Assignment 1

Sakif Fahmid Zaman
B00756635
Dalhousie University
ECED 3403

July 23, 2019

Contents

1 Source Files

1.1 cmake

```
cmake_minimum_required(VERSION 3.13)

set(CMAKE_CXX_STANDARD 17)
set(C_STANDARD 18)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
set(CMAKE_CXX_FLAGS "-O3 -Wall")

project(assembler)

add_executable(assembler Globals.cpp Pass1.cpp main.cpp)
```

1.2 Main

```
#include "Globals.h"
#include "Pass1.h"

int main(int argc, char *argv[]) {

  init_globals(argv[1]);
  if (Pass1(argv[1])) {
    cout << "Pass1 done" << endl;
  }
}
```

1.3 Instructions

```
BL 000 a
BEQ 001000 l
BZ 001000 l
BNE 001001 l
BNZ 001001 l
BC 001010 l
BHS 001010 l
BNC 001011 l
BLO 001011 l
BN 001100 l
BGE 001101 l
BLT 001110 l
BRA 001111 l
ADD 01000000 v,r
ADD.B 01000000 v,r
ADD.W 01000000 v,r
ADDC 01000001 v,r
ADDC.B 01000001 v,r
ADDC.W 01000001 v,r
SUB 01000010 v,r
SUB.B 01000010 v,r
SUB.W 01000010 v,r
SUBC 01000011 v,r
SUBC.B 01000011 v,r
SUBC.W 01000011 v,r
```

```
DADD 01000100 v,r
DADD.B 01000100 v,r
DADD.W 01000100 v,r
CMP 01000101 v,r
CMP.B 01000101 v,r
CMP.W 01000101 v,r
XOR 01000110 v,r
XOR.B 01000110 v,r
XOR.W 01000110 v,r
AND 01000111 v,r
AND.B 01000111 v,r
AND.W 01000111 v,r
BIT 01001000 v,r
BIT.B 01001000 v,r
BIT.W 01001000 v,r
BIC 01001001 v,r
BIC.B 01001001 v,r
BIC.W 01001001 v,r
BIS 01001010 v,r
BIS.B 01001010 v,r
BIS.W 01001010 v,r
MOV 01001011 v,r
MOV.B 01001011 v,r
MOV.W 01001011 v,r
SWAP 01001100 r,r
SRA 010011010 r
SRA.B 010011010 r
SRA.W 010011010 r
RRC 010011100 r
RRC.B 010011010 r
RRC.W 010011010 r
SWPB 0100111100000 r
SWPB.W 0100111100000 r
SXT 0100111110000 r
SXT.W 0100111110000 r
SVC 010110000000 s
LD 010100 p,r
LD.B 010100 p,r
LD.W 010100 p,r
ST 010101 r,p
ST.B 010100 r,p
ST.W 010100 r,p
CEX 010111 c,t,t
MOVL 01100 b,r
MOVLZ 01101 b,r
MOVLS 01110 b,r
MOVH 01110 b,r
LDR 10 r,o,r
LDR.B 10 r,o,r
LDR.W 10 r,o,r
STR 11 r,r,o
STR.B 11 r,r,o
STR.W 11 r,r,o
```

1.4 Globals

Header

```
#ifndef _GLOBALS_H
#define _GLOBALS_H

#include "stdio.h"
#include <algorithm>
#include <cctype>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <map>
#include <sstream>
#include <stdlib.h>
#include <string>
#include <vector>

#define INVALID_INDEX -1
#define BYTE_MIN 0
#define BYTE_INCREASE 1
#define WORD_INCREASE 2

using namespace std;

enum Operand_T { //operands tyoe
  IDR,           //pre/post increment decrement register - denoted as 'p' in instruction file
  R,             //register (no, pre/post +/-) - 'r'
  OFFSET,        //offset value - signed number with range [-64,63] - 'o'
  L13,           //label with 13-bit offset - 'a'
  L10,           //label with 10-bit offset - 'l'
  BYTE,          //byte value - 'b'
  CON_R,         //register or fixed/constant value [0,1,2,8,16,32,-1] - 'v'
  SA,            //vector value [0-15], 's'
  COND_CEC,      //cec value - 'c'
  TCFC           //TC/FC type [0-7] - 't'
};

enum Error_T {
  NO_ERR = 0,                       //no error :-)
  MISSING_OPERAND,                  //missing operand, eg. instruction/directive without required operand
  ILLEGAL_OPERAND,                  //unwanted operand, eg, operand after ALIGN
  NUMBER_OF_OPERANDS_MISMATCH,      //too many ortoo few operands compared to expected
  INVALID_OPERAND,                  //operand present but invalid. eg, byte value is not in byte range
  INVALID_REGISTER,                 //expected register operand but got something else, possibly undefined
  MISSING_INSTRUCTION_DIRECTIVE,    //token following a LBL must be either an INST or a DIR
  INVALID_LABEL_FORMAT,          //label format is not valid
  UNDEFINED_SYMBOL,              //Symbol(label or register) is not defined  - might be capyured at the end of first pass/ in 2nd pass??
  DUPLICATE_LABEL,               //duplicate label in symbol table
  INVALID_NUMBER,                //the string is not a valid number - compare with INVALID_OPERAND???
  INVALID_RECORD                 //too many tokens
};

struct Symbol {
  string name; //name of the symbol
  string type; //SymbolType type; //label (LBL), register (REG) or unknown (UNK)
  int value;   //value - decimal
};

struct Inst {                             //instruction
  string mnemonic;                        //mnemonic of the instruction
  string opcode;                          //most significant binary bits of this mnemonic -stored as string to join rest of the bits to create full opcode
  vector<Operand_T> expected_operands; //rule for the expected operands for this instruction

  //Inst(string m, string o, vector<Operand_T> e):mnemonic(m), opcode(o), expected_operands(e){}
};

//global vars
extern short int loc_counter;
extern bool has_error;
extern vector<Inst> inst_set; //instruction set - global scope
extern ofstream ofs;
const vector<string> directives = {"ALIGN", "BSS", "BYTE", "END", "EQU", "ORG", "WORD"};
enum directiveIndexes {
  dirALIGN,
  dirBSS,
  dirBYTE,
  dirEND,
  dirEQU,
  dirORG,
  dirWORD
};

const vector<string> cecs = {"EQ", "NE", "CS", "HS", "CC", "LO", "MI", "PL", "VS", "VC", "HI", "LS", "GE", "LT", "GT", "LE", "AL"};
const vector<unsigned short int> cec_values = {0, 1, 2, 2, 3, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
extern vector<Symbol> sym_tab;

//functions
bool is_numeric(string s);
Error_T str2int(string s, short int &value);
```

```
short int is_register(string s);
void get_tokens(string &l, vector<string> &toks);
short int check_if_instruction(string s);
short int check_if_directive(string s);
bool is_valid_label_name(string lbl);
short int is_label_in_sym_tab(string lbl);
short int is_cond(string s);

void init_globals(string src_fname, string inst_fname = "instructions.txt"
    );

#endif //_GLOBALS_H
```

