Assignment Pass 2

Sakif Fahmid Zaman
B00756635
Dalhousie University
ECED 3403

August 7, 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 Pass2.cpp main.cpp)
```

1.2 Main

```
#include "Globals.h"
#include "Pass1.h"
#include "Pass2.h"

int main(int argc, char *argv[]) {
  init_globals(argv[1]);
  if (Pass1(argv[1])) {
    if (Pass2(argv[1]))
      cout << "Assembly completed successfully..." << endl;
    else {
      cout << "Pass 2 failed... program terminates..." << endl;
    }
  } else
    cout << "Pass 1 failed... program terminates..." << endl;
  return 0;
}
```

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
```

```
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
```

```
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 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"
#include "Pass2.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:
```

```
27      ofs << "\t ***** Invalid operands: " << s << endl;
28      break;
29    case INVALID_REGISTER:
30      ofs << "\t ***** Invalid REG: " << s << endl;
31      break;
32    case MISSING_INSTRUCTION_DIRECTIVE:
33      ofs << "\t ***** Expected INST/DIR: " << s << endl;
34      break;
35    case INVALID_LABEL_FORMAT:
36      ofs << "\t ***** Not valid label: " << s << endl;
37      break;
38    case UNDEFINED_SYMBOL:
39      ofs << "\t ***** Undefined operand(symbol): " << s << endl;
40      break;
41    case DUPLICATE_LABEL:
42      ofs << "\t ***** Duplicate LBL: " << s << endl;
43      break;
44    case INVALID_NUMBER:
45      ofs << "\t ***** Invalid Number: " << s << endl;
46      break;
47    case INVALID_RECORD:
48      ofs << "\t ***** Invalid Record: " << s << endl;
49      break;
50    default:
51      break;
52    }
53  }
54
55  /* validates the operands of an instruction - if not, error message is
        written to the LIS file */
56  void validate_instruction(short int inst_id, vector<string> &toks) {
57    vector<string> operands = {};
58    auto ops = toks[1];   //first token is the instruction and 2nd token
        holds operand(s)
59    stringstream ss(ops); //to split into separate operands from the token
60    string tok;
61    while (getline(ss, tok, ',')) { //operands are separated by comma ','
62      operands.push_back(tok);
63    }
64    auto ex_ops = inst_set[inst_id].expected_operands;
65    if (operands.size() != ex_ops.size()) { //number of operands and
        expected number of operands for this inst is not same
66      Error_T e = NUMBER_OF_OPERANDS_MISMATCH;
67      string s = "Expected: " + to_string(ex_ops.size()) + " Has: " +
          to_string(operands.size()) + " operands ";
68      print_err_to_lis(e, s);
69      has_error = true;
70    } else { // validate each operand - either register, or numeric or label
        type and handle accordingly
71      for (unsigned short int i = 0; i < operands.size(); ++i) {
72        auto op = operands[i];
73        if (is_register(op) != INVALID_INDEX && (ex_ops[i] == IDR || ex_ops[
            i] == R || ex_ops[i] == CON_R)) { //valid operand - do nothinf
74          continue;
```

```
75        } else if (is_numeric(op)) { //numeric operand
76          //obtain value of the operand
77          if (ex_ops[i] == L10 || ex_ops[i] == L13) {
78            print_err_to_lis(INVALID_OPERAND, "Only labels are permitted for
                  branch targt");
79            has_error = true;
80          }
81          short int r;
82          Error_T e = str2int(op, r);
83          if (e != NO_ERR) {
84            if (r > INT16_MAX || r < INT16_MIN) {
85              string s = "Too large or small value..";
86              print_err_to_lis(e, s);
87              has_error = true;
88            } else if ((r > UINT8_MAX || r < BYTE_MIN) && ex_ops[i] == BYTE)
                  {
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
                  != 32 || r != -1) && ex_ops[i] == CON_R) {
93              print_err_to_lis(INVALID_NUMBER, "CON value should be [0, 1,2,
                    8,16, 32 or -1]");
94              has_error = true;
95            } else if ((r < 0 || r > 15) && ex_ops[i] == SA) {
96              print_err_to_lis(INVALID_OPERAND, "SA value should be (0,15)")
                    ;
97              has_error = true;
98            } else if ((r < 0 || r > 7) && ex_ops[i] == TCFC) {
99              print_err_to_lis(INVALID_OPERAND, "TC/FC value should be (0,7)
                    ");
100             has_error = true;
101           }
102         } else {
103           if ((ex_ops[i] == IDR || ex_ops[i] == R)) { //REG expected but
                  number
104             print_err_to_lis(INVALID_REGISTER, ">" + op + "<");
105             has_error = true;
106           }
107         }
108
109       } else if (is_cond(op) != INVALID_INDEX) { // cond
110         if (ex_ops[i] != COND_CEC) {
111           print_err_to_lis(INVALID_OPERAND, "COND operand is not valid..")
