

# Assignment 1

Sakif Fahmid Zaman  
B00756635  
Dalhousie University  
ECED 3403

July 23, 2019

# Contents

<b>1</b>	<b>Source Files</b>	<b>2</b>
1.1	cmake . . . . .	2
1.2	Main . . . . .	2
1.3	Instructions . . . . .	2
1.4	Globals . . . . .	4
1.5	Pass 1 . . . . .	11
<b>2</b>	<b>Testing Files</b>	<b>23</b>
2.1	Input ASM . . . . .	23
2.2	Output LIS . . . . .	25

# 1 Source Files

## 1.1 cmake

```
1 cmake_minimum_required(VERSION 3.13)
2
3 set(CMAKE_CXX_STANDARD 17)
4 set(C_STANDARD 18)
5 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
6 set(CMAKE_CXX_FLAGS "-O3 -Wall")
7
8 project(assembly)
9
10 add_executable(assembly Globals.cpp Pass1.cpp main.cpp)
```

## 1.2 Main

```
1 #include "Globals.h"
2 #include "Pass1.h"
3
4 int main(int argc, char *argv[]) {
5
6     init_globals(argv[1]);
7     if (Pass1(argv[1])) {
8         cout << "Pass1 done" << endl;
9     }
10 }
```

## 1.3 Instructions

```
1 BL 000 a
2 BEQ 001000 1
3 BZ 001000 1
4 BNE 001001 1
5 BNZ 001001 1
6 BC 001010 1
7 BHS 001010 1
8 BNC 001011 1
9 BLO 001011 1
10 BN 001100 1
11 BGE 001101 1
12 BLT 001110 1
13 BRA 001111 1
14 ADD 01000000 v,r
15 ADD.B 01000000 v,r
16 ADD.W 01000000 v,r
17 ADDC 01000001 v,r
18 ADDC.B 01000001 v,r
19 ADDC.W 01000001 v,r
20 SUB 01000010 v,r
21 SUB.B 01000010 v,r
22 SUB.W 01000010 v,r
23 SUBC 01000011 v,r
24 SUBC.B 01000011 v,r
25 SUBC.W 01000011 v,r
```

```
26 DADD 01000100 v,r
27 DADD.B 01000100 v,r
28 DADD.W 01000100 v,r
29 CMP 01000101 v,r
30 CMP.B 01000101 v,r
31 CMP.W 01000101 v,r
32 XOR 01000110 v,r
33 XOR.B 01000110 v,r
34 XOR.W 01000110 v,r
35 AND 01000111 v,r
36 AND.B 01000111 v,r
37 AND.W 01000111 v,r
38 BIT 01001000 v,r
39 BIT.B 01001000 v,r
40 BIT.W 01001000 v,r
41 BIC 01001001 v,r
42 BIC.B 01001001 v,r
43 BIC.W 01001001 v,r
44 BIS 01001010 v,r
45 BIS.B 01001010 v,r
46 BIS.W 01001010 v,r
47 MOV 01001011 v,r
48 MOV.B 01001011 v,r
49 MOV.W 01001011 v,r
50 SWAP 01001100 r,r
51 SRA 010011010 r
52 SRA.B 010011010 r
53 SRA.W 010011010 r
54 RRC 010011100 r
55 RRC.B 010011010 r
56 RRC.W 010011010 r
57 SWPB 0100111100000 r
58 SWPB.W 0100111100000 r
59 SXT 0100111110000 r
60 SXT.W 0100111110000 r
61 SVC 010110000000 s
62 LD 010100 p,r
63 LD.B 010100 p,r
64 LD.W 010100 p,r
65 ST 010101 r,p
66 ST.B 010100 r,p
67 ST.W 010100 r,p
68 CEX 010111 c,t,t
69 MOVL 01100 b,r
70 MOVLZ 01101 b,r
71 MOVLS 01110 b,r
72 MOVH 01110 b,r
73 LDR 10 r,o,r
74 LDR.B 10 r,o,r
75 LDR.W 10 r,o,r
76 STR 11 r,r,o
77 STR.B 11 r,r,o
78 STR.W 11 r,r,o
```