Source

```
#include "Globals.h"

short loc_counter;
bool has_error;
vector<Inst> inst_set;
vector<Symbol> sym_tab;
ofstream ofs;

/* check if the string has format of Numeric type operand */
bool is_numeric(string s) {
  if (((s.front() == '#') || (s.front() == '$')) || ((s.front() == '\'')
      && (s.back() == '\''))) {
    return true;
  } else {
    return false;
  }
}

/* converts numeric type strings to corresponding integer value, error is
     captured and returns flase then */
Error_T str2int(string s, short int &value) {
  Error_T err = NO_ERR;
  if (s.front() == '#') { //integer value
    s.erase(0, 1);         //remove leading '#'
    value = stoi(s);
  } else if (s.front() == '$') { //hex value
    s.erase(0, 1);                //remove leading '$'
    value = stoi(s, nullptr, 16);
  } else if (s.front() == '\'' && s.back() == '\'') { //char - could be
      escaped or alpha-numeric
    unsigned int result = 0;
    s.erase(0, 1);                                   //remove leading
        "'"
    s.pop_back();                                    // remove trailing
        "'"
    for (unsigned short int i = 0; i < s.size(); i++) { //each char in
        string
      result <<= 8;
      result += s[i];
    }
    value = result & 0xffff;
  } else {
    err = INVALID_NUMBER;
  }
  return err;
}

/* check if the string is defined as register in symbol table */
short int is_register(string s) {
  short int flag = INVALID_INDEX;
  s.erase(remove(s.begin(), s.end(), '+'), s.end()); //erase any pre/post Increment (+)
  s.erase(remove(s.begin(), s.end(), '-'), s.end()); //erase any pre/post Decrement (-)
  for (unsigned short int i = 0; i < sym_tab.size(); ++i) {
    if (!sym_tab[i].name.compare(s) && (!sym_tab[i].type.compare("REG"))) { // found a register
      flag = i;
      break;
    }
  }
  return flag;
}

/* remove everything after comment char ';' and split the remaining into different tokens separated by space ' '  */
void get_tokens(string &l, vector<string> &toks) {
  string stripped = l.substr(0, l.find(";")); //strip off comments
  if (stripped.empty()) {
    return;
  } //comment line, nothing to tokenize
  else {
    replace(stripped.begin(), stripped.end(), '\t', ' '); //replace tab witha space, if there is any
    stringstream ss(stripped);
    string token;
    while (getline(ss, token, ' ')) {
      token.erase(remove_if(token.begin(), token.end(), [](char &c) { return isspace<char>(c, locale::classic()); }), token.end()); // remove any unwated space
      if (!token.empty()) {
        toks.push_back(token);
      }
    }
    return;
  }
}

/* returns the index of an instruction, if present in the instruction set, otherwise returns INVALID_INDEX (-1) */
short int check_if_instruction(string s) {
  short int flag = INVALID_INDEX;
  for (unsigned short int i = 0; i < inst_set.size(); ++i) {
    string mnem = inst_set[i].mnemonic;
    bool b = (mnem.size() == s.size()) && (equal(mnem.begin(), mnem.end(),
         s.begin(), [](char &c1, char &c2) { return toupper(c1) == toupper(
        c2); }));
    if (b) {
      flag = i;
      break;
    }
  }

  return flag;
}

/* returns the index of the directive, if present in the directives */
short int check_if_directive(string s) {
  short int flag = INVALID_INDEX;

  for (unsigned short int i = 0; i < directives.size(); ++i) {
    string d = directives[i];
    bool b = (d.size() == s.size()) && (equal(d.begin(), d.end(), s.begin
        (), [](char &c1, char &c2) { return toupper(c1) == toupper(c2); }))
        ;
    if (b) {
      flag = i;
      break;
    }
  }

  return flag;
}

/* impose rules for label name */
bool is_valid_label_name(string lbl) {
  bool flag = true;
  // -- first letter must be alphabetic
  if (!(isalpha(lbl.front()) || lbl.front() == '_') || !(std::find_if(lbl.
      begin(), lbl.end(), [](char c) { return !(std::isalnum(c) || (c == '_
      ')); }) == lbl.end()) || lbl.size() > 32) {
    flag = false;
  }

  return flag;
}

/*return the index of the symbol in sym_tab other than registers - LBL and
     UNK */
short int is_label_in_sym_tab(string lbl) {
  short int flag = INVALID_INDEX;
  for (unsigned short int i = 0; i < sym_tab.size(); ++i) {
    if (!sym_tab[i].name.compare(lbl) && ((!sym_tab[i].type.compare("LBL")
        ) || (!sym_tab[i].type.compare("UNK")))) { // found an entry
      flag = i;
      break;
    }
  }
```

```cpp
  return flag;
}

/* check if the string is CEC */
short int is_cond(string s) {
  short int flag = INVALID_INDEX;
  for (unsigned short int i = 0; i < cecs.size(); ++i) {
    string d = cecs[i];
    bool b = (d.size() == s.size()) && (equal(d.begin(), d.end(), s.begin
        (), [](char &c1, char &c2) { return toupper(c1) == toupper(c2); }))
        ;
    if (b) {
      flag = i;
      break;
    }
  }
  return flag;
}

void init_globals(string src_fname, string inst_fname /* = "instructions.
    txt"*/) {
  /* create operand map to populate expected operands for each
      instructions  */
  map<string, Operand_T> inst_map;      //maps operand string from input
      file to Operand_T
  map<string, Operand_T>::iterator it; //iterator for the map
  inst_map["p"] = IDR;                  //pre/post inc/dec Reg
  inst_map["r"] = R;                    //Reg
  inst_map["o"] = OFFSET;               //offset
  inst_map["a"] = L13;                  //label for 13-bit offset BL
  inst_map["l"] = L10;                  //label for 10-bit offset - other
      branching inst
  inst_map["b"] = BYTE;                 //byte
  inst_map["v"] = CON_R;                //CON or Reg
  inst_map["s"] = SA;                   //SA vector
  inst_map["c"] = COND_CEC;             //val from cec
  inst_map["t"] = TCFC;                 //TC or FC vallue

  inst_set = {};
  ifstream ifs;
  ifs.open(inst_fname);        //open the instruction file
  if (ifs.is_open()) {         //could open the file
    string l;                  //line
    while (getline(ifs, l)) { //read each line in l
      if (l.empty()) {
        continue;
      }
      stringstream ss(l);
      vector<string> tokens;
      string token;
      while (getline(ss, token, ' ')) {
        tokens.push_back(token);
      }
      if (tokens.size() != 3) {
        cout << "Invalid entry in instruction file: " << l << endl;
        break;
      }
      //third token contains operands - decipher it
      auto s = tokens[2];
      vector<Operand_T> v_ops_t;
      if (s.size() < 2) {
        v_ops_t.push_back(inst_map.find(s)->second);
      } else {
        stringstream ss2(s);
        vector<string> v_ops;
        string tok;
        while (getline(ss2, tok, ',')) {
          v_ops.push_back(tok);
        }
        for (auto o : v_ops) {
          v_ops_t.push_back(inst_map.find(o)->second);
        }
      }
      Inst an_inst;
      an_inst.mnemonic = tokens[0];
      an_inst.opcode = tokens[1];
      an_inst.expected_operands = v_ops_t;
      inst_set.push_back(an_inst);
    }

  } else {
    cout << "\nCould not open instruction file " << inst_fname << endl;
  }
  ifs.close(); //end reading instruction file
  /****************** done with instruction set*/
  loc_counter = 0;
  has_error = false;
  //directives = {"ALIGN","BSS","BYTE","END","EQU","ORG","WORD"};
  //cecs = {"EQ","NE","CS","HS","CC","LO","MI","PL","VS","VC","HI","LS","GE","LT","GT","LE","AL"};
  //initialize symbol table and populate with the default registers
  sym_tab = {};
  for (short int i = 0; i < 8; ++i) {
    sym_tab.insert(sym_tab.begin(), Symbol{"R" + to_string(i), "REG", i});
  }

  //create the LIS output file to write the result
  string of_name = src_fname.substr(0, src_fname.find_last_of('.')) + ".lis"; //lis file name
  ofs.open(of_name);
  ofs << " .ASM file: " << src_fname << endl;
  ofs << "\n\n\n"; //put 3 blank lines
}
```