                  ;
112           has_error = true;
113         }
114       } else {
            //label
115         if ((ex_ops[i] == IDR || ex_ops[i] == R || ex_ops[i] == CON_R)) {
              //REG expected but label
116           print_err_to_lis(INVALID_REGISTER, ">" + op + "<");
117           has_error = true;
118         }
```

```
119
120         //validate label name and check if the label is in sym_tab else
                store it
121         else if (is_valid_label_name(op)) {
122
123           if (is_label_in_sym_tab(op) == INVALID_INDEX) { //not in sym_tab
                - store
124             sym_tab.insert(sym_tab.begin(), Symbol{op, "UNK", -1});
125           } else {
126             continue;
127           }
128
129         } else {
130           print_err_to_lis(INVALID_LABEL_FORMAT, "Invalid Label nmae..");
131           has_error = true;
132         }
133       }
134     }
135   }
136   loc_counter += 2; //each instruction needs 2-bytes
137 }
138 /* process ALIGN directive */
139 void handleDirALIGN(vector<string> &ops) {
140   if (ops.size() != 0) { //has operand
141     print_err_to_lis(ILLEGAL_OPERAND, "directive ALIGN does not take an
          operand");
142     has_error = true;
143   } else {
144     if (loc_counter % 2 != 0) {
145       loc_counter++;
146     } //if odd increment the address
147   }
148 }
149
150 /* process BSS directive */
151 void handleDirBSS(vector<string> &ops) {
152   short int r;
153   if (ops.size() != 1) { //no operand or more than one operand
154     print_err_to_lis(NUMBER_OF_OPERANDS_MISMATCH, "BSS must have one and
          only one operand");
155     has_error = true;
156   } else {
157     if (is_numeric(ops[0])) {
158       Error_T e = str2int(ops[0], r);
159       if (e == NO_ERR) {
160         loc_counter += r;
161       } else {
162         print_err_to_lis(e, "BSS operand should be a valid number");
163         has_error = true;
164       }
165
166     } else {
167       auto r2 = is_label_in_sym_tab(ops[0]);
168       if (r2 == INVALID_INDEX) { // not in symbol table - check name,
```

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

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

```
265             sym_tab[r1].value = r;
266         }
267       }
268     }
269   }
270
271   /* process ORG directive */
272   void handleDirORG(vector<string> &ops) {
273     short int r;
274     if (ops.size() != 1) {
275       print_err_to_lis(INVALID_OPERAND, "ORG should have an operand");
276       has_error = true;
277     } else {
278       if (is_numeric(ops[0])) {
279         Error_T e = str2int(ops[0], r);
280         if (e == NO_ERR) {
281           loc_counter = r;
282         } else {
283           print_err_to_lis(INVALID_NUMBER, "ORG operand should be a valid
                number");
284           has_error = true;
285         }
286       } else {
287         print_err_to_lis(INVALID_OPERAND, "ORG operand should be a valid
              number");
288         has_error = true;
289       }
290     }
291   }
292
293   /* process WORD directive */
294   void handleDirWORD(vector<string> &ops) {
295     short int r;
296     if (ops.size() != 1) { //no operand or more than one operand
297       print_err_to_lis(NUMBER_OF_OPERANDS_MISMATCH, "WORD must have an
            operand");
298       has_error = true;
299     } else {
300
301       if (is_numeric(ops[0])) {
302         Error_T e = str2int(ops[0], r);
303         if (e == NO_ERR && (r < BYTE_MIN || r > UINT16_MAX)) {
304           print_err_to_lis(INVALID_OPERAND, "WORD must be 16-bit size
                (0,65535)");
305           has_error = true;
306         } else if (e != NO_ERR) { //str2err could not convert
307           print_err_to_lis(e, " Invalid operand for WORD directive ");
308           has_error = true;
309         } else { //no error and valid BYTE size
310         }
311       } else { //operand is not a number - consider a label
312         auto r2 = is_label_in_sym_tab(ops[0]);
313         if (r2 == INVALID_INDEX) { // not in symbol table - check name,
              store label in sym_tab and emit error
```

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

```
365       loc_counter += WORD_INCREASE;
366       break;
367     default:
368       break;
369     }
370     return;
371   }
372
373   /*function to validate tokens in a record for pass 1*/
374   void validate_tokens(vector<string> &toks) {
375
376     if (toks.size() > 3) { //too many tokens in a record
377       print_err_to_lis(INVALID_RECORD, "too many tokens ");
378       has_error = true;
379       return;
380     }
381
382     //consider 1st token as instruction
383     short int id = check_if_instruction(toks[0]);
384
385     if (id != INVALID_INDEX) { //an instruction - next token must present
          and operand(s)
386
387       validate_instruction(id, toks);
388     } else { //either directive or label
389       id = check_if_directive(toks[0]);
390
391       if (id != INVALID_INDEX) { // a directive found - there may or may not
            have operand(s)
392         if (id == 4) {           //EQU should not be here
393           print_err_to_lis(UNDEFINED_SYMBOL, "EQU should be preceded by LBL"
              );
394           has_error = true;
395         } else {
396           process_directive(id, toks);
397         }
398
399       } else { // this token is a label - following token must be INST/DIR,
            if any
400
401         if (is_valid_label_name(toks[0])) { //valid name - go ahead
402           id = is_label_in_sym_tab(toks[0]);
403           if (id == INVALID_INDEX) { //no label in sym_tab with this name -
              so insert
404             sym_tab.insert(sym_tab.begin(), Symbol{toks[0], "LBL",
                loc_counter});
405           } else if (!sym_tab[id].type.compare("UNK")) { //UNK label found -
                change type and value
406             sym_tab[id].type = "LBL";
407             sym_tab[id].value = loc_counter;
408           } else { //duplicate label
409             print_err_to_lis(DUPLICATE_LABEL, "");
410             has_error = true;
411           }
```

```
412       } else { //invalid label name
413         print_err_to_lis(INVALID_LABEL_FORMAT, "Invalid Label nmae..");
414         has_error = true;
415       }
416       if (toks.size() > 1) { //more tokens are there and has to be INST/
              DIR
417         string prev_tok = toks[0];
418         toks.erase(toks.begin()); //erase first token
419         id = check_if_instruction(toks[0]);
420         if (id != INVALID_INDEX) { //an instruction - next token must
              present and operand(s)
421           validate_instruction(id, toks);
422         } else { //either directive or label
423           id = check_if_directive(toks[0]);
424           if (id != INVALID_INDEX) { // a directive found - there may or
                may not have operand(s)
425             process_directive(id, toks, prev_tok);
426           } else { // this tok is a label - again! -Error
427             print_err_to_lis(MISSING_INSTRUCTION_DIRECTIVE, "Expected INST
                  /DIR after a LBL");
428             has_error = true;
429           }
430         }
431
432         /****** */
433       }
434     }
435   }
436 }
437
438 /* conducts pass1 */
439 bool Pass1(string src_fname) {
440   // read the src file line by line and process it
441   ifstream ifs;
442   ifs.open(src_fname);
443   string line;          //to hold the content of a line
444   short int n_line = 0; //corresponding line number in the src file
445   if (ifs.is_open()) {
446     //bool no_err = true;
447     //read each line of src file and process it
448     while (getline(ifs, line)) {
449       n_line++;
450       ofs << "\t" << n_line << "\t" << line << endl;
451       if (line.empty()) {
452         continue;
453       } else {
454         //get tokens from the line
455         vector<string> tokens = {};
456         get_tokens(line, tokens);
457         if (tokens.size() > 0) {
458           validate_tokens(tokens);
459         } else {
460           continue;
461         }
```

```
462         }
463       }
464
465       ifs.close();
466
467     } else {
468       cout << "Could not open source file: " << src_fname << endl;
469       return false;
470     }
471
472     for (unsigned short int i = 0; i < sym_tab.size(); ++i) {
473       if (!sym_tab[i].type.compare("UNK")) { // found an 'UNK' entry
474         has_error = true;
475         break;
476       }
477     }
478
479     if (has_error) {
480       //print sym_tab
481       cout << "Has Error...printing to LIS" << endl;
482       ofs << "First pass error....assembly terminated...." << endl;
483
484       ofs << "\n ****  Symbol Table ***" << std::endl;
485       ofs << "Name\t\tType\tValue\tDecimal" << std::endl;
486
487       for (auto s : sym_tab) {
488         stringstream ss;
489         ss << std::uppercase << std::setfill('0') << std::setw(4) << std::
              hex << s.value;
490         string hex_v;
491         if (s.value == -1) {
492           hex_v = "FFFF";
493         } else {
494           string t = ss.str();
495           hex_v = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
496         }
497         ofs << s.name << "\t\t" << s.type << "\t" << hex_v << "\t" << s.
