CS 222: Assembler Phase#1

Due on Sunday, May 18, 2014

Mahmoud A.Gawad

Contents

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. This module is 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

Besides the **Handmade** *HashTable*, the following data structures are used:

- Vector.

Algorithms Description

Assumptions

- A comment-line always starts (possible after some blank spaces) with ”.”

Sample Runs