## 1.4 Globals

### Header

```
1 #ifndef _GLOBALS_H
2 #define _GLOBALS_H
3
4 #include "stdio.h"
5 #include <algorithm>
6 #include <cctype>
7 #include <fstream>
8 #include <iomanip>
9 #include <iostream>
10 #include <map>
11 #include <sstream>
12 #include <stdlib.h>
13 #include <string>
14 #include <vector>
15
16 #define INVALID_INDEX -1
17 #define BYTE_MIN 0
18 #define BYTE_INCREASE 1
19 #define WORD_INCREASE 2
20
21 using namespace std;
22
23 enum Operand_T { //operands tyoe
24     IDR, //pre/post increment decrement register - denoted as 'p'
25         in instruction file
26     R, //register (no, pre/post +/-) - 'r'
27     OFFSET, //offset value - signed number with range [-64,63] - 'o'
28     L13, //label with 13-bit offset - 'a'
29     L10, //label with 10-bit offset - 'l'
30     BYTE, //byte value - 'b'
31     CON_R, //register or fixed/constant value [0,1,2,8,16,32,-1] - 'v'
32     SA, //vector value [0-15], 's'
33     COND_CEC, //cec value - 'c'
34     TCFC //TC/FC type [0-7] - 't'
35 };
36
37 enum Error_T {
38     NO_ERR = 0, //no error :-)
39     MISSING_OPERAND, //missing operand, eg. instruction/
40         directive without required operand
41     ILLEGAL_OPERAND, //unwanted operand, eg, operand after
42         ALIGN
43     NUMBER_OF_OPERANDS_MISMATCH, //too many ortoo few operands compared to
44         expected
45     INVALID_OPERAND, //operand present but invalid. eg, byte
46         value is not in byte range
47     INVALID_REGISTER, //expected register operand but got
48         something else, possibly undefined
49     MISSING_INSTRUCTION_DIRECTIVE, //token following a LBL must be either an
50         INST or a DIR
```

```

44     INVALID_LABEL_FORMAT,           //label format is not valid
45     UNDEFINED_SYMBOL,               //Symbol(label or register) is not
        defined - might be capyured at the end of first pass/ in 2nd pass??
46     DUPLICATE_LABEL,               //duplicate label in symbol table
47     INVALID_NUMBER,                //the string is not a valid number -
        compare with INVALID_OPERAND???
48     INVALID_RECORD                  //too many tokens
49 };
50
51 struct Symbol {
52     string name; //name of the symbol
53     string type; //SymbolType type; //label (LBL), register (REG) or unknown
        (UNK)
54     int value; //value - decimal
55 };
56
57 struct Inst {                       //instruction
58     string mnemonic;                //mnemonic of the instruction
59     string opcode;                  //most significant binary bits of
        this mnemonic -stored as string to join rest of the bits to create
        full opcode
60     vector<Operand_T> expected_operands; //rule for the expected operands
        for this instruction
61
62     //Inst(string m, string o, vector<Operand_T> e):mnemonic(m), opcode(o),
        expected_operands(e){}
63 };
64
65 //global vars
66 extern short int loc_counter;
67 extern bool has_error;
68 extern vector<Inst> inst_set; //instruction set - global scope
69 extern ofstream ofs;
70 const vector<string> directives = {"ALIGN", "BSS", "BYTE", "END", "EQU", "
    ORG", "WORD"};
71 enum directiveIndexes {
72     dirALIGN,
73     dirBSS,
74     dirBYTE,
75     dirEND,
76     dirEQU,
77     dirORG,
78     dirWORD
79 };
80
81 const vector<string> cecs = {"EQ", "NE", "CS", "HS", "CC", "LO", "MI", "PL
    ", "VS", "VC", "HI", "LS", "GE", "LT", "GT", "LE", "AL"};
82 const vector<unsigned short int> cec_values = {0, 1, 2, 2, 3, 3, 4, 5, 6,
    7, 8, 9, 10, 11, 12, 13, 14};
83 extern vector<Symbol> sym_tab;
84
85 //functions
86 bool is_numeric(string s);
87 Error_T str2int(string s, short int &value);

```

```

88 short int is_register(string s);
89 void get_tokens(string &l, vector<string> &toks);
90 short int check_if_instruction(string s);
91 short int check_if_directive(string s);
92 bool is_valid_label_name(string lbl);
93 short int is_label_in_sym_tab(string lbl);
94 short int is_cond(string s);
95
96 void init_globals(string src_fname, string inst_fname = "instructions.txt"
    );
97
98 #endif //_GLOBALS_H

```

### Source

```

1 #include "Globals.h"
2
3 short loc_counter;
4 bool has_error;
5 vector<Inst> inst_set;
6 vector<Symbol> sym_tab;
7 ofstream ofs;
8
9 /* check if the string has format of Numeric type operand */
10 bool is_numeric(string s) {
11     if (((s.front() == '#') || (s.front() == '$')) || ((s.front() == '\\')
        && (s.back() == '\\'))) {
12         return true;
13     } else {
14         return false;
15     }
16 }
17
18 /* converts numeric type strings to corresponding integer value, error is
    captured and returns false then */
19 Error_T str2int(string s, short int &value) {
20     Error_T err = NO_ERR;
21     if (s.front() == '#') { //integer value
22         s.erase(0, 1);          //remove leading '#'
23         value = stoi(s);
24     } else if (s.front() == '$') { //hex value
25         s.erase(0, 1);          //remove leading '$'
26         value = stoi(s, nullptr, 16);
27     } else if (s.front() == '\\' && s.back() == '\\') { //char - could be
        escaped or alpha-numeric
28         unsigned int result = 0;
29         s.erase(0, 1);          //remove leading
            " , "
30         s.pop_back();          // remove trailing
            " , "
31         for (unsigned short int i = 0; i < s.size(); i++) { //each char in
            string
32             result <= 8;
33             result += s[i];
34         }

```

```

35     value = result & 0xffff;
36 } else {
37     err = INVALID_NUMBER;
38 }
39 return err;
40 }
41
42 /* check if the string is defined as register in symbol table */
43 short int is_register(string s) {
44     short int flag = INVALID_INDEX;
45     s.erase(remove(s.begin(), s.end(), '+'), s.end()); //erase any pre/post
         Increment (+)
46     s.erase(remove(s.begin(), s.end(), '-'), s.end()); //erase any pre/post
         Decrement (-)
47     for (unsigned short int i = 0; i < sym_tab.size(); ++i) {
48         if (!sym_tab[i].name.compare(s) && (!sym_tab[i].type.compare("REG")))
49             { // found a register
50                 flag = i;
51                 break;
52             }
53     return flag;
54 }
55
56 /* remove everything after comment char ';' and split the remaining into
different tokens separated by space ' ' */
57 void get_tokens(string &l, vector<string> &toks) {
58     string stripped = l.substr(0, l.find(";")); //strip off comments
59     if (stripped.empty()) {
60         return;
61     } //comment line, nothing to tokenize
62     else {
63         replace(stripped.begin(), stripped.end(), '\\t', ' '); //replace tab
            with a space, if there is any
64         stringstream ss(stripped);
65         string token;
66         while (getline(ss, token, ' ')) {
67             token.erase(remove_if(token.begin(), token.end(), [](char &c) {
68                 return isspace<char>(c, locale::classic()); }), token.end()); //
                remove any unwanted space
69             if (!token.empty()) {
70                 toks.push_back(token);
71             }
72         }
73     return;
74 }
75
76 /* returns the index of an instruction, if present in the instruction set,
otherwise returns INVALID_INDEX (-1) */
77 short int check_if_instruction(string s) {
78     short int flag = INVALID_INDEX;
79     for (unsigned short int i = 0; i < inst_set.size(); ++i) {
80         string mnem = inst_set[i].mnemonic;