1.5 Pass 1

Header

```
#pragma once
#ifndef _PASS1_H
#define _PASS1_H

#include "Globals.h"
extern short loc_counter;
extern bool has_error;
extern vector<Inst> inst_set;
extern vector<Symbol> sym_tab;

void print_err_to_lis(Error_T e, string s);
void validate_instruction(short int inst_id, vector<string> &toks);
void process_directive(short int d_id, vector<string> &rec, string p_tok =
    "");
void validate_tokens(vector<string> &toks);
bool Pass1(string src_fname);

#endif //_PASS1_H
```

Source

```
#include "Pass1.h"

extern short loc_counter;
extern bool has_error;
extern vector<Inst> inst_set;
extern vector<Symbol> sym_tab;

//prints error message to LIS file
void print_err_to_lis(Error_T e, string s) {

  string msg = "***** ";
  switch (e) {
  case NO_ERR:
    ofs << "\t*****" << s << endl;
    break;
  case MISSING_OPERAND:
    ofs << "\t****** Expected operand: " << s << endl;
    break;
  case ILLEGAL_OPERAND:
    ofs << "\t****** Illegal operand: " << s << endl;
    break;
  case NUMBER_OF_OPERANDS_MISMATCH:
    ofs << "\t ***** Too many/few number of operands: " << s << endl;
    break;
  case INVALID_OPERAND:
    ofs << "\t ***** Invalid operands: " << s << endl;
    break;
  case INVALID_REGISTER:
    ofs << "\t ***** Invalid REG: " << s << endl;
    break;
  case MISSING_INSTRUCTION_DIRECTIVE:
    ofs << "\t ***** Expected INST/DIR: " << s << endl;
    break;
  case INVALID_LABEL_FORMAT:
    ofs << "\t ***** Not valid label: " << s << endl;
    break;
  case UNDEFINED_SYMBOL:
    ofs << "\t ***** Undefined operand(symbol): " << s << endl;
    break;
  case DUPLICATE_LABEL:
    ofs << "\t ***** Duplicate LBL: " << s << endl;
    break;
  case INVALID_NUMBER:
    ofs << "\t ***** Invalid Number: " << s << endl;
    break;
  case INVALID_RECORD:
    ofs << "\t ***** Invalid Record: " << s << endl;
    break;
  default:
    break;
  }
}

/* validates the operands of an instruction - if not, error message is written to the LIS file */
void validate_instruction(short int inst_id, vector<string> &toks) {
  vector<string> operands = {};
  auto ops = toks[1];   //first token is the instruction and 2nd token holds operand(s)
  stringstream ss(ops); //to split into separate operands from the token
  string tok;
  while (getline(ss, tok, ',')) { //operands are separated by comma ','
    operands.push_back(tok);
  }
  auto ex_ops = inst_set[inst_id].expected_operands;
  if (operands.size() != ex_ops.size()) { //number of operands and expected number of operands for this inst is not same
    Error_T e = NUMBER_OF_OPERANDS_MISMATCH;
    string s = "Expected: " + to_string(ex_ops.size()) + " Has: " + to_string(operands.size()) + " operands ";
    print_err_to_lis(e, s);
    has_error = true;
  } else { // validate each operand - either register, or numeric or label type and handle accordingly
    for (unsigned short int i = 0; i < operands.size(); ++i) {
      auto op = operands[i];
      if (is_register(op) != INVALID_INDEX && (ex_ops[i] == IDR || ex_ops[i] == R || ex_ops[i] == CON_R)) { //valid operand - do nothinf
        continue;
      } else if (is_numeric(op)) { //numeric operand
        //obtain value of the operand
        if (ex_ops[i] == L10 || ex_ops[i] == L13) {
          print_err_to_lis(INVALID_OPERAND, "Only labels are permitted for branch targt");
          has_error = true;
        }
        short int r;
        Error_T e = str2int(op, r);
        if (e != NO_ERR) {
          if (r > INT16_MAX || r < INT16_MIN) {
            string s = "Too large or small value..";
            print_err_to_lis(e, s);
            has_error = true;
          } else if ((r > UINT8_MAX || r < BYTE_MIN) && ex_ops[i] == BYTE) {
            string s = "BYTE value should be (0,255)";
            print_err_to_lis(INVALID_OPERAND, s);
            has_error = true;
          } else if ((r != 0 || r != 1 || r != 2 || r != 8 || r != 16 || r != 32 || r != -1) && ex_ops[i] == CON_R) {
            print_err_to_lis(INVALID_NUMBER, "CON value should be [0, 1,2, 8,16, 32 or -1]");
            has_error = true;
          } else if ((r < 0 || r > 15) && ex_ops[i] == SA) {
            print_err_to_lis(INVALID_OPERAND, "SA value should be (0,15)");
            has_error = true;
          } else if ((r < 0 || r > 7) && ex_ops[i] == TCFC) {
            print_err_to_lis(INVALID_OPERAND, "TC/FC value should be (0,7)");
            has_error = true;
          }
        } else {
          if ((ex_ops[i] == IDR || ex_ops[i] == R)) { //REG expected but number
            print_err_to_lis(INVALID_REGISTER, ">" + op + "<");
