**Name : Ahmed Ali Abd EL Megiud ID : 05**

**Name : Sherif Mohamed Mostafa ID : 20**

**Name : Hazem Morsy hassan ID : 16**

**Name : Bassam Rageh Ibrahim ID : 13**

phase 2

***Requirement Specifications:***

It is required to implement pass2 of the assembler producing code for absolute loader:

* Pass2 is executed by entering the source file name (assemble<source-file-name>).
* The output includes:
* Object – code file having the header, text and end records base on the object codes of the program instructions.
* The erroneous lines have the errors written beneath them.
* Supports EQU, ORG, BASE and LTORG statements.
* Supports simple expression evaluation in the operand fields.
* Supports literals including generation of literal pool at the LTORG statement.

***Assumptions:***

* The error report is replaced by the error beneath each erroneous line taking into consideration that if an error occurs during pass1, pass2 won’t be executed at all.
* The literals have a maximum size of 3 bytes (one word).
* The expressions having \* or / operations are erroneous as only + or – is allowed (according to the lecture slides).
* The literals object codes are written in text records before the end record even if they were placed after the end.
* The program length is calculated as the difference between the start and end locations taking into consideration the org statement that may alter the location counter.

***Language :***

Program is implemented in c++ language.

***Design :***

The Main function of program is pass2 function that is executed following pass1 working on the intermediate file generated after pass1 is completed:

* The program reads each line from the intermediate file skipping any comments or labeling lines and also skipping the symtab (stops at the statement pass1 completed).
* It doesn’t stop at the end statement in order to read all the literals even those after the end statement.
* It sets the starting location according to that obtained from pass1.
* It then parses each line obtaining the instruction address, operation and operand.
* Then the get object code function is called that obtains the object code for this instruction (to be covered).
* Then a vector for the text records is to be filled and we check for the need to produce a new record when:
* The maximum record size limit is reached.
* The location counter is changed without adding a new object code (statements as: ORG, RESW, RESB).
* Now it checks whether the operation was ORG or END to update the program length.
* After pass 2 is completed successfully we print all the records into the object file.
* In case any error occurred during pass2 the object file is not generated and the erroneous line has no generated object code.

***Main Data Structures and structs :***

**Structs:**

Data : it holds the data of each successfully parsed line

* Label.
* Operation.
* Operand.

operation: it holds the data of each operation

* Mnemonic : name of the operation .
* Operation code
* Format : it has three values

1. (2) for register to register operations (such as ADDR, SUBR,…..).
2. (3) for memory to register or register to memory or register to register operations (such as LDA,STA,RSUB,….)
3. (4) operations of format (3) but preceded by (+).

* Number of allowed operands.

**Maps:**

Map<string , integer> called sym\_tab : it stores the successfully parsed labels(string) as the key and the address of this label (integer)as the value, it also stores the literals and their addresses (instead of LITTAB so that the literal can be treated as a symbolic label for convenience) .

Map<string,operation> called op\_tab : it stores all the allowed operation’s mnemonic as the key and the operation struct of this operation as the value.

Registers\_tab: it stores each register with its corresponding address value.

**Vectors:**

Operandliterals: stores all the encountered literals before the END or LTORG statements and is cleared when the pool is generated.

Absolute\_labels: stores the labels having an absolute address due to the EQU statement having an expression resulting in an absolute address.

Textrecord:vector having the current text record being concatenated upon.

Allrecords: vector having all the generated text records.

***Main Algorithms and Functions:***

**Parsed\_intermediate\_data** :

* The function that takes the input line from the intermediate file and performs the needed processing.
* It sets the current address for the instruction, current label, current operation and current operand.
* It processes literals differently if the label encountered was “\*” indicating a label.
* If the directive base was met it sets the BASE value for base relative addressing.
* Then using the address as the current location counter, we calculate the current PC value for PC relative addressing.

**Get\_objcode:**

* Given the output from the parsing process, this functions generates the complete object code of the instruction.
* It handles the directives WORD and BYTE alone as the object codes of these directives are according to their operands only.
* It handles literals also alone as their object code is similar to that of operands of WORD and BYTE.
* Otherwise, it generates object codes for the operation, flags and memory and concatenates them.
* If an error occurs or the instruction is a directive other than WORD and BYTE no object code is returned.

**Get\_opcode:**

* This functions returns the operation code for the instruction.
* It adds the values of the flags n,I to the operation code of the instruction mnemonic

and then returns it.

**Get\_flags:**

* This function returns the hexa code for the flags x,b,p,e.
* Starting wth a value 0 it adds each flag value correspondingly (if the flag was set its value is added).
* It then returns the code for the flags.