              value << endl;
498       }
499       ofs.close();
500       return false;
501     } else {
502       cout << "No Error in Pass1...starting Pass 2" << endl;
503       // ofs<<"First pass error....assembly terminated...."<<endl;
504       //ofs.close();
505
506       if (Pass2(src_fname)) {
507         return true;
508       } else {
509         cout << " Problem in Pass2.." << endl;
510         return false;
511       }
512     }
513 }
```

1.5 Pass 2

Header

```
1  #pragma once
2  #ifndef _PASS2_H
3  #define _PASS2_H
4
5  #include "Globals.h"
6
7  #define SET_BIT 1
8  #define CLEAR_BIT 0
9  #define INSTRUCTION_LEN 2
10 #define BSS_OPCODE 0
11 #define MAX_BYTE_SREC_DATA 28 //maximum number of byte in srec data field
12 #define ADDR_BYTES_SREC 2     //number of byte in address fieled
13 #define CHKSUM_BYTE_SREC 1    //number of byte in checksumS fieled
14
15 extern short loc_counter;
16 extern bool has_error;
17 extern vector<Inst> inst_set;
18 extern vector<Symbol> sym_tab;
19
20 /*********** Instruction Set Structs for Pass 2  **************/
21
22 enum PrePostIncrDecr {
23   None,
24   PreIncrement,
25   PreDecrement,
26   PostIncrement,
27   PostDecrement
28 };
29
30 enum InstType {
31   MemAccessLD,
32   MemAccessST,
33   MemAccessRelLD,
34   MemAccessRelST,
35   RegInit,
36   Branch13,
37   Branch10,
38   Cex,
39   Arith,
40   RegExchange,
41   OneAddr
42 };
43
44 /* Register direct and register direct with pre or post auto-increment or
      auto-decrement Inst - Section 6.1.1 (LD and ST) used when expected
      operands of type - (IDR,R) or (R,IDR) */
45 struct MemAccessInstruction {
46   unsigned _dst : 3;      //dest register
47   unsigned _src : 3;      //src register
48   unsigned _wordByte : 1; //word or byte flag
49   unsigned _inc : 1;      //increment flag
```

```
50   unsigned _dec : 1;      //decrement flag
51   unsigned _prpo : 1;     //pre/post flag
52   unsigned _opCode : 6;   //opcode for ld or st instruction
53 };
54
55 union MemAccessOverlay {
56   unsigned short sh;
57   MemAccessInstruction inst;
58 };
59
60 /* Register Relative memory access Inst - Section 6.1.2 (LDR and STR) used
       when expected operands of type - (R,OFFSET,R) or (R,R,OFFSET) */
61 struct MemAccessRelativeInstruction {
62   unsigned _dst : 3;      //dest register
63   unsigned _src : 3;      //src register
64   unsigned _wordByte : 1; //word or byte flag
65   short int _offset : 7;  //offset value
66   unsigned _opCode : 2;   //opcode for ldr or str instruction
67 };
68
69 union MemAccessRelativeOverlay {
70   unsigned short sh;
71   MemAccessRelativeInstruction inst;
72 };
73
74 /* Register Initialization Instruction - Section 6.2 used when expected
      operands of type - (BYTE,R)  */
75 struct RegisterInitInstruction {
76   unsigned _dst : 3;    //dest register
77   unsigned _Byte : 8;   //8-bit value
78   unsigned _opCode : 5; //opcode for the instruction
79 };
80
81 union RegisterInitOverlay {
82   unsigned short sh;
83   RegisterInitInstruction inst;
84 };
85
86 /* Branching with 13-bit offset Inst - Section 6.3 (BL) used when expected
       operands of type - L13 */
87 struct Br13Instruction {
88   short int _offset : 13; //offset value
89   short int _opCode : 3;  //opcode for BL instruction
90 };
91
92 union Br13Overlay {
93   unsigned short sh;
94   Br13Instruction inst;
95 };
96
97 /* Branching with 10-bit offset Inst - Section 6.3 (Branching other than
      BL) used when expected operands of type - L10 */
98 struct Br10Instruction {
99   short int _offset : 10; //offset value
```

```
100   short int _opCode : 6;  //opcode for branching instruction except BL
101 };
102
103 union Br10Overlay {
104   unsigned short sh;
105   Br10Instruction inst;
106 };
107
108 /* Conditional Execution Inst - Section 6.4 (CEX) expected operands (
        COND_CEC,TCFC,TCFC) */
109 struct CexInstruction {
110   unsigned _fc : 3;
111   unsigned _tc : 3;
112   unsigned _condCec : 4;
113   unsigned _opCode : 6;
114 };
115
116 union CexOverlay {
117   unsigned short sh;
118   CexInstruction inst;
119 };
120
121 /* 2-operand (REG-REG or CON-REG) Inst - Section 6.5 and Reg Exchange MOV
        (.B or .W) expected operands (CON_R,R) */
122 struct ArithInstruction {
123   unsigned _dst : 3;
124   unsigned _src : 3;
125   unsigned _wordByte : 1;
126   unsigned _regCon : 1;
127   unsigned _opCode : 8;
128 };
129
130 union ArithOverlay {
131   unsigned short sh;
132   ArithInstruction inst;
133 };
134
135 /* Register exchange (REG-REG) Inst - Section 6.6 SWAP, except MOV(.B or .