            has_error = true;
          }
        }

      } else if (is_cond(op) != INVALID_INDEX) { // cond
        if (ex_ops[i] != COND_CEC) {
          print_err_to_lis(INVALID_OPERAND, "COND operand is not valid..");
          has_error = true;
        }
      } else { //label
        if ((ex_ops[i] == IDR || ex_ops[i] == R || ex_ops[i] == CON_R)) { //REG expected but label
          print_err_to_lis(INVALID_REGISTER, ">" + op + "<");
          has_error = true;
        }

        //validate label name and check if the label is in sym_tab else store it
        else if (is_valid_label_name(op)) {

          if (is_label_in_sym_tab(op) == INVALID_INDEX) { //not in sym_tab
```

```
              - store
            sym_tab.insert(sym_tab.begin(), Symbol{op, "UNK", -1});
          } else {
            continue;
          }

        } else {
          print_err_to_lis(INVALID_LABEL_FORMAT, "Invalid Label nmae..");
          has_error = true;
        }
      }
    }
  }
  loc_counter += 2; //each instruction needs 2-bytes
}
/* process ALIGN directive */
void handleDirALIGN(vector<string> &ops) {
  if (ops.size() != 0) { //has operand
    print_err_to_lis(ILLEGAL_OPERAND, "directive ALIGN does not take an operand");
    has_error = true;
  } else {
    if (loc_counter % 2 != 0) {
      loc_counter++;
    } //if odd increment the address
  }
}

/* process BSS directive */
void handleDirBSS(vector<string> &ops) {
  short int r;
  if (ops.size() != 1) { //no operand or more than one operand
    print_err_to_lis(NUMBER_OF_OPERANDS_MISMATCH, "BSS must have one and only one operand");
    has_error = true;
  } else {
    if (is_numeric(ops[0])) {
      Error_T e = str2int(ops[0], r);
      if (e == NO_ERR) {
        loc_counter += r;
      } else {
        print_err_to_lis(e, "BSS operand should be a valid number");
        has_error = true;
      }

    } else {
      auto r2 = is_label_in_sym_tab(ops[0]);
      if (r2 == INVALID_INDEX) { // not in symbol table - check name, store label in sym_tab and emit error
        if (is_valid_label_name(ops[0])) {
          sym_tab.insert(sym_tab.begin(), Symbol{ops[0], "UNK", -1});
        } else {
          print_err_to_lis(INVALID_LABEL_FORMAT, "");
          has_error = true;
        }

      } else {
        loc_counter += sym_tab[r2].value;
      }
    }
  }
}

/* process BSS directive */
void handleDirBYTE(vector<string> &ops) {
  short int r;
  if (ops.size() != 1) { //no operand or more than one operand
    print_err_to_lis(NUMBER_OF_OPERANDS_MISMATCH, "BYTE must have one and
        only one operand");
    has_error = true;
  } else {
    if (is_numeric(ops[0])) {
      Error_T e = str2int(ops[0], r);
      if (e == NO_ERR && (r < BYTE_MIN || r > INT8_MAX)) {
        print_err_to_lis(INVALID_OPERAND, "BYTE must be 8-bit size (0,255)
            ");
        has_error = true;
      } else if (e != NO_ERR) { //str2err could not convert
        print_err_to_lis(e, " Invalid operand for BYTE directive ");
        has_error = true;
      } else { //no error and valid BYTE size
      }
    } else { //operand is not a number - consider a label
      auto r2 = is_label_in_sym_tab(ops[0]);
      if (r2 == INVALID_INDEX) { // not in symbol table - check name,
          store label in sym_tab and emit error
        if (is_valid_label_name(ops[0])) {
          sym_tab.insert(sym_tab.begin(), Symbol{ops[0], "UNK", -1});
        } else {
          print_err_to_lis(INVALID_LABEL_FORMAT, "");
          has_error = true;
        }
      } else if ((sym_tab[r2].value < BYTE_MIN || sym_tab[r2].value >
          INT8_MAX)) { //label in sym_tab
        print_err_to_lis(INVALID_OPERAND, "BYTE must be 8-bit size (0,255)
            ");
        has_error = true;
      }
    }
  }
}

/* process END directive */
void handleDirEND(vector<string> &ops) {
  short int r;
  if (ops.size() == 1) {
    Error_T e = str2int(ops[0], r);
    if (e != NO_ERR) { //not a valid number - may be label name
```

```
      auto r1 = is_label_in_sym_tab(ops[0]);
      if (r1 == INVALID_INDEX) { // not in symbol table - check name, store label in sym_tab and emit error
        if (is_valid_label_name(ops[0])) {
          sym_tab.insert(sym_tab.begin(), Symbol{ops[0], "UNK", -1});
        } else {
          print_err_to_lis(INVALID_LABEL_FORMAT, "");
          has_error = true;
        }
      }
    }
  }
}

/* process EQU directive */
void handleDirEQU(vector<string> &ops, string &p_tok) {
  short int r;
  if (p_tok.empty()) { //preceding token must be label but not present