```



```

81     bool b = (mnem.size() == s.size()) && (equal(mnem.begin(), mnem.end(),
            s.begin(), [](char &c1, char &c2) { return toupper(c1) == toupper(
                c2); }));
82     if (b) {
83         flag = i;
84         break;
85     }
86 }
87
88 return flag;
89 }
90
91 /* returns the index of the directive, if present in the directives */
92 short int check_if_directive(string s) {
93     short int flag = INVALID_INDEX;
94
95     for (unsigned short int i = 0; i < directives.size(); ++i) {
96         string d = directives[i];
97         bool b = (d.size() == s.size()) && (equal(d.begin(), d.end(), s.begin
            (), [](char &c1, char &c2) { return toupper(c1) == toupper(c2); }));
98         if (b) {
99             flag = i;
100             break;
101         }
102     }
103
104     return flag;
105 }
106
107 /* impose rules for label name */
108 bool is_valid_label_name(string lbl) {
109     bool flag = true;
110     // -- first letter must be alphabetic
111     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) {
112         flag = false;
113     }
114
115     return flag;
116 }
117
118 /*return the index of the symbol in sym_tab other than registers - LBL and
    UNK */
119 short int is_label_in_sym_tab(string lbl) {
120     short int flag = INVALID_INDEX;
121     for (unsigned short int i = 0; i < sym_tab.size(); ++i) {
122         if (!sym_tab[i].name.compare(lbl) && ((!sym_tab[i].type.compare("LBL")
            ) || (!sym_tab[i].type.compare("UNK")))) { // found an entry
123             flag = i;
124             break;
125         }
126     }

```

```

127     return flag;
128 }
129
130 /* check if the string is CEC */
131 short int is_cond(string s) {
132     short int flag = INVALID_INDEX;
133     for (unsigned short int i = 0; i < cecs.size(); ++i) {
134         string d = cecs[i];
135         bool b = (d.size() == s.size()) && (equal(d.begin(), d.end(), s.begin
            (), [](char &c1, char &c2) { return toupper(c1) == toupper(c2); }));
136         if (b) {
137             flag = i;
138             break;
139         }
140     }
141     return flag;
142 }
143
144 void init_globals(string src_fname, string inst_fname /* = "instructions.
    txt"*/) {
145     /* create operand map to populate expected operands for each
        instructions */
146     map<string, Operand_T> inst_map;           //maps operand string from input
        file to Operand_T
147     map<string, Operand_T>::iterator it; //iterator for the map
148     inst_map["p"] = IDR;                     //pre/post inc/dec Reg
149     inst_map["r"] = R;                       //Reg
150     inst_map["o"] = OFFSET;                  //offset
151     inst_map["a"] = L13;                     //label for 13-bit offset BL
152     inst_map["l"] = L10;                     //label for 10-bit offset - other
        branching inst
153     inst_map["b"] = BYTE;                    //byte
154     inst_map["v"] = CON_R;                   //CON or Reg
155     inst_map["s"] = SA;                      //SA vector
156     inst_map["c"] = COND_CEC;                //val from cec
157     inst_map["t"] = TCFC;                    //TC or FC vallue
158
159     inst_set = {};
160     ifstream ifs;
161     ifs.open(inst_fname);                    //open the instruction file
162     if (ifs.is_open()) {                     //could open the file
163         string l;                             //line
164         while (getline(ifs, l)) { //read each line in l
165             if (l.empty()) {
166                 continue;
167             }
168             stringstream ss(l);
169             vector<string> tokens;
170             string token;
171             while (getline(ss, token, ' ')) {
172                 tokens.push_back(token);
173             }
174             if (tokens.size() != 3) {

```

```

175         cout << "Invalid entry in instruction file: " << l << endl;
176         break;
177     }
178     //third token contains operands - decipher it
179     auto s = tokens[2];
180     vector<Operand_T> v_ops_t;
181     if (s.size() < 2) {
182         v_ops_t.push_back(inst_map.find(s)->second);
183     } else {
184         stringstream ss2(s);
185         vector<string> v_ops;
186         string tok;
187         while (getline(ss2, tok, ',')) {
188             v_ops.push_back(tok);
189         }
190         for (auto o : v_ops) {
191             v_ops_t.push_back(inst_map.find(o)->second);
192         }
193     }
194     Inst an_inst;
195     an_inst.mnemonic = tokens[0];
196     an_inst.opcode = tokens[1];
197     an_inst.expected_operands = v_ops_t;
198     inst_set.push_back(an_inst);
199 }
200
201 } else {
202     cout << "\nCould not open instruction file " << inst_fname << endl;
203 }
204 ifs.close(); //end reading instruction file
205 ***** done with instruction set*/
206 loc_counter = 0;
207 has_error = false;
208 //directives = {"ALIGN", "BSS", "BYTE", "END", "EQU", "ORG", "WORD"};
209 //cecs = {"EQ", "NE", "CS", "HS", "CC", "LO", "MI", "PL", "VS", "VC", "HI", "LS", "
    GE", "LT", "GT", "LE", "AL"};
210 //initialize symbol table and populate with the default registers
211 sym_tab = {};
212 for (short int i = 0; i < 8; ++i) {
213     sym_tab.insert(sym_tab.begin(), Symbol{"R" + to_string(i), "REG", i});
214 }
215
216 //create the LIS output file to write the result
217 string of_name = src_fname.substr(0, src_fname.find_last_of('.') + 1) + ".
    lis"; //lis file name
218 ofs.open(of_name);
219 ofs << " .ASM file: " << src_fname << endl;
220 ofs << "\n\n\n"; //put 3 blank lines
221 }

