CS 222: Assembler Phase#1

Due on Sunday, May 18, 2014

Mahmoud A.Gawad

Contents

Requirements Specification	
Design	3
Main Data Structures	4
Algorithms Description	4
Assumptions	5
Sample Runs	5

Requirements Specification

The term project is to implement a (cross) assembler for (a subset of) SIC/XE assembler, written in C/C++, producing code for the absolute loader used in the SIC/XE programming assignments. In phase 1 of the project, it is required to implement Pass1 of the assembler. The output of this phase should be used as input for subsequent phases.

- 1. The pass1 is to execute by entering pass1 <source-file-name>
- 2. The source file for the main program for this phase is to be named "pass1.c".
- 3. You should build a parser that is capable of handling source lines that are instructions, storage declaration, comments, and assembler directives (a directive that is not implemented should be ignored possibly with a warning)
 - (a) For instructions, the parser is to minimally be capable of decoding 2, 3 and 4-byte instructions as follows:
 - i. 2-byte with 1 or 2 symbolic register reference (e.g., TIXR A, ADDR S,A)
 - ii. RSUB (ignoring any operand or perhaps issuing a warning.)
 - iii. 3-byte PC-relative with symbolic operand to include immediate, indirect, and indexed addressing.
 - iv. 3-byte absolute with non-symbolic operand to include immediate, indirect, and indexed addressing.
 - v. 4-byte absolute with symbolic or non-symbolic operand to include immediate, indirect, and indexed addressing.
 - (b) The parser is to handle all storage directives (BYTE, WORD, RESW, and RESB).
- 4. The output of this phase should contain (at least):
 - (a) The symbol table.
 - (b) The source program in a format similar to the listing file described in your text book except that the object code is not generated as shown below. A meaningful error message is printed below the line in which the error occurred.

Design

The Design is consisting of 4 main modules (Control Unit-Parser-Validator-HashTable):

- 1. ControlUnit: This module is responsible of the following:
 - (a) Reading the operations file, and storing the operations in a *HashTable*.
 - (b) Reading the file line by line, and sending it to the *Parser*, then sending the parsed line (if it's not a comment..etc) to the *Validator*.
 - (c) Assigning addresses to the instructions based on the response of the *Validator* concerning the instruction's size.
 - (d) Writing a new formated source file with addresses of each instruction.
 - (e) Writing the symbol table based on the HashTable created in the Validator.
- 2. Parser: This module is responsible of parsing an instruction line to its components(label, operation, and operand).

- 3. Validator: This module is responsible of validating an instruction line, and generally do the following:
 - (a) Checking the syntax of the operation field (e.g. LDA from LDZ).
 - (b) Assigning addresses to the instructions and labels.
 - (c) Storing the symbols in a HashTable with their assigned addresses.
- 4. *HashTable*: This module is a general-purpose mapping module that can handle any type of <Key, Value> pairs. This module is used twice in this phase: In Operation Table, and Symbol Table.

Main Data Structures

The following data structures are used:

- Handmade Hashtable.
- Vector.

Algorithms Description

- \bullet Algorithm in Parser:
 - 1. Splitting the line by space into a vector of strings.
 - 2. Checking for comments and check for "RSUB."
 - 3. If it is neither a comment nor RSUB then,
 - 4. check for existance of comma ',' at the operand field.
 - 5. If a comma is not found then the operand field can be detected and operation field can be detected, then checking if the statement has a label or not.
 - 6. Else if a comma is found, then specify its type i. Avaliable types are:
 - (a) "STR,X"
 - (b) "STR(space(s)),X"
 - (c) "STR,(space(s))X"
 - (d) "STR(space(s)),(space(s))X"
 - 7. For every type there's exist a method to find its operand, operation, and label if exists.
- \bullet Methods in Validator:
 - checkSyntax (label, operation, operand):
 - 1. If the instruction contains a label, then check if the label is valid using the appropriate method.
 - 2. If the label is correct then it will check if the operation is valid, if it is found in the operation Table then it is valid, if not then it is not valid, if the information is valid then it will get all the information and check if it is valid.
 - 3. If the instruction is valid then it will check its operand and index modes.
 - isValid(): This method is used by the controller to check if the instruction is valid.
 - getAddress(): This method is used to get a string containing the error(empty string if the instruction is valid).
 - checkLabelSyntax(): This method is used to check if the label name is valid.
 - checkOpernadSyntax(): To check if the operand name is valid.
 - checkDirectiveOpernadSyntax(): If the operation is a directive word this method is used to check
 if it's correct.