    print_err_to_lis(UNDEFINED_SYMBOL, "EQU directive must be preceded by a LBL");
    has_error = true;

  }

  else if (ops.size() != 1) {
    print_err_to_lis(MISSING_OPERAND, "EQU must have an operand");
    has_error = true;
  } else if (ops.size() == 1) { // looks fine - operand could be a value or a register
    auto r1 = is_label_in_sym_tab(p_tok);

    auto rr = is_register(ops[0]);
    if (rr == INVALID_INDEX && !is_numeric(ops[0])) { //operand is neither a register nor a value - error
      print_err_to_lis(INVALID_OPERAND, "Operand of EQU must be a value or a REG");
      has_error = true;
    } else if (rr != INVALID_INDEX && !sym_tab[r1].type.compare("UNK")) { //label is a REG with the corresponding REG value
      cout << sym_tab[r1].type << "\tvalue " << sym_tab[r1].value << endl;
      sym_tab[r1].type = "REG";
      sym_tab[r1].value = sym_tab[rr].value;
    } else { //operand is numeric

      Error_T e = str2int(ops[0], r);
      if (e != NO_ERR) { //has error in the value
        print_err_to_lis(e, "Operand of EQU is not valid");
        has_error = true;
      } else {

        if (!sym_tab[r1].type.compare("UNK")) {
          sym_tab[r1].type = "LBL";
          sym_tab[r1].value = r;
        }
      }
    }
  }
}

/* process ORG directive */
void handleDirORG(vector<string> &ops) {
  short int r;
  if (ops.size() != 1) {
    print_err_to_lis(INVALID_OPERAND, "ORG should have an operand");
    has_error = true;
  } else {
    if (is_numeric(ops[0])) {
      Error_T e = str2int(ops[0], r);
      if (e == NO_ERR) {
        loc_counter = r;
      } else {
        print_err_to_lis(INVALID_NUMBER, "ORG operand should be a valid number");
        has_error = true;
      }
    } else {
      print_err_to_lis(INVALID_OPERAND, "ORG operand should be a valid number");
      has_error = true;
    }
  }
}

/* process WORD directive */
void handleDirWORD(vector<string> &ops) {
  short int r;
  if (ops.size() != 1) { //no operand or more than one operand
    print_err_to_lis(NUMBER_OF_OPERANDS_MISMATCH, "WORD must have an operand");
    has_error = true;
  } else {

    if (is_numeric(ops[0])) {
      Error_T e = str2int(ops[0], r);
      if (e == NO_ERR && (r < BYTE_MIN || r > UINT16_MAX)) {
        print_err_to_lis(INVALID_OPERAND, "WORD must be 16-bit size (0,65535)");
        has_error = true;
      } else if (e != NO_ERR) { //str2err could not convert
        print_err_to_lis(e, " Invalid operand for WORD directive ");
        has_error = true;
      } else { //no error and valid BYTE size
      }
    } else { //operand is not a number - consider a label
      auto r2 = is_label_in_sym_tab(ops[0]);
      if (r2 == INVALID_INDEX) { // not in symbol table - check name, store label in sym_tab and emit error
        if (is_valid_label_name(ops[0])) {
          sym_tab.insert(sym_tab.begin(), Symbol{ops[0], "UNK", -1});
        } else {
          print_err_to_lis(INVALID_LABEL_FORMAT, "");
          has_error = true;
        }
      } else if ((sym_tab[r2].value < BYTE_MIN || sym_tab[r2].value >
           UINT16_MAX)) { //label in sym_tab
        print_err_to_lis(INVALID_OPERAND, "WORD must be 8-bit size (0,255)
            ");
        has_error = true;
      }
    }
  }
}

/* process the directive based on the index in directives */
void process_directive(short int d_id, vector<string> &rec, string p_tok)
    {
  /************ */
  vector<string> operands = {};
  if (rec.size() == 2) { //has operand
    auto ops = rec[1];
    /* split different operands - separated by comma */
    stringstream ss(ops); //
    string tok;
    while (getline(ss, tok, ',')) {
      operands.push_back(tok);
    }
  }
  directiveIndexes di = static_cast<directiveIndexes>(d_id);

  switch (di) {
  case dirALIGN:
    handleDirALIGN(operands);
    break;
  case dirBSS:
    handleDirBSS(operands);
    break;
  case dirBYTE:
    handleDirBYTE(operands);
    loc_counter += BYTE_INCREASE;
    break;
  case dirEND:
    handleDirEND(operands);
    break;
  case dirEQU:
    handleDirEQU(operands, p_tok);
    break;
  case dirORG:
    handleDirORG(operands);
    break;
  case dirWORD: //6: //WORD
    handleDirWORD(operands);
    loc_counter += WORD_INCREASE;
    break;
  default:
    break;
  }
  return;
}

/*function to validate tokens in a record for pass 1*/
void validate_tokens(vector<string> &toks) {

  if (toks.size() > 3) { //too many tokens in a record
    print_err_to_lis(INVALID_RECORD, "too many tokens ");
    has_error = true;
    return;
  }

  //consider 1st token as instruction