```

## 1.5 Pass 1

### Header

```
1 #pragma once
2 #ifndef _PASS1_H
3 #define _PASS1_H
4
5 #include "Globals.h"
6 extern short loc_counter;
7 extern bool has_error;
8 extern vector<Inst> inst_set;
9 extern vector<Symbol> sym_tab;
10
11 void print_err_to_lis(Error_T e, string s);
12 void validate_instruction(short int inst_id, vector<string> &toks);
13 void process_directive(short int d_id, vector<string> &rec, string p_tok =
    "");
14 void validate_tokens(vector<string> &toks);
15 bool Pass1(string src_fname);
16
17 #endif //_PASS1_H
```

### Source

```
1 #include "Pass1.h"
2
3 extern short loc_counter;
4 extern bool has_error;
5 extern vector<Inst> inst_set;
6 extern vector<Symbol> sym_tab;
7
8 //prints error message to LIS file
9 void print_err_to_lis(Error_T e, string s) {
10
11     string msg = "***** ";
12     switch (e) {
13     case NO_ERR:
14         ofs << "\t*****" << s << endl;
15         break;
16     case MISSING_OPERAND:
17         ofs << "\t***** Expected operand: " << s << endl;
18         break;
19     case ILLEGAL_OPERAND:
20         ofs << "\t***** Illegal operand: " << s << endl;
21         break;
22     case NUMBER_OF_OPERANDS_MISMATCH:
23         ofs << "\t ***** Too many/few number of operands: " << s << endl;
24         break;
25     case INVALID_OPERAND:
26         ofs << "\t ***** Invalid operands: " << s << endl;
27         break;
28     case INVALID_REGISTER:
29         ofs << "\t ***** Invalid REG: " << s << endl;
30         break;
31     case MISSING_INSTRUCTION_DIRECTIVE:
```

```

32     ofs << "\t ***** Expected INST/DIR: " << s << endl;
33     break;
34 case INVALID_LABEL_FORMAT:
35     ofs << "\t ***** Not valid label: " << s << endl;
36     break;
37 case UNDEFINED_SYMBOL:
38     ofs << "\t ***** Undefined operand(symbol): " << s << endl;
39     break;
40 case DUPLICATE_LABEL:
41     ofs << "\t ***** Duplicate LBL: " << s << endl;
42     break;
43 case INVALID_NUMBER:
44     ofs << "\t ***** Invalid Number: " << s << endl;
45     break;
46 case INVALID_RECORD:
47     ofs << "\t ***** Invalid Record: " << s << endl;
48     break;
49 default:
50     break;
51 }
52 }
53
54 /* validates the operands of an instruction - if not, error message is
   written to the LIS file */
55 void validate_instruction(short int inst_id, vector<string> &toks) {
56     vector<string> operands = {};
57     auto ops = toks[1]; //first token is the instruction and 2nd token
   holds operand(s)
58     stringstream ss(ops); //to split into separate operands from the token
59     string tok;
60     while (getline(ss, tok, ',')) { //operands are separated by comma ','
61         operands.push_back(tok);
62     }
63     auto ex_ops = inst_set[inst_id].expected_operands;
64     if (operands.size() != ex_ops.size()) { //number of operands and
   expected number of operands for this inst is not same
65         Error_T e = NUMBER_OF_OPERANDS_MISMATCH;
66         string s = "Expected: " + to_string(ex_ops.size()) + " Has: " +
            to_string(operands.size()) + " operands ";
67         print_err_to_lis(e, s);
68         has_error = true;
69     } else { // validate each operand - either register, or numeric or label
   type and handle accordingly
70         for (unsigned short int i = 0; i < operands.size(); ++i) {
71             auto op = operands[i];
72             if (is_register(op) != INVALID_INDEX && (ex_ops[i] == IDR || ex_ops[
   i] == R || ex_ops[i] == CON_R)) { //valid operand - do nothing
73                 continue;
74             } else if (is_numeric(op)) { //numeric operand
75                 //obtain value of the operand
76                 if (ex_ops[i] == L10 || ex_ops[i] == L13) {
77                     print_err_to_lis(INVALID_OPERAND, "Only labels are permitted for
   branch target");
78                     has_error = true;

```

```

79     }
80     short int r;
81     Error_T e = str2int(op, r);
82     if (e != NO_ERR) {
83         if (r > INT16_MAX || r < INT16_MIN) {
84             string s = "Too large or small value..";
85             print_err_to_lis(e, s);
86             has_error = true;
87         } else if ((r > UINT8_MAX || r < BYTE_MIN) && ex_ops[i] == BYTE)
88             {
89                 string s = "BYTE value should be (0,255)";
90                 print_err_to_lis(INVALID_OPERAND, s);
91                 has_error = true;
92             } else if ((r != 0 || r != 1 || r != 2 || r != 8 || r != 16 || r
93                 != 32 || r != -1) && ex_ops[i] == CON_R) {
94                 print_err_to_lis(INVALID_NUMBER, "CON value should be [0, 1,2,
95                     8,16, 32 or -1]");
96                 has_error = true;
97             } else if ((r < 0 || r > 15) && ex_ops[i] == SA) {
98                 print_err_to_lis(INVALID_OPERAND, "SA value should be (0,15)")
99                 ;
100                 has_error = true;
101             } else if ((r < 0 || r > 7) && ex_ops[i] == TCFC) {
102                 print_err_to_lis(INVALID_OPERAND, "TC/FC value should be (0,7)
103                     ");
104                 has_error = true;
105             }
106         } else {
107             if ((ex_ops[i] == IDR || ex_ops[i] == R)) { //REG expected but
108                 number
109                 print_err_to_lis(INVALID_REGISTER, ">" + op + "<");
110                 has_error = true;
111             }
112         } else {
113             //label
114             if ((ex_ops[i] == IDR || ex_ops[i] == R || ex_ops[i] == CON_R)) {
115                 //REG expected but label
116                 print_err_to_lis(INVALID_REGISTER, ">" + op + "<");
117                 has_error = true;
118             }
119             //validate label name and check if the label is in sym_tab else
120             store it
121             else if (is_valid_label_name(op)) {
122                 if (is_label_in_sym_tab(op) == INVALID_INDEX) { //not in sym_tab