**Get\_memory:**

* This function calculates and returns the memory code of the operand for the object code.
* It has many cases:
* if the operation is of format 2 then its operands are one or two registers, so the object code is the concatenation of their address values.
* If no operand exists (RSUB) then it returns zeros.
* We then obtain the total size of the memory code according to the operation format (3 or 4).
* If the operand is an expression we evaluate the expression value and process the existence of any error (errors are to be covered later).
* If the operand is “\*” then the target address is the location counter.
* Otherwise the target address is set according to the value of the operand whether it is a symbol or a numerical value.
* In case of immediate numerical or direct addressing the returned is the target address.
* Other wise we try PC relative addressing if it is valid the displacement is calculated and returned.
* If PC relative wasn’t valid we try BASE relative addressing if it was valid the displacement is calculated and returned.
* If BASE relative also was invalid then an error has occurred.
* This function returns no memory code In case of any error.

***Errors handeled in pass2:***

* Undeclared operand.
* Length of WORD is out of bounds.
* Address out of both PC and BASE relative addressing ranges.
* Address is out of range (3 hexa for format 3 and 5 hexa for format 4).
* Invalid literal format.
* Not a decimal literal.
* Not a valid hexadecimal literal.
* Odd length of hexa string.
* Literal length out of bounds (defined as 1 word).
* Cannot have a negative relative address In the expression.
* Negative address is invalid.
* Cannot add two relative addresses.

***Division of Work :***

Bassam : Text record.

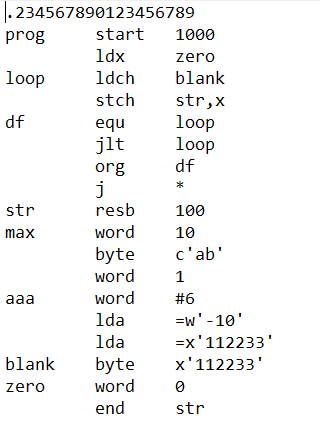
Ahmed : pass2.

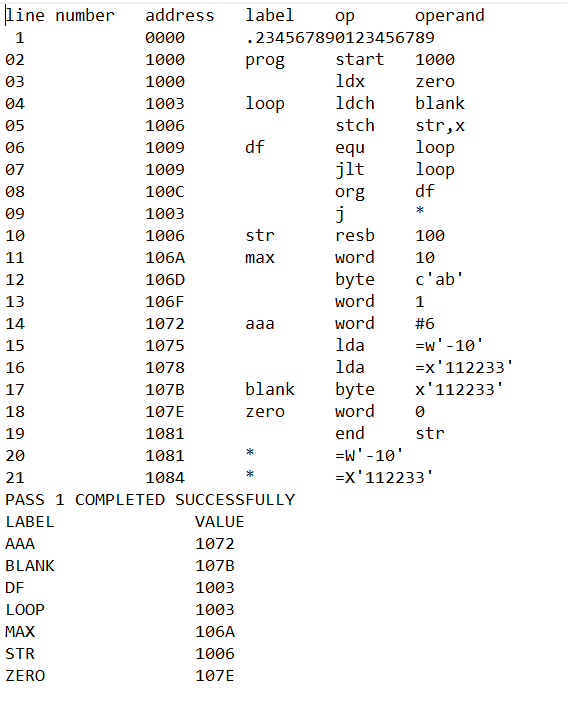
Sherif : object code processing.

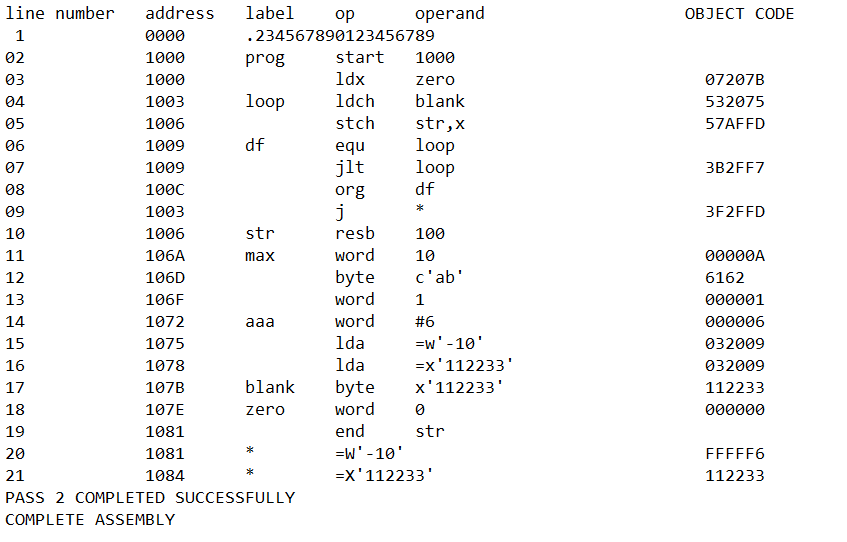
Hazem : parsing , pass2, text records, reading and writing to file.