        W) expected operands (R,R) */
136 struct RegExchangeInstruction {
137   unsigned _dst : 3;
138   unsigned _src : 3;
139   unsigned _wordByte : 1;
140   unsigned _regCon : 1;
141   unsigned _opCode : 8;
142 };
143
144 union RegExchangeOverlay {
145   unsigned short sh;
146   RegExchangeInstruction inst;
147 };
148
149 /* Single register Inst - Section 6.7 expected operands (R) */
150 struct OneAddrInstruction {
```

```
151   unsigned _dst : 3;
152   unsigned _zeros : 3;
153   unsigned _wordByte : 1;
154   unsigned _zero : 1;
155   unsigned _opCode : 8;
156 };
157
158 union OneAddrOverlay {
159   unsigned short sh;
160   OneAddrInstruction inst;
161 };
162
163 /*<<<<<<<<<<>>>>>>>>>>>>>>>>>>>> */
164 enum SRec_Type {
165   S0,
166   S1,
167   S9
168 };
169
170 struct SRec {
171   SRec_Type _type;
172   string _count;    //char _count[2];
173   string _addr;     //char _addr[4];
174   string _data;     //should be max 28 byte
175   string _checksum; //char _checksum[2];
176 };
177
178 //std::filesystem::path getexepath();
179 void proc_instruction(short int inst_id, vector<string> &toks);
180 void proc_directive(short int d_id, vector<string> &rec, string p_tok = ""
      );
181 void proc_tokens(vector<string> &toks);
182 bool Pass2(string src_fname);
183
184 #endif //_PASS2_H
```

Source

```
 1 #ifdef WINDOWS
 2 #include <windows.h>
 3 #else
 4 #include <dirent.h>
 5 #include <limits.h>
 6 #include <unistd.h>
 7 #endif
 8 #include "Pass2.h"
 9 #include <stdlib.h>
10
11 extern short loc_counter;
12 extern bool has_error;
13 extern vector<Inst> inst_set;
14 extern vector<Symbol> sym_tab;
15 ofstream of_xme;
16 string s1str, s9str;
17
```

```
18 InstType getInstType(vector<Operand_T> ops) {
19   if (ops.size() == 1 && ops[0] == L10) {
20     return Branch10;
21   } else if (ops.size() == 1 && ops[0] == L13) {
22     return Branch13;
23   } else if (ops.size() == 1 && (ops[0] == R || ops[0] == SA)) {
24     return OneAddr;
25   } else if (ops.size() == 2 && ops[0] == CON_R && ops[1] == R) {
26     return Arith;
27   } else if (ops.size() == 2 && ops[0] == R && ops[1] == R) {
28     return RegExchange;
29   } else if (ops.size() == 2 && ops[0] == IDR && ops[1] == R) {
30     return MemAccessLD;
31   } else if (ops.size() == 2 && ops[0] == R && ops[1] == IDR) {
32     return MemAccessST;
33   } else if (ops.size() == 2 && ops[0] == BYTE && ops[1] == R) {
34     return RegInit;
35   } else if (ops.size() == 3 && ops[0] == COND_CEC && ops[1] == TCFC &&
       ops[2] == TCFC) {
36     return Cex;
37   } else if (ops.size() == 3 && ops[0] == R && ops[1] == OFFSET && ops[2]
       == R) {
38     return MemAccessRelLD;
39   } else {
40     return MemAccessRelST;
41   }
42 }
43
44 /* generate opcode for Br13 instruction BL */
45 void handleBranch13(short int inst_id, vector<string> &ops, unsigned short
       &opcode) {
46   // find label index from sym_tab
47   auto lbl_index = is_label_in_sym_tab(ops[0]);
48   short int offset = sym_tab[lbl_index].value - loc_counter - 2;
49   auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
50   Br13Overlay instCode;
51   instCode.inst._offset = offset >> 1;
52   instCode.inst._opCode = opc;
53   opcode = instCode.sh;
54 }
55
56 /* generate opcode for Br10 instruction branching other than BL */
57 void handleBranch10(short int inst_id, vector<string> &ops, unsigned short
       &opcode) {
58   // find label index from sym_tab
59   auto lbl_index = is_label_in_sym_tab(ops[0]);
60   short int offset = sym_tab[lbl_index].value - loc_counter - 2;
61   auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
62   Br10Overlay instCode;
63   instCode.inst._offset = offset >> 1;
64   instCode.inst._opCode = opc;
65   opcode = instCode.sh;
66 }
67
```

```
68  /* generate opcode for Arithmatic (REG/CON, REG) instruction */
69  void handleArith(short int inst_id, vector<string> &ops, unsigned short &
        opcode) {
70    // find label index from sym_tab
71    bool is_const = false;
72    short int lbl_index;
73    auto src_index = is_register(ops[0]);
74    auto dst_index = is_register(ops[1]);
75    short int v;
76    if (src_index == INVALID_INDEX) {
77      // src is not a REG but CONST
78      is_const = true;
79      // find the value of the CONST then
80      if (is_numeric(ops[0])) {
81        str2int(ops[0], v);
82      } else {
83        lbl_index = is_label_in_sym_tab(ops[0]);
84        v = sym_tab[lbl_index].value;
85      }
86    }
87    bool is_byte = false;
88    // check last 2 chars to check whether ".B" is present in instruction
89    string last2chars = inst_set[inst_id].mnemonic.substr(inst_set[inst_id].