```

```

        - store
123     sym_tab.insert(sym_tab.begin(), Symbol{op, "UNK", -1});
124     } else {
125         continue;
126     }
127
128     } else {
129         print_err_to_lis(INVALID_LABEL_FORMAT, "Invalid Label name..");
130         has_error = true;
131     }
132 }
133 }
134 }
135 loc_counter += 2; //each instruction needs 2-bytes
136 }
137 /* process ALIGN directive */
138 void handleDirALIGN(vector<string> &ops) {
139     if (ops.size() != 0) { //has operand
140         print_err_to_lis(ILLEGAL_OPERAND, "directive ALIGN does not take an
            operand");
141         has_error = true;
142     } else {
143         if (loc_counter % 2 != 0) {
144             loc_counter++;
145         } //if odd increment the address
146     }
147 }
148
149 /* process BSS directive */
150 void handleDirBSS(vector<string> &ops) {
151     short int r;
152     if (ops.size() != 1) { //no operand or more than one operand
153         print_err_to_lis(NUMBER_OF_OPERANDS_MISMATCH, "BSS must have one and
            only one operand");
154         has_error = true;
155     } else {
156         if (is_numeric(ops[0])) {
157             Error_T e = str2int(ops[0], r);
158             if (e == NO_ERR) {
159                 loc_counter += r;
160             } else {
161                 print_err_to_lis(e, "BSS operand should be a valid number");
162                 has_error = true;
163             }
164         }
165     } else {
166         auto r2 = is_label_in_sym_tab(ops[0]);
167         if (r2 == INVALID_INDEX) { // not in symbol table - check name,
            store label in sym_tab and emit error
168             if (is_valid_label_name(ops[0])) {
169                 sym_tab.insert(sym_tab.begin(), Symbol{ops[0], "UNK", -1});
170             } else {
171                 print_err_to_lis(INVALID_LABEL_FORMAT, "");
172                 has_error = true;

```

```

173         }
174
175     } else {
176         loc_counter += sym_tab[r2].value;
177     }
178 }
179 }
180 }
181
182 /* process BSS directive */
183 void handleDirBYTE(vector<string> &ops) {
184     short int r;
185     if (ops.size() != 1) { //no operand or more than one operand
186         print_err_to_lis(NUMBER_OF_OPERANDS_MISMATCH, "BYTE must have one and
187             only one operand");
188         has_error = true;
189     } else {
190         if (is_numeric(ops[0])) {
191             Error_T e = str2int(ops[0], r);
192             if (e == NO_ERR && (r < BYTE_MIN || r > INT8_MAX)) {
193                 print_err_to_lis(INVALID_OPERAND, "BYTE must be 8-bit size (0,255)
194                     ");
195                 has_error = true;
196             } else if (e != NO_ERR) { //str2err could not convert
197                 print_err_to_lis(e, " Invalid operand for BYTE directive ");
198                 has_error = true;
199             } else { //no error and valid BYTE size
200             }
201         } else { //operand is not a number - consider a label
202             auto r2 = is_label_in_sym_tab(ops[0]);
203             if (r2 == INVALID_INDEX) { // not in symbol table - check name,
204                 store label in sym_tab and emit error
205                 if (is_valid_label_name(ops[0])) {
206                     sym_tab.insert(sym_tab.begin(), Symbol{ops[0], "UNK", -1});
207                 } else {
208                     print_err_to_lis(INVALID_LABEL_FORMAT, "");
209                     has_error = true;
210                 }
211             } else if ((sym_tab[r2].value < BYTE_MIN || sym_tab[r2].value >
212                 INT8_MAX)) { //label in sym_tab
213                 print_err_to_lis(INVALID_OPERAND, "BYTE must be 8-bit size (0,255)
214                     ");
215                 has_error = true;
216             }
217         }
218     }
219 }
220
221 /* process END directive */
222 void handleDirEND(vector<string> &ops) {
223     short int r;
224     if (ops.size() == 1) {
225         Error_T e = str2int(ops[0], r);
226         if (e != NO_ERR) { //not a valid number - may be label name

```



```

222     auto r1 = is_label_in_sym_tab(ops[0]);
223     if (r1 == INVALID_INDEX) { // not in symbol table - check name,
        store label in sym_tab and emit error
224         if (is_valid_label_name(ops[0])) {
225             sym_tab.insert(sym_tab.begin(), Symbol{ops[0], "UNK", -1});
226         } else {
227             print_err_to_lis(INVALID_LABEL_FORMAT, "");
228             has_error = true;
229         }
230     }
231 }
232 }
233 }
234
235 /* process EQU directive */
236 void handleDirEQU(vector<string> &ops, string &p_tok) {
237     short int r;
238     if (p_tok.empty()) { //preceding token must be label but not present
239         print_err_to_lis(UNDEFINED_SYMBOL, "EQU directive must be preceded by
            a LBL");
240         has_error = true;
241     }
242 }
243
244 else if (ops.size() != 1) {
245     print_err_to_lis(MISSING_OPERAND, "EQU must have an operand");
246     has_error = true;
247 } else if (ops.size() == 1) { // looks fine - operand could be a value
    or a register
248     auto r1 = is_label_in_sym_tab(p_tok);
249
250     auto rr = is_register(ops[0]);
251     if (rr == INVALID_INDEX && !is_numeric(ops[0])) { //operand is neither
        a register nor a value - error
252         print_err_to_lis(INVALID_OPERAND, "Operand of EQU must be a value or
            a REG");
253         has_error = true;
254     } else if (rr != INVALID_INDEX && !sym_tab[r1].type.compare("UNK")) {
        //label is a REG with the corresponding REG value
255         cout << sym_tab[r1].type << "\tvalue " << sym_tab[r1].value << endl;
256         sym_tab[r1].type = "REG";
257         sym_tab[r1].value = sym_tab[rr].value;
258     } else { //operand is numeric
259
260         Error_T e = str2int(ops[0], r);
261         if (e != NO_ERR) { //has error in the value
262             print_err_to_lis(e, "Operand of EQU is not valid");
263             has_error = true;
264         } else {
265
266             if (!sym_tab[r1].type.compare("UNK")) {
267                 sym_tab[r1].type = "LBL";
268                 sym_tab[r1].value = r;
269             }