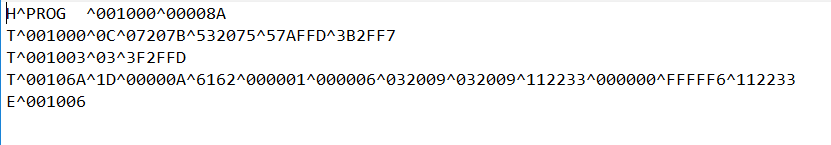
***Sample runs :***

**Example 1 :**

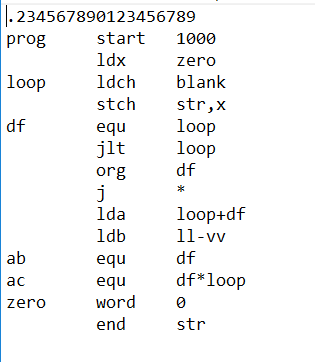


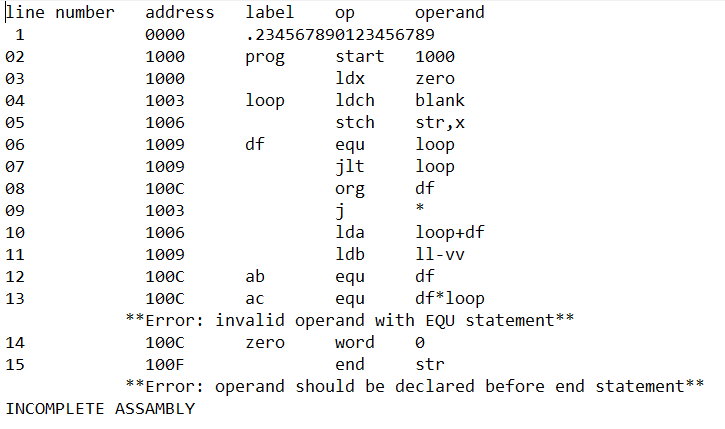




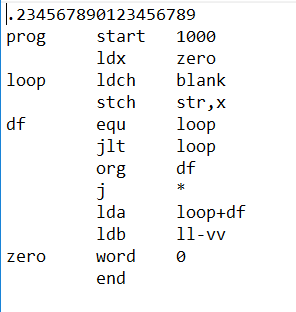


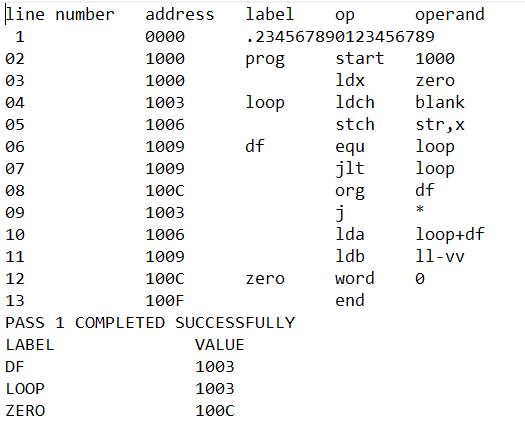