          mnemonic.size() - 2);
90    if (last2chars.compare(".B") == 0) {
91      // instruction ends with .B
92      is_byte = true;
93    }
94    auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
95    ArithOverlay instCode;
96    // populate
97    instCode.inst._dst = sym_tab[dst_index].value;
98    if (is_const) {
99      short int src_value = 0;
100     if (v == 0) {
101       src_value = 0;
102     } else if (v == 1) {
103       src_value = 1;
104     } else if (v == 2) {
105       src_value = 2;
106     } else if (v == 4) {
107       src_value = 3;
108     } else if (v == 8) {
109       src_value = 4;
110     } else if (v == 16) {
111       src_value = 5;
112     } else if (v == 32) {
113       src_value = 6;
114     } else if (v == -1) {
115       src_value = 7;
116     }
117
118     instCode.inst._src = src_value;
119     instCode.inst._regCon = SET_BIT;
```

```
120   } else {
121     instCode.inst._src = sym_tab[src_index].value;
122     instCode.inst._regCon = CLEAR_BIT;
123   }
124
125   if (is_byte) {
126     instCode.inst._wordByte = SET_BIT;
127   } else {
128     instCode.inst._wordByte = CLEAR_BIT;
129   }
130
131   instCode.inst._opCode = opc;
132
133   opcode = instCode.sh;
134 }
135
136 /* generate opcode for Register Exchange (REG, REG) instruction SWAP -
      note: MOV constants as well, so, it is handled using 2-op inst (Arith)
      */
137 void handleRegExchange(short int inst_id, vector<string> &ops, unsigned
      short &opcode) {
138   // find REG index from sym_tab
139
140   auto src_index = is_register(ops[0]);
141   auto dst_index = is_register(ops[1]);
142
143   auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
144   RegExchangeOverlay instCode;
145   // populate
146   instCode.inst._dst = sym_tab[dst_index].value;
147   instCode.inst._src = sym_tab[src_index].value;
148   instCode.inst._wordByte = CLEAR_BIT; // this bit and the next are
        ignored but set to 0 for completeness
149   instCode.inst._regCon = CLEAR_BIT;
150   instCode.inst._opCode = opc;
151
152   opcode = instCode.sh;
153 }
154
155 /* generate opcode for OneAddr (REG) instruction  SRA, RRC SWPB SXT */
156 void handleOneAddr(short int inst_id, vector<string> &ops, unsigned short
      &opcode) {
157   // find label index from sym_tab
158   auto dst_index = is_register(ops[0]);
159   bool is_byte = false;
160   // check last 2 chars to check whether ".B" is present in instruction
161   string last2chars = inst_set[inst_id].mnemonic.substr(inst_set[inst_id].
        mnemonic.size() - 2);
162   if (last2chars.compare(".B") == 0) {
163     // instruction ends with .B
164     is_byte = true;
165   }
166   auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
167   OneAddrOverlay instCode;
```

```
168   // populate
169   instCode.inst._dst = sym_tab[dst_index].value;
170   instCode.inst._zeros = CLEAR_BIT;
171   if (is_byte) {
172     instCode.inst._wordByte = SET_BIT;
173   } else {
174     instCode.inst._wordByte = CLEAR_BIT;
175   }
176   instCode.inst._zero = CLEAR_BIT;
177   instCode.inst._opCode = opc;
178
179   opcode = instCode.sh;
180 }
181
182 /* generate opcode for Direct Memory Loading  (LD) instruction */
183 void handleMemAccessLD(short int inst_id, vector<string> &ops, unsigned
       short &opcode) {
184   PrePostIncrDecr addr_modifier = None;
185   if (ops[0].front() == '+') { // pre-increment
186     addr_modifier = PreIncrement;
187   } else if (ops[0].front() == '-') {
188     addr_modifier = PreDecrement;
189   } else if (ops[0].back() == '+') {
190     addr_modifier = PostIncrement;
191   } else if (ops[0].back() == '-') {
192     addr_modifier = PostDecrement;
193   }
194
195   auto src_index = is_register(ops[0]);
196   auto dst_index = is_register(ops[1]);
197   bool is_byte = false;
198   // check last 2 chars to check whether ".B" is present in instruction
199   string last2chars = inst_set[inst_id].mnemonic.substr(inst_set[inst_id].
         mnemonic.size() - 2);
200   if (last2chars.compare(".B") == 0) {
201     // instruction ends with .B
202     is_byte = true;
203   }
204   auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
205   MemAccessOverlay instCode;
206   // populate
207   instCode.inst._dst = sym_tab[dst_index].value;
208   instCode.inst._src = sym_tab[src_index].value;
209   switch (addr_modifier) {
210   case PreIncrement:
211     instCode.inst._prpo = SET_BIT;
212     instCode.inst._dec = CLEAR_BIT;
213     instCode.inst._inc = SET_BIT;
214     break;
215   case PreDecrement:
216     instCode.inst._prpo = SET_BIT;
217     instCode.inst._dec = SET_BIT;
218     instCode.inst._inc = CLEAR_BIT;
219     break;
```

```
220   case PostIncrement:
221     instCode.inst._prpo = CLEAR_BIT;
222     instCode.inst._dec = CLEAR_BIT;
223     instCode.inst._inc = SET_BIT;
224     break;
225   case PostDecrement:
226     instCode.inst._prpo = CLEAR_BIT;
227     instCode.inst._dec = SET_BIT;
228     instCode.inst._inc = CLEAR_BIT;
229     break;
230   default: // no pre/post increment/decrement
231     instCode.inst._prpo = CLEAR_BIT;
232     instCode.inst._dec = CLEAR_BIT;
233     instCode.inst._inc = CLEAR_BIT;
234     break;
235   }
236
237   if (is_byte)
238     instCode.inst._wordByte = SET_BIT;
239   else
240     instCode.inst._wordByte = CLEAR_BIT;
241   instCode.inst._opCode = opc;
242   opcode = instCode.sh;
243 }
244
245 /* generate opcode for Direct Memory Store  (ST) instruction */
246 void handleMemAccessST(short int inst_id, vector<string> &ops, unsigned
       short &opcode) {
247   PrePostIncrDecr addr_modifier = None;
248   if (ops[1].front() == '+') { // pre-increment
249     addr_modifier = PreIncrement;
250   } else if (ops[1].front() == '-') {
251     addr_modifier = PreDecrement;
252   } else if (ops[1].back() == '+') {
253     addr_modifier = PostIncrement;
254   } else if (ops[1].back() == '-') {
255     addr_modifier = PostDecrement;
256   }
257
258   auto src_index = is_register(ops[0]);
259   auto dst_index = is_register(ops[1]);
260   bool is_byte = false;
261   // check last 2 chars to check whether ".B" is present in instruction
262   string last2chars = inst_set[inst_id].mnemonic.substr(inst_set[inst_id].