```

```

270     }
271 }
272 }
273 }
274
275 /* process ORG directive */
276 void handleDirORG(vector<string> &ops) {
277     short int r;
278     if (ops.size() != 1) {
279         print_err_to_lis(INVALID_OPERAND, "ORG should have an operand");
280         has_error = true;
281     } else {
282         if (is_numeric(ops[0])) {
283             Error_T e = str2int(ops[0], r);
284             if (e == NO_ERR) {
285                 loc_counter = r;
286             } else {
287                 print_err_to_lis(INVALID_NUMBER, "ORG operand should be a valid
                number");
288                 has_error = true;
289             }
290         } else {
291             print_err_to_lis(INVALID_OPERAND, "ORG operand should be a valid
                number");
292             has_error = true;
293         }
294     }
295 }
296
297 /* process WORD directive */
298 void handleDirWORD(vector<string> &ops) {
299     short int r;
300     if (ops.size() != 1) { //no operand or more than one operand
301         print_err_to_lis(NUMBER_OF_OPERANDS_MISMATCH, "WORD must have an
                operand");
302         has_error = true;
303     } else {
304
305         if (is_numeric(ops[0])) {
306             Error_T e = str2int(ops[0], r);
307             if (e == NO_ERR && (r < BYTE_MIN || r > UINT16_MAX)) {
308                 print_err_to_lis(INVALID_OPERAND, "WORD must be 16-bit size
                (0,65535)");
309                 has_error = true;
310             } else if (e != NO_ERR) { //str2err could not convert
311                 print_err_to_lis(e, " Invalid operand for WORD directive ");
312                 has_error = true;
313             } else { //no error and valid BYTE size
314             }
315         } else { //operand is not a number - consider a label
316             auto r2 = is_label_in_sym_tab(ops[0]);
317             if (r2 == INVALID_INDEX) { // not in symbol table - check name,
                store label in sym_tab and emit error
318                 if (is_valid_label_name(ops[0])) {

```

```

319         sym_tab.insert(sym_tab.begin(), Symbol{ops[0], "UNK", -1});
320     } else {
321         print_err_to_lis(INVALID_LABEL_FORMAT, "");
322         has_error = true;
323     }
324     } else if ((sym_tab[r2].value < BYTE_MIN || sym_tab[r2].value >
        UINT16_MAX)) { //label in sym_tab
325         print_err_to_lis(INVALID_OPERAND, "WORD must be 8-bit size (0,255)
            ");
326         has_error = true;
327     }
328 }
329 }
330 }
331
332 /* process the directive based on the index in directives */
333 void process_directive(short int d_id, vector<string> &rec, string p_tok)
    {
334     /****** */
335     vector<string> operands = {};
336     if (rec.size() == 2) { //has operand
337         auto ops = rec[1];
338         /* split different operands - separated by comma */
339         stringstream ss(ops); //
340         string tok;
341         while (getline(ss, tok, ',')) {
342             operands.push_back(tok);
343         }
344     }
345     directiveIndexes di = static_cast<directiveIndexes>(d_id);
346
347     switch (di) {
348     case dirALIGN:
349         handleDirALIGN(operands);
350         break;
351     case dirBSS:
352         handleDirBSS(operands);
353         break;
354     case dirBYTE:
355         handleDirBYTE(operands);
356         loc_counter += BYTE_INCREASE;
357         break;
358     case dirEND:
359         handleDirEND(operands);
360         break;
361     case dirEQU:
362         handleDirEQU(operands, p_tok);
363         break;
364     case dirORG:
365         handleDirORG(operands);
366         break;
367     case dirWORD: //6: //WORD
368         handleDirWORD(operands);
369         loc_counter += WORD_INCREASE;

```

```

370     break;
371     default:
372         break;
373     }
374     return;
375 }
376
377 /*function to validate tokens in a record for pass 1*/
378 void validate_tokens(vector<string> &toks) {
379
380     if (toks.size() > 3) { //too many tokens in a record
381         print_err_to_lis(INVALID_RECORD, "too many tokens ");
382         has_error = true;
383         return;
384     }
385
386     //consider 1st token as instruction
387     short int id = check_if_instruction(toks[0]);
388
389     if (id != INVALID_INDEX) { //an instruction - next token must present
390         and operand(s)
391         validate_instruction(id, toks);
392     } else { //either directive or label
393         id = check_if_directive(toks[0]);
394
395         if (id != INVALID_INDEX) { // a directive found - there may or may not
396             have operand(s)
397             if (id == 4) { //EQU should not be here
398                 print_err_to_lis(UNDEFINED_SYMBOL, "EQU should be preceded by LBL"
399                     );
400                 has_error = true;
401             } else {
402                 process_directive(id, toks);
403             }
404         } else { // this token is a label - following token must be INST/DIR,
405             if any
406             if (is_valid_label_name(toks[0])) { //valid name - go ahead
407                 id = is_label_in_sym_tab(toks[0]);
408                 if (id == INVALID_INDEX) { //no label in sym_tab with this name -
409                     so insert
410                     sym_tab.insert(sym_tab.begin(), Symbol{toks[0], "LBL",
411                         loc_counter});
412                 } else if (!sym_tab[id].type.compare("UNK")) { //UNK label found -
413                     change type and value
414                     sym_tab[id].type = "LBL";
415                     sym_tab[id].value = loc_counter;
416                 } else { //duplicate label
417                     print_err_to_lis(DUPLICATE_LABEL, "");
418                     has_error = true;
419                 }
420             } else { //invalid label name

