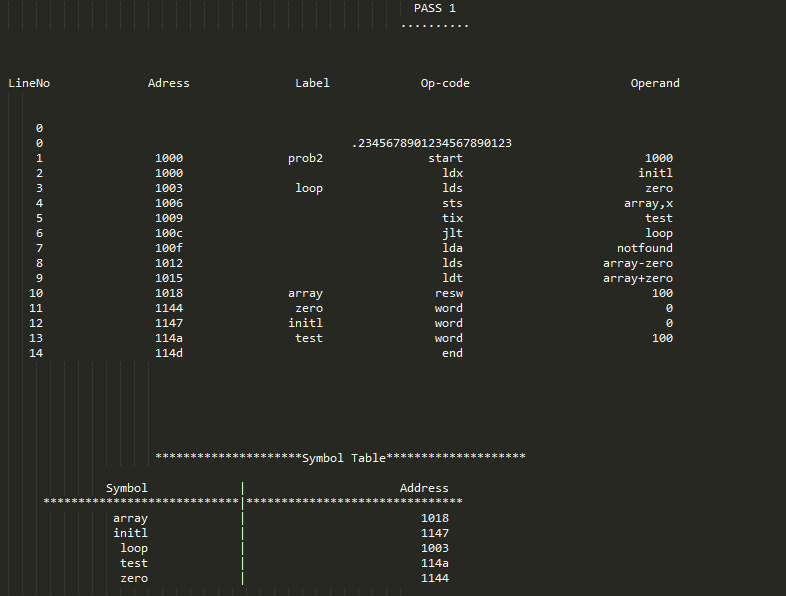
****

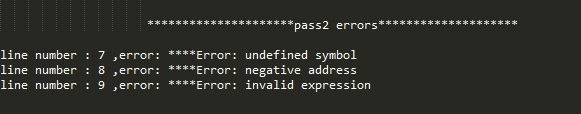
**Third sample:**

Input:



Output:





Object file:

There is no object file because there are errors in pass2.