  short int id = check_if_instruction(toks[0]);

  if (id != INVALID_INDEX) { //an instruction - next token must present and operand(s)

    validate_instruction(id, toks);
  } else { //either directive or label
    id = check_if_directive(toks[0]);

    if (id != INVALID_INDEX) { // a directive found - there may or may not have operand(s)
      if (id == 4) {            //EQU should not be here
        print_err_to_lis(UNDEFINED_SYMBOL, "EQU should be preceded by LBL");
        has_error = true;
      } else {
        process_directive(id, toks);
      }

    } else { // this token is a label - following token must be INST/DIR, if any

      if (is_valid_label_name(toks[0])) { //valid name - go ahead
        id = is_label_in_sym_tab(toks[0]);
        if (id == INVALID_INDEX) { //no label in sym_tab with this name - so insert
          sym_tab.insert(sym_tab.begin(), Symbol{toks[0], "LBL", loc_counter});
        } else if (!sym_tab[id].type.compare("UNK")) { //UNK label found - change type and value
          sym_tab[id].type = "LBL";
          sym_tab[id].value = loc_counter;
        } else { //duplicate label
          print_err_to_lis(DUPLICATE_LABEL, "");
          has_error = true;
        }
      } else { //invalid label name
```

```
        print_err_to_lis(INVALID_LABEL_FORMAT, "Invalid Label nmae..");
        has_error = true;
      }
      if (toks.size() > 1) { //more tokens are there and has to be INST/DIR
        string prev_tok = toks[0];
        toks.erase(toks.begin()); //erase first token
        id = check_if_instruction(toks[0]);
        if (id != INVALID_INDEX) { //an instruction - next token must present and operand(s)
          validate_instruction(id, toks);
        } else { //either directive or label
          id = check_if_directive(toks[0]);
          if (id != INVALID_INDEX) { // a directive found - there may or may not have operand(s)
            process_directive(id, toks, prev_tok);
          } else { // this tok is a label - again! -Error
            print_err_to_lis(MISSING_INSTRUCTION_DIRECTIVE, "Expected INST/DIR after a LBL");
            has_error = true;
          }
        }

        /****** */
      }
    }
  }
}

/* conducts pass1 */
bool Pass1(string src_fname) {
  // read the src file line by line and process it
  ifstream ifs;
  ifs.open(src_fname);
  string line;           //to hold the content of a line
  short int n_line = 0; //corresponding line number in the src file
  if (ifs.is_open()) {
    //bool no_err = true;
    //read each line of src file and process it
    while (getline(ifs, line)) {
      n_line++;
      ofs << "\t" << n_line << "\t" << line << endl;
      if (line.empty()) {
        continue;
      } else {
        //get tokens from the line
        vector<string> tokens = {};
        get_tokens(line, tokens);
        if (tokens.size() > 0) {
          validate_tokens(tokens);
        } else {
          continue;
        }
      }
    }

    ifs.close();

  } else {
    cout << "Could not open source file: " << src_fname << endl;
    return false;
  }

  for (unsigned short int i = 0; i < sym_tab.size(); ++i) {
    if (!sym_tab[i].type.compare("UNK")) { // found an 'UNK' entry
      has_error = true;
      break;
    }
  }

  if (has_error) {
    //print sym_tab
    cout << "Has Error...printing to LIS" << endl;
    ofs << "First pass error....assembly terminated...." << endl;

    ofs << "\n ****  Symbol Table ***" << std::endl;
    ofs << "Name\t\tType\tValue\tDecimal" << std::endl;

    for (auto s : sym_tab) {
      stringstream ss;
      ss << std::uppercase << std::setfill('0') << std::setw(4) << std::hex << s.value;
      string hex_v;
      if (s.value == -1) {
        hex_v = "FFFF";
      } else {
        string t = ss.str();
        hex_v = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
      }
      ofs << s.name << "\t\t" << s.type << "\t" << hex_v << "\t" << s.value << endl;
    }
    ofs.close();
    return false;
  } else {
    cout << "No Error in Pass1...starting Pass 2" << endl;
    // ofs<<"First pass error....assembly terminated...."<<endl;

    ofs << "\n ****  Symbol Table ***" << std::endl;
    ofs << "Name\t\tType\tValue\tDecimal" << std::endl;

    for (auto s : sym_tab) {
      stringstream ss;
      ss << std::uppercase << std::setfill('0') << std::setw(4) << std::hex << s.value;
      string t = ss.str();
      string hex_v = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;

      ofs << s.name << "\t\t" << s.type << "\t" << hex_v << "\t" << s.value << endl;
    }
    ofs.close();
    return true;
  }
}
```