       mnemonic.size() - 2);
263   if (last2chars.compare(".B") == 0) {
264     // instruction ends with .B
265     is_byte = true;
266   }
267   auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
268   MemAccessOverlay instCode;
269   // populate
270   instCode.inst._dst = sym_tab[dst_index].value;
271   instCode.inst._src = sym_tab[src_index].value;
```

```
272   switch (addr_modifier) {
273   case PreIncrement:
274     instCode.inst._prpo = SET_BIT;
275     instCode.inst._dec = CLEAR_BIT;
276     instCode.inst._inc = SET_BIT;
277     break;
278   case PreDecrement:
279     instCode.inst._prpo = SET_BIT;
280     instCode.inst._dec = SET_BIT;
281     instCode.inst._inc = CLEAR_BIT;
282     break;
283   case PostIncrement:
284     instCode.inst._prpo = CLEAR_BIT;
285     instCode.inst._dec = CLEAR_BIT;
286     instCode.inst._inc = SET_BIT;
287     break;
288   case PostDecrement:
289     instCode.inst._prpo = CLEAR_BIT;
290     instCode.inst._dec = SET_BIT;
291     instCode.inst._inc = CLEAR_BIT;
292     break;
293   default: // no pre/post increment/decrement
294     instCode.inst._prpo = CLEAR_BIT;
295     instCode.inst._dec = CLEAR_BIT;
296     instCode.inst._inc = CLEAR_BIT;
297     break;
298   }
299
300   if (is_byte)
301     instCode.inst._wordByte = SET_BIT;
302   else
303     instCode.inst._wordByte = CLEAR_BIT;
304
305   instCode.inst._opCode = opc;
306   opcode = instCode.sh;
307 }
308
309 /* generate opcode for Cex  (CEX) instruction */
310 void handleCex(short int inst_id, vector<string> &ops, unsigned short &
       opcode) {
311   // operand format: COND, TC, FC
312   auto cond_index = is_cond(ops[0]); // index of the COND in `cecs`
313   // find tc value
314   short int tc;
315   if (is_numeric(ops[1])) { // tc is a value (not label)
316     str2int(ops[1], tc);
317   } else { // tc value is suuplied as label - find the label value in
         sym_tab
318     auto lbl_index = is_label_in_sym_tab(ops[1]);
319     tc = sym_tab[lbl_index].value;
320   }
321   // find fc value
322   short int fc;
323   if (is_numeric(ops[2])) { // tc is a value (not label)
```

```
324       str2int(ops[2], fc);
325     } else { // tc value is suuplied as label - find the label value in
            sym_tab
326       auto lbl_index = is_label_in_sym_tab(ops[2]);
327       fc = sym_tab[lbl_index].value;
328     }
329
330     auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
331     CexOverlay instCode;
332     // populate
333     instCode.inst._fc = fc;
334     instCode.inst._tc = tc;
335     instCode.inst._condCec = cec_values[cond_index];
336     instCode.inst._opCode = opc;
337
338     opcode = instCode.sh;
339 }
340
341 /* generate opcode for Register Initialization  (MOVL, MOVLZ, etc)
        instruction */
342 void handleRegInit(short int inst_id, vector<string> &ops, unsigned short
        &opcode) {
343     // find the BYTE value
344     short int b;
345     if (is_numeric(ops[0])) { // tc is a value (not label)
346       str2int(ops[0], b);
347     } else { // tc value is suuplied as label - find the label value in
            sym_tab
348       auto lbl_index = is_label_in_sym_tab(ops[0]);
349       b = sym_tab[lbl_index].value;
350     }
351     // find REG index
352     auto reg_index = is_register(ops[1]);
353     auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
354     RegisterInitOverlay instCode;
355     // populate
356     instCode.inst._Byte = b;
357     instCode.inst._dst = sym_tab[reg_index].value;
358     instCode.inst._opCode = opc;
359
360     opcode = instCode.sh;
361 }
362
363 /* generate opcode for Relative Memory Loading (LDR) instruction */
364 void handleMemAccessRelLD(short int inst_id, vector<string> &ops, unsigned
        short &opcode) {
365     auto src_index = is_register(ops[0]);
366     auto dst_index = is_register(ops[2]);
367     // get offset value
368     short int offset;
369     if (is_numeric(ops[1])) {
370       str2int(ops[1], offset);
371     } else { // should not reach here, but in case
372       auto lbl_index = is_label_in_sym_tab(ops[1]);
```

```
373     offset = sym_tab[lbl_index].value;
374   }
375   bool is_byte = false;
376   // check last 2 chars to check whether ".B" is present in instruction
377   string last2chars = inst_set[inst_id].mnemonic.substr(inst_set[inst_id].