**Example 2:**

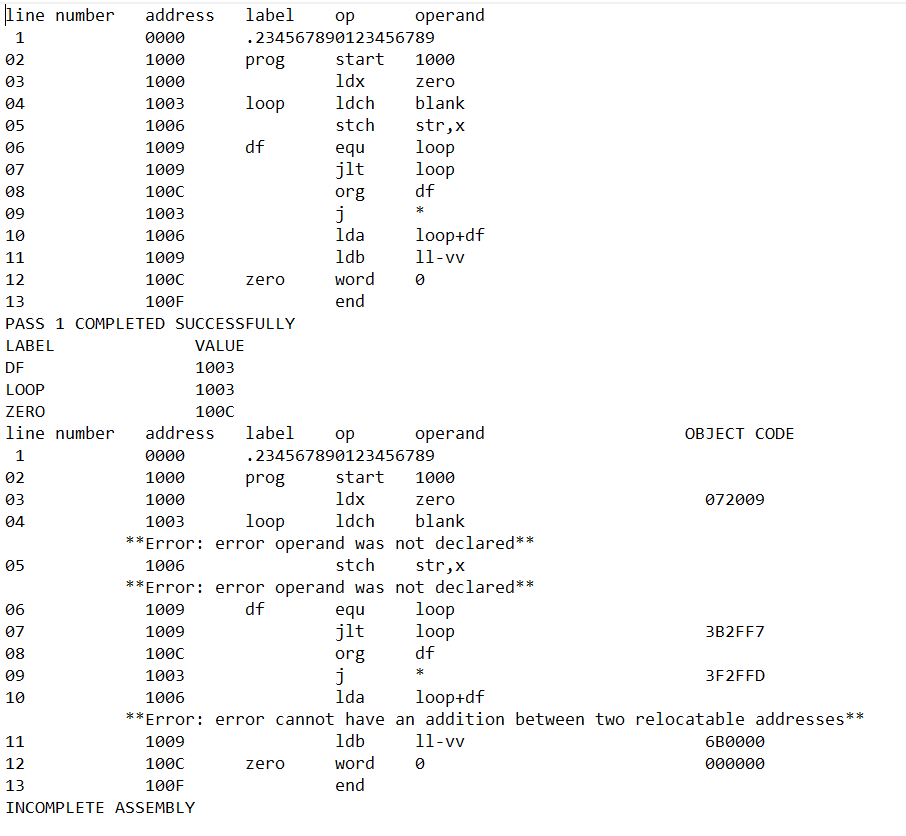




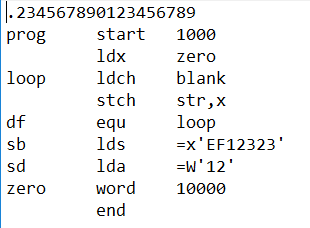
**Example 3:**

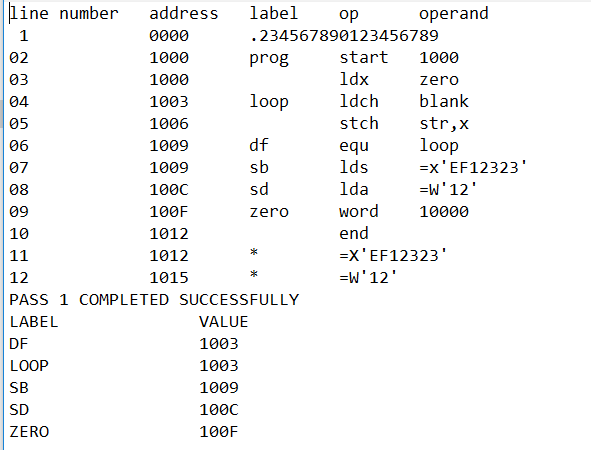


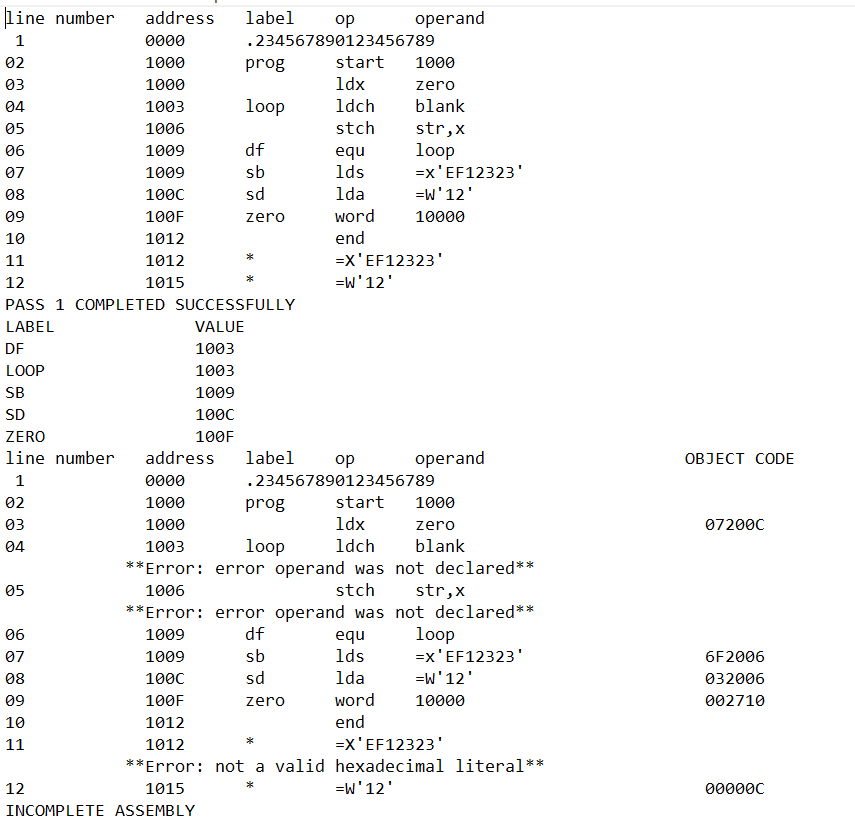




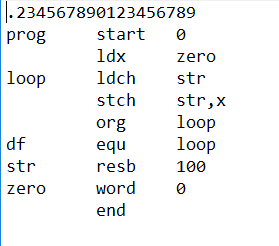
**Example 4:**

****





**Example 5:**

****