Assumptions

- Assumptions in *Parser*:
 - 1. No operation other than RSUB takes no operand else it works with standard format of SIC/XE.
 - 2. Free format is allowed when leaving any number of spaces between the label and operation or between operation and operand.
 - 3. Free format will be allowed when entering two registers separated by comma at oprand field and all kinds of this case are handleed.
 - 4. A comment-line always starts (possible after some blank spaces) with "."
 - 5. If at any stage the splitted words were too much, then this line is invalid from *Parser*'s point of view.

\bullet Assumptions in Validator:

- 1. the label name can't start with a number, it must contain characters , it must not contain any valid characters like (@, #, +, *....etc), and it can't be the name of instruction.
- 2. Invalid index modes indexing can't be used with immediate or indirect addressing.
- 3. The address next to the directive word(START) must be valid hexadecimal digit.
- 4. The address can't be greater than (1048576) integer.
- 5. In case of the directive BYTE:
 - If the operand start with (X) then it is a hexadecimal number and the number of the digit must be even number.
 - If the operand start with (C) then it is a string.
- 6. If the instruction needs two register then the operand must be two valid registers.
- 7. User can't define the label more than once.
- 8. If the instruction is invalid then the address of this instruction will be the same as the previous instruction.
- 9. If the user didn't specify the start of the program then it will be zero.

Sample Runs

Page 5 of 7

1	.23456789012345678901234567890			
2		START	1000	
3	ALPHA	LDT	#10	
4	BETA	+LDCH	#6	
5		ADD	@GAMMA	
6		J	*	
7	GAMMA	BYTE	X'01'	
8	THETA	RESB	2	
9		END		
10				

Figure 1: Test 1

```
>> Source Program statements with value of LC indicated
       .23456789012345678901234567890
   001000
   001000 ALPHA
               LDT
                      #10
   001003 BETA
               +LDCH #6
   001007
               ADD
                      @GAMMA
   00100A
   00100D GAMMA
               BYTE
                      x'01'
   00100E THETA
                RESB
                      2
11
   001010
13
14
        e n d
               gf pass 1
       symbol table (values in decimal)
                  value
                   4096
20
21
         ALPHA
         BETA
                   4099
         GAMMA
                    4109
         THETA
```

Figure 2: LISAFILE Test 1

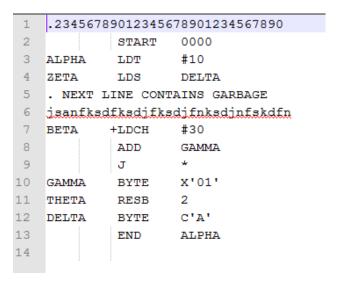


Figure 3: Test 2

```
1 >> Source Program statements with value of LC indicated
         .23456789012345678901234567890
   START 0000
000000 ALPHA LDT #10
000003 ZETA
 4 000000
6 000003 ZETA
                          DELTA
     . NEXT LINE CONTAINS GARBAGE
8 jsanfksdfksdjfksdjfnksdjnfskdfn
9 ****** unrecognized operation code
10 000006 BETA +LDCH #30
   A00000
                  ADD
                          GAMMA
   00000D
12
                  J
13
   000010 GAMMA BYTE X'01'
                 RESB 2
BYTE C'A'
   000011 THETA
15 000013 DELTA
16
17
   000014
                  END
                          ALPHA
18
19 >>
       incompletly assembly
20
```

Figure 4: LISAFILE Test 2

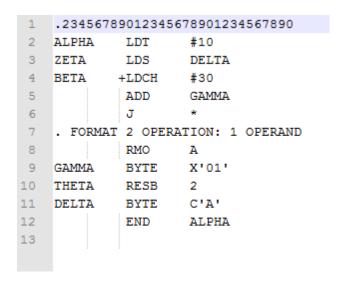


Figure 5: Test 3

```
1 >> Source Program statements with value of LC indicated
        .23456789012345678901234567890
  000000 ALPHA LDT #10
   000003 ZETA
                 LDS
                       DELTA
   000006 BETA
                +LDCH #30
   00000A ADD
                        GAMMA
   00000D
                J
  . FORMAT 2 OPERATION: 1 OPERAND 00000D RMO n
   not valid register !
                      X'01'
   000010 GAMMA BYTE
13
   000011 THETA
                 RESB
   000013 DELTA BYTE C'A'
   000014
                END
                       ALPHA
      incompletly assembly
19
```

Figure 6: LISAFILE Test 3