          mnemonic.size() - 2);
378   if (last2chars.compare(".B") == 0) {
379     // instruction ends with .B
380     is_byte = true;
381   }
382   auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
383   MemAccessRelativeOverlay instCode;
384   // populate
385   instCode.inst._dst = sym_tab[dst_index].value;
386   instCode.inst._offset = offset;
387   instCode.inst._src = sym_tab[src_index].value;
388
389   if (is_byte)
390     instCode.inst._wordByte = SET_BIT;
391   else
392     instCode.inst._wordByte = CLEAR_BIT;
393
394   instCode.inst._opCode = opc;
395   opcode = instCode.sh;
396 }
397
398 /* generate opcode for Relative Memory Store (STR) instruction */
399 void handleMemAccessRelST(short int inst_id, vector<string> &ops, unsigned
        short &opcode) {
400   auto src_index = is_register(ops[0]);
401   auto dst_index = is_register(ops[1]);
402   // get offset value
403   short int offset;
404   if (is_numeric(ops[2])) {
405     str2int(ops[2], offset);
406   } else { // should not reach here, but in case
407     auto lbl_index = is_label_in_sym_tab(ops[2]);
408     offset = sym_tab[lbl_index].value;
409   }
410   bool is_byte = false;
411   // check last 2 chars to check whether ".B" is present in instruction
412   string last2chars = inst_set[inst_id].mnemonic.substr(inst_set[inst_id].
          mnemonic.size() - 2);
413   if (last2chars.compare(".B") == 0) {
414     // instruction ends with .B
415     is_byte = true;
416   }
417   auto opc = stoi(inst_set[inst_id].opcode, nullptr, 2);
418   MemAccessRelativeOverlay instCode;
419   //populate
420   instCode.inst._dst = sym_tab[dst_index].value;
421   instCode.inst._offset = offset;
422   instCode.inst._src = sym_tab[src_index].value;
423
```

```
424   if (is_byte) {
425     instCode.inst._wordByte = SET_BIT;
426   } else {
427     instCode.inst._wordByte = CLEAR_BIT;
428   }
429
430   instCode.inst._opCode = opc;
431
432   opcode = instCode.sh;
433 }
434
435 /* validates the operands of an instruction - if not, error message is
       written to the LIS file */
436 void proc_instruction(short int inst_id, vector<string> &toks, string &
       line, short int &n) {
437   vector<string> operands = {};
438   auto ops = toks[1];   //first token is the instruction and 2nd token
       holds operand(s)
439   stringstream ss(ops); //to split into separate operands from the token
440   string tok;
441   while (getline(ss, tok, ',')) { //operands are separated by comma ','
442     operands.push_back(tok);
443   }
444   auto ex_ops = inst_set[inst_id].expected_operands;
445   //assume - error checked in pass1
446   InstType it = getInstType(ex_ops);
447   unsigned short opcode;
448   switch (it) {
449   case Branch13:
450     handleBranch13(inst_id, operands, opcode);
451   case Branch10:
452     handleBranch10(inst_id, operands, opcode);
453     break;
454   case Arith:
455     handleArith(inst_id, operands, opcode);
456     break;
457   case RegExchange:
458     handleRegExchange(inst_id, operands, opcode);
459     break;
460   case OneAddr:
461     handleOneAddr(inst_id, operands, opcode);
462     break;
463   case MemAccessLD:
464     handleMemAccessLD(inst_id, operands, opcode);
465     break;
466   case MemAccessST:
467     handleMemAccessST(inst_id, operands, opcode);
468     break;
469   case Cex:
470     handleCex(inst_id, operands, opcode);
471     break;
472   case RegInit:
473     handleRegInit(inst_id, operands, opcode);
474     break;
```

```
475     case MemAccessRelLD:
476       handleMemAccessRelLD(inst_id, operands, opcode);
477       break;
478     case MemAccessRelST:
479       handleMemAccessRelST(inst_id, operands, opcode);
480       break;
481     default:
482       break;
483     }
484
485     // print to file
486     stringstream ss_loc, ss_opcode;
487     ss_loc << std::uppercase << std::setfill('0') << std::setw(4) << std::
          hex << loc_counter;
488     ss_opcode << std::uppercase << std::setfill('0') << std::setw(4) << std
          ::hex << opcode;
489     string t = ss_loc.str();
490     string hex_loc = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
491     t = ss_opcode.str();
492     string hex_opcode = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
493
494     ofs << n << "\t" << hex_loc << "\t" << hex_opcode << "\t" << line <<
          endl;
495     s1str = s1str + hex_loc + hex_opcode;
496
497     loc_counter += INSTRUCTION_LEN; //each instruction needs 2-bytes
498   }
499
500   /* generates opcode for BSS */
501   void handleBSS(vector<string> &ops, string &line, short int &n) {
502     short int r;
503     if (is_numeric(ops[0])) { // BSS operand is numeric
504       str2int(ops[0], r);
505     } else { // the value is in sym_tab
506       auto lbl_index = is_label_in_sym_tab(ops[0]);
507       r = sym_tab[lbl_index].value;
508     }
509     // print to file
510     stringstream ss_loc, ss_opcode;
511     ss_loc << std::uppercase << std::setfill('0') << std::setw(4) << std::
          hex << loc_counter;
512     ss_opcode << std::uppercase << std::setfill('0') << std::setw(4) << std
          ::hex << BSS_OPCODE;
513     string t = ss_loc.str();
514     string hex_loc = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
515     t = ss_opcode.str();
516     string hex_opcode = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
517
518     ofs << n << "\t" << hex_loc << "\t" << hex_opcode << "\t" << line <<
          endl;
519     s1str = s1str + hex_loc + hex_opcode;
520     loc_counter += r;
521   }
522
```

```
523 /* generates opcode for BYTE directive */
524 void handleBYTE(vector<string> &ops, string &line, short int &n) {
525   short int r;
526   if (is_numeric(ops[0])) {
527     str2int(ops[0], r);
528   } else { // operand is not a number - consider a label
529     auto r2 = is_label_in_sym_tab(ops[0]);
530     r = sym_tab[r2].value;
531   }
532
533   //print to file
534   stringstream ss_loc, ss_opcode;
535   ss_loc << std::uppercase << std::setfill('0') << std::setw(4) << std::
        hex << loc_counter;
536   ss_opcode << std::uppercase << std::setfill('0') << std::setw(4) << std
        ::hex << r;
537   string t = ss_loc.str();
538   string hex_loc = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
539   t = ss_opcode.str();
540   string hex_opcode = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