2 Testing Files

2.1 Input ASM

a1.asm

```

;
; Sample XM2 Assembler program to increment a number by 1
; A. N. O'Nymous
; 7 May 2019
;
; Data space
org $80
Number word #0
;
; Code
org $1000
Start
;
; R0 = Address of Number
;
movlz Number,R0
;
; R1 = Value stored in Number [R0]
;
ld R0,R1
;
; Increment R1
;
add #1,R1
;
; Number [R0] = R1
;
st R1,R0
;
; End of program
; Specifying first executable location (Start)
;
end Start
```

a2.asm

```
org $80
Label01 word #0
org $FF00
Label02 word #1
org $100
Label03 MOVLZ Label01,R0
MOVLS Label02,R1
LD R0,R0
LD R1,R2
Label04 ADD R0,R2
CMP #16,R2
BNE Label04
ST R2,R1
```

```
Done BRA Done
END Label03
```

example1.asm

```
SIZE equ $26
CAP_A equ 'A'

org #80
BASE bss SIZE
org $1000

Start movlz CAP_A,R0
```

test1.asm

```

; Operands in wrong order:
  ld  R0,#1
  ldr R5,#-60,r0
```

FirstPassErrorsExample.asm

```
;
; Examples of some first pass errors
; - Look at console output and .LIS file
;
; L. Hughes
; 29 May 2019 - Revised for XM2
; 11 May 2018 - original

; Missing operand:
START equ

; Invalid operand
  byte  '123'

; Unexpected operand
  align R1

; Missing '#' or '$':
  org 100

; Symbols not in symbol table:
  ld  r0,r1

; Unknown instruction:
Loop  lda r3

; >"< not supported, should be >'<:
Data  byte  "x"

; Operands in wrong order:
  add R0,#1

; Unknown instruction (beqx) treated as a label.  "Loop" then examined
; and treated as an instruction:
  beqx  loop

; Unknown register
  swap  R1,R8

; Bad BSS value - unknown value
  BSS ArraySize

; Duplicate label
START equ #12

; Start not defined (put in symbol table as UNK - unknown symbol):
  end Start
```