```

```

417         print_err_to_lis(INVALID_LABEL_FORMAT, "Invalid Label name..");
418         has_error = true;
419     }
420     if (toks.size() > 1) { //more tokens are there and has to be INST/
        DIR
421         string prev_tok = toks[0];
422         toks.erase(toks.begin()); //erase first token
423         id = check_if_instruction(toks[0]);
424         if (id != INVALID_INDEX) { //an instruction - next token must
            present and operand(s)
425             validate_instruction(id, toks);
426         } else { //either directive or label
427             id = check_if_directive(toks[0]);
428             if (id != INVALID_INDEX) { // a directive found - there may or
                may not have operand(s)
429                 process_directive(id, toks, prev_tok);
430             } else { // this tok is a label - again! -Error
431                 print_err_to_lis(MISSING_INSTRUCTION_DIRECTIVE, "Expected INST
                    /DIR after a LBL");
432                 has_error = true;
433             }
434         }
435     }
436     /****** */
437 }
438 }
439 }
440 }
441
442 /* conducts pass1 */
443 bool Pass1(string src_fname) {
444     // read the src file line by line and process it
445     ifstream ifs;
446     ifs.open(src_fname);
447     string line;           //to hold the content of a line
448     short int n_line = 0; //corresponding line number in the src file
449     if (ifs.is_open()) {
450         //bool no_err = true;
451         //read each line of src file and process it
452         while (getline(ifs, line)) {
453             n_line++;
454             ofs << "\t" << n_line << "\t" << line << endl;
455             if (line.empty()) {
456                 continue;
457             } else {
458                 //get tokens from the line
459                 vector<string> tokens = {};
460                 get_tokens(line, tokens);
461                 if (tokens.size() > 0) {
462                     validate_tokens(tokens);
463                 } else {
464                     continue;
465                 }
466             }

```

```

467     }
468
469     ifs.close();
470
471 } else {
472     cout << "Could not open source file: " << src_fname << endl;
473     return false;
474 }
475
476 for (unsigned short int i = 0; i < sym_tab.size(); ++i) {
477     if (!sym_tab[i].type.compare("UNK")) { // found an 'UNK' entry
478         has_error = true;
479         break;
480     }
481 }
482
483 if (has_error) {
484     //print sym_tab
485     cout << "Has Error...printing to LIS" << endl;
486     ofs << "First pass error....assembly terminated...." << endl;
487
488     ofs << "\n **** Symbol Table ***" << std::endl;
489     ofs << "Name\t\tType\tValue\tDecimal" << std::endl;
490
491     for (auto s : sym_tab) {
492         stringstream ss;
493         ss << std::uppercase << std::setfill('0') << std::setw(4) << std::
            hex << s.value;
494         string hex_v;
495         if (s.value == -1) {
496             hex_v = "FFFF";
497         } else {
498             string t = ss.str();
499             hex_v = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
500         }
501         ofs << s.name << "\t\t" << s.type << "\t" << hex_v << "\t" << s.
            value << endl;
502     }
503     ofs.close();
504     return false;
505 } else {
506     cout << "No Error in Pass1...starting Pass 2" << endl;
507     // ofs<<"First pass error....assembly terminated...."<<endl;
508
509     ofs << "\n **** Symbol Table ***" << std::endl;
510     ofs << "Name\t\tType\tValue\tDecimal" << std::endl;
511
512     for (auto s : sym_tab) {
513         stringstream ss;
514         ss << std::uppercase << std::setfill('0') << std::setw(4) << std::
            hex << s.value;
515         string t = ss.str();
516         string hex_v = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
517

```

```
518         ofs << s.name << "\t\t" << s.type << "\t" << hex_v << "\t" << s.  
            value << endl;  
519     }  
520     ofs.close();  
521     return true;  
522 }  
523 }
```

## 2 Testing Files

### 2.1 Input ASM

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
```

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
```

#### **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
```

#### **test1.asm**

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

#### **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
11
12 ; Invalid operand
13 byte '123'
14
15 ; Unexpected operand
16 align R1
17
18 ; Missing '#' or '$':
19 org 100
20
21 ; Symbols not in symbol table:
22 ld r0,r1
23
24 ; Unknown instruction:
25 Loop lda r3
26
27 ; ">"< not supported, should be ">"<:
28 Data byte "x"
29
30 ; Operands in wrong order:
31 add R0,#1
32
33 ; Unknown instruction (beqx) treated as a label. "Loop" then examined
34 ; and treated as an instruction:
```

```

35     beqx  loop
36
37 ; Unknown register
38     swap  R1,R8
39
40 ; Bad BSS value - unknown value
41     BSS ArraySize
42
43 ; Duplicate label
44 START equ #12
45
46 ; Start not defined (put in symbol table as UNK - unknown symbol):
47     end Start

```

## 2.2 Output LIS

a1.asm

```

1  .ASM file: a1.asm
2
3
4
5  1
6  2 ;
7  3 ; Sample XM2 Assembler program to increment a number by 1
8  4 ; A. N. O'Nymous
9  5 ; 7 May 2019
10 6 ;
11 7 ; Data space
12 8 org $80
13 9 Number word #0
14 10 ;
15 11 ; Code
16 12 org $1000
17 13 Start
18 14 ;
19 15 ; R0 = Address of Number
20 16 ;
21 17 movlz Number,R0
22 18 ;
23 19 ; R1 = Value stored in Number [R0]
24 20 ;
25 21 ld R0,R1
26 22 ;
27 23 ; Increment R1
28 24 ;
29 25 add #1,R1
30 26 ;
31 27 ; Number [R0] = R1
32 28 ;
33 29 st R1,R0
34 30 ;
35 31 ; End of program
36 32 ; Specifying first executable location (Start)