2.2 Output LIS

a1.asm

```
 .ASM file: a1.asm



  1
  2 ;
  3 ; Sample XM2 Assembler program to increment a number by 1
  4 ; A. N. O'Nymous
  5 ; 7 May 2019
  6 ;
  7 ; Data space
  8 org $80
  9 Number word #0
  10  ;
  11  ; Code
  12  org $1000
  13  Start
  14  ;
  15  ; R0 = Address of Number
  16  ;
  17  movlz Number,R0
  18  ;
  19  ; R1 = Value stored in Number [R0]
  20  ;
  21  ld R0,R1
  22  ;
  23  ; Increment R1
  24  ;
  25  add #1,R1
  26  ;
  27  ; Number [R0] = R1
  28  ;
  29  st R1,R0
  30  ;
  31  ; End of program
  32  ; Specifying first executable location (Start)
  33  ;
  34  end Start

 ****  Symbol Table ***
Name     Type  Value Decimal
Start   LBL 1000  4096
Number    LBL 0080  128
R7    REG 0007  7
R6    REG 0006  6
R5    REG 0005  5
R4    REG 0004  4
R3    REG 0003  3
R2    REG 0002  2
R1    REG 0001  1
R0    REG 0000  0
```

a2.asm

```
 .ASM file: a2.asm



  1 org $80
  2 Label01 word #0
  3 org $FF00
  4 Label02 word #1
  5 org $100
  6 Label03 MOVLZ Label01,R0
  7 MOVLS Label02,R1
  8 LD R0,R0
  9 LD R1,R2
  10  Label04 ADD R0,R2
  11  CMP #16,R2
  12  BNE Label04
  13  ST R2,R1
  14  Done BRA Done
  15  END Label03

 ****  Symbol Table ***
Name     Type  Value Decimal
Done     LBL 0110  272
Label04   LBL 0108  264
Label03   LBL 0100  256
Label02   LBL FF00  -256
Label01   LBL 0080  128
R7    REG 0007  7
R6    REG 0006  6
R5    REG 0005  5
R4    REG 0004  4
R3    REG 0003  3
R2    REG 0002  2
R1    REG 0001  1
R0    REG 0000  0
```

example1.asm

```
 .ASM file: example1.asm



  1 SIZE equ $26
  2 CAP_A equ 'A'
  3
  4 org #80
  5 BASE bss SIZE
  6 org $1000
  7
  8 Start movlz CAP_A,R0
  9

 ****  Symbol Table ***
Name     Type  Value Decimal
Start    LBL 1000  4096
BASE     LBL 0050  80
CAP_A    LBL 0000  0
SIZE     LBL 0000  0
R7    REG 0007  7
R6    REG 0006  6
R5    REG 0005  5
R4    REG 0004  4
R3    REG 0003  3
R2    REG 0002  2
R1    REG 0001  1
R0    REG 0000  0
```

test1.asm

```
 .ASM file: test1.asm



  1
  2 ; Operands in wrong order:
  3   ld  R0,#1
   ***** Invalid REG: >#1<
  4   ldr R5,#-60,r0
   ***** Invalid REG: >r0<
First pass error....assembly terminated....

 ****  Symbol Table ***
Name     Type  Value Decimal
R7    REG 0007  7
R6    REG 0006  6
R5    REG 0005  5
R4    REG 0004  4
R3    REG 0003  3
R2    REG 0002  2
R1    REG 0001  1
R0    REG 0000  0
```

FirstPassErrorsExample.asm

```
.ASM file: FirstPassErrorsExample.asm



 1 ;
 2 ; Examples of some first pass errors
 3 ; - Look at console output and .LIS file
 4 ;
 5 ; L. Hughes
 6 ; 29 May 2019 - Revised for XM2
 7 ; 11 May 2018 - original
 8
 9 ; Missing operand:
 10  START equ
 ****** Expected operand: EQU must have an operand
 11
 12  ; Invalid operand
 13    byte  '123'
  ***** Invalid operands: BYTE must be 8-bit size (0,255)
 14
 15  ; Unexpected operand
 16    align R1
 ****** Illegal operand: directive ALIGN does not take an operand
 17
 18  ; Missing '#' or '$':
 19    org 100
  ***** Invalid operands: ORG operand should be a valid number
 20
 21  ; Symbols not in symbol table:
 22    ld  r0,r1
  ***** Invalid REG: >r0<
  ***** Invalid REG: >r1<
 23
 24  ; Unknown instruction:
 25  Loop  lda r3
  ***** Expected INST/DIR: Expected INST/DIR after a LBL
 26
 27  ; >"< not supported, should be >'<:
 28  Data  byte  "x"
  ***** Not valid label:
 29
 30  ; Operands in wrong order:
 31    add R0,#1
  ***** Invalid REG: >#1<
 32
 33  ; Unknown instruction (beqx) treated as a label.  "Loop" then examined
 34  ; and treated as an instruction:
 35    beqx  loop
  ***** Expected INST/DIR: Expected INST/DIR after a LBL
 36
 37  ; Unknown register
 38    swap  R1,R8
  ***** Invalid REG: >R8<
  39
  40  ; Bad BSS value - unknown value
  41    BSS ArraySize
  42
  43  ; Duplicate label
  44  START equ #12
   ***** Duplicate LBL:
  45
  46  ; Start not defined (put in symbol table as UNK - unknown symbol):
  47    end Start
First pass error....assembly terminated....

 ****  Symbol Table ***
Name    Type  Value Decimal
Start   UNK FFFF  -1
ArraySize   UNK FFFF  -1
beqx    LBL 0006  6
Data    LBL 0003  3
Loop    LBL 0003  3
START   LBL 0000  0
R7    REG 0007  7
R6    REG 0006  6
R5    REG 0005  5
R4    REG 0004  4
R3    REG 0003  3
R2    REG 0002  2
R1    REG 0001  1
R0    REG 0000  0
```

Thank you for being so patient with me.