```

```

37 33 ;
38 34 end Start
39
40 **** Symbol Table ***
41 Name      Type  Value Decimal
42 Start     LBL 1000 4096
43 Number    LBL 0080 128
44 R7        REG 0007 7
45 R6        REG 0006 6
46 R5        REG 0005 5
47 R4        REG 0004 4
48 R3        REG 0003 3
49 R2        REG 0002 2
50 R1        REG 0001 1
51 R0        REG 0000 0

```

#### a2.asm

```

1  .ASM file: a2.asm
2
3
4
5 1 org $80
6 2 Label01 word #0
7 3 org $FF00
8 4 Label02 word #1
9 5 org $100
10 6 Label03 MOVLZ Label01,R0
11 7 MOVLS Label02,R1
12 8 LD R0,R0
13 9 LD R1,R2
14 10 Label04 ADD R0,R2
15 11 CMP #16,R2
16 12 BNE Label04
17 13 ST R2,R1
18 14 Done BRA Done
19 15 END Label03
20
21 **** Symbol Table ***
22 Name      Type  Value Decimal
23 Done      LBL 0110 272
24 Label04    LBL 0108 264
25 Label03    LBL 0100 256
26 Label02    LBL FF00 -256
27 Label01    LBL 0080 128
28 R7        REG 0007 7
29 R6        REG 0006 6
30 R5        REG 0005 5
31 R4        REG 0004 4
32 R3        REG 0003 3
33 R2        REG 0002 2
34 R1        REG 0001 1
35 R0        REG 0000 0

```

#### example1.asm

```

1  .ASM file: example1.asm
2
3
4
5  1 SIZE equ $26
6  2 CAP_A equ 'A'
7  3
8  4 org #80
9  5 BASE bss SIZE
10 6 org $1000
11 7
12 8 Start movlz CAP_A,R0
13 9
14
15 **** Symbol Table ***
16 Name      Type  Value Decimal
17 Start     LBL 1000  4096
18 BASE      LBL 0050   80
19 CAP_A     LBL 0000    0
20 SIZE      LBL 0000    0
21 R7        REG 0007    7
22 R6        REG 0006    6
23 R5        REG 0005    5
24 R4        REG 0004    4
25 R3        REG 0003    3
26 R2        REG 0002    2
27 R1        REG 0001    1
28 R0        REG 0000    0

```

#### test1.asm

```

1  .ASM file: test1.asm
2
3
4
5  1
6  2 ; Operands in wrong order:
7  3  ld R0,#1
8  ***** Invalid REG: >#1<
9  4  ldr R5,#-60,r0
10 ***** Invalid REG: >r0<
11 First pass error....assembly terminated....
12
13 **** Symbol Table ***
14 Name      Type  Value Decimal
15 R7        REG 0007    7
16 R6        REG 0006    6
17 R5        REG 0005    5
18 R4        REG 0004    4
19 R3        REG 0003    3
20 R2        REG 0002    2
21 R1        REG 0001    1
22 R0        REG 0000    0

```

#### FirstPassErrorsExample.asm

```

1  .ASM file: FirstPassErrorsExample.asm
2
3
4
5  1 ;
6  2 ; Examples of some first pass errors
7  3 ; - Look at console output and .LIS file
8  4 ;
9  5 ; L. Hughes
10 6 ; 29 May 2019 - Revised for XM2
11 7 ; 11 May 2018 - original
12 8
13 9 ; Missing operand:
14 10 START equ
15 ***** Expected operand: EQU must have an operand
16 11
17 12 ; Invalid operand
18 13 byte '123'
19 ***** Invalid operands: BYTE must be 8-bit size (0,255)
20 14
21 15 ; Unexpected operand
22 16 align R1
23 ***** Illegal operand: directive ALIGN does not take an operand
24 17
25 18 ; Missing '#' or '$':
26 19 org 100
27 ***** Invalid operands: ORG operand should be a valid number
28 20
29 21 ; Symbols not in symbol table:
30 22 ld r0,r1
31 ***** Invalid REG: >r0<
32 ***** Invalid REG: >r1<
33 23
34 24 ; Unknown instruction:
35 25 Loop lda r3
36 ***** Expected INST/DIR: Expected INST/DIR after a LBL
37 26
38 27 ; >"< not supported, should be >'<:
39 28 Data byte "x"
40 ***** Not valid label:
41 29
42 30 ; Operands in wrong order:
43 31 add R0,#1
44 ***** Invalid REG: >#1<
45 32
46 33 ; Unknown instruction (beqx) treated as a label. "Loop" then
    examined
47 34 ; and treated as an instruction:
48 35 beqx loop
49 ***** Expected INST/DIR: Expected INST/DIR after a LBL
50 36
51 37 ; Unknown register
52 38 swap R1,R8
53 ***** Invalid REG: >R8<

```

```

54 39
55 40 ; Bad BSS value - unknown value
56 41 BSS ArraySize
57 42
58 43 ; Duplicate label
59 44 START equ #12
60 ***** Duplicate LBL:
61 45
62 46 ; Start not defined (put in symbol table as UNK - unknown symbol):
63 47 end Start
64 First pass error....assembly terminated....
65
66 ***** Symbol Table ***
67 Name      Type  Value Decimal
68 Start     UNK FFFF  -1
69 ArraySize UNK FFFF  -1
70 beqx      LBL 0006  6
71 Data      LBL 0003  3
72 Loop      LBL 0003  3
73 START     LBL 0000  0
74 R7        REG 0007  7
75 R6        REG 0006  6
76 R5        REG 0005  5
77 R4        REG 0004  4
78 R3        REG 0003  3
79 R2        REG 0002  2
80 R1        REG 0001  1
81 R0        REG 0000  0

```

Thank you for being so patient with me.