541
542   ofs << n << "\t" << hex_loc << "\t" << hex_opcode << "\t" << line <<
        endl;
543   s1str = s1str + hex_loc + hex_opcode;
544   loc_counter += BYTE_INCREASE;
545 }
546
547 /* generates opcode for BYTE directive */
548 void handleWORD(vector<string> &ops, string &line, short int &n) {
549   short int r;
550   if (is_numeric(ops[0])) {
551     str2int(ops[0], r);
552   } else { // operand is not a number - consider a label
553     auto r2 = is_label_in_sym_tab(ops[0]);
554     r = sym_tab[r2].value;
555   }
556
557   // print to file
558   stringstream ss_loc, ss_opcode;
559   ss_loc << std::uppercase << std::setfill('0') << std::setw(4) << std::
        hex << loc_counter;
560   ss_opcode << std::uppercase << std::setfill('0') << std::setw(4) << std
        ::hex << r;
561   string t = ss_loc.str();
562   string hex_loc = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
563   t = ss_opcode.str();
564   string hex_opcode = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
565
566   ofs << n << "\t" << hex_loc << "\t" << hex_opcode << "\t" << line <<
        endl;
567   s1str = s1str + hex_loc + hex_opcode;
568   loc_counter += WORD_INCREASE;
569 }
570 /* process the directive based on the index in directives */
```

```
571 void proc_directive(short int d_id, vector<string> &rec, string &line,
        short int &n, string p_tok) {
572   vector<string> operands = {};
573   if (rec.size() == 2) { // has operand
574     auto ops = rec[1];
575     /* split different operands - separated by comma */
576     stringstream ss(ops); //
577     string tok;
578     while (getline(ss, tok, ',')) {
579       operands.push_back(tok);
580     }
581   }
582   directiveIndexes di = static_cast<directiveIndexes>(d_id);
583   short int r;
584   switch (di) {
585   case dirALIGN:
586     if (loc_counter % 2 != 0) {
587       loc_counter++;
588     } /* if odd increment the address do nothing, print to ofs */
589     ofs << n << "\t\t\t" << line << endl;
590     break;
591   case dirBSS:
592     handleBSS(operands, line, n);
593     break;
594   case dirBYTE: //2: //BYTE
595     handleBYTE(operands, line, n);
596     break;
597   case dirEND:
598     // extract the start address, if provided
599     if (operands.size() == 1) {
600       short int r;
601       if (is_numeric(operands[0])) {
602         str2int(operands[0], r);
603       } else { // operand is not a number - consider a label
604         auto r2 = is_label_in_sym_tab(operands[0]);
605         r = sym_tab[r2].value;
606       }
607       // print to file
608       stringstream ss_opcode;
609       ss_opcode << std::uppercase << std::setfill('0') << std::setw(4) <<
            std::hex << r;
610       string t = ss_opcode.str();
611       string hex_opcode = (t.length() > 4) ? t.substr(t.length() - 4, 4) :
            t;
612       s9str = s9str + hex_opcode;
613     }
614     ofs << n << "\t\t\t" << line << endl;
615     break;
616   case dirEQU:
617     /* do nothing - everything is taken care of at Pass 1 */
618     ofs << n << "\t\t\t" << line << endl;
619     break;
620   case dirORG:
621     // no opcode - just loc_counter  is assigned new value
```

```
622       ofs << n << "\t\t\t" << line << endl;
623       str2int(operands[0], r);
624       loc_counter = r;
625       break;
626     case dirWORD: // 6: //WORD
627       handleWORD(operands, line, n);
628       break;
629     default:
630       break;
631     }
632     return;
633   }
634
635   /* function to validate tokens in a record for pass 1 */
636   void proc_tokens(vector<string> &toks, string &line, short int &n) {
637     //consider 1st token as instruction
638     short int id = check_if_instruction(toks[0]);
639
640     if (id != INVALID_INDEX) { // an instruction - next token must present
          and operand(s)
641       proc_instruction(id, toks, line, n);
642     } else { // either directive or label
643       id = check_if_directive(toks[0]);
644       if (id != INVALID_INDEX) { // a directive found - there may or may not
            have operand(s)
645         proc_directive(id, toks, line, n, "");
646       } else {                  // this token is a label - following token
           must be INST/DIR, if any
647         if (toks.size() > 1) { // more tokens are there and has to be INST/
              DIR
648           string prev_tok = toks[0];
649           toks.erase(toks.begin()); // erase first token
650           id = check_if_instruction(toks[0]);
651           if (id != INVALID_INDEX) { // an instruction - next token must
                present and operand(s)
652             proc_instruction(id, toks, line, n);
653           } else { // either directive or label
654             id = check_if_directive(toks[0]);
655             if (id != INVALID_INDEX) { // a directive found - there may or
                  may not have operand(s)
656               proc_directive(id, toks, line, n, prev_tok);
657             }
658           }
659         }
660       }
661     }
662   }
663
664   /* populate checksum for a s-rec (assuming, count, addr and data is
        present) */
665   void populateChecksum(SRec &srec) {
666     int result = 0;
667     short int r;
668     // process len field (1 byte) of srec
```

```
669   r = stoi(srec._count, nullptr, 16);
670   result += r;
671   // precess addr field (2 bytes) of srec
672   for (unsigned short int i = 0; i < srec._addr.size(); i += 2) {
673     string v = srec._addr.substr(i, 2);
674     r = stoi(v, nullptr, 16);
675     result += r;
676   }
677   // things are different for S0 and S1 then
678   if (srec._type == S0) {
679     for (unsigned short int i = 0; i < srec._data.size(); i++) { // each
          char in string
680       r = srec._data[i];
681       result += r;
682     }
683   } else if (srec._type == S1) {
684     for (unsigned short int i = 0; i < srec._data.size(); i += 2) {
685       string v = srec._data.substr(i, 2);
686       r = stoi(v, nullptr, 16);
687       result += r;
688     }
689   }
690   // get ones complement of the sum
691   stringstream ss;
692   ss << std::uppercase << std::setw(2) << std::hex << ~result; // ones
       complement
693   string t = ss.str();
694   srec._checksum = (t.length() > 2) ? t.substr(t.length() - 2, 2) : t;
695 }
696
697 void populateS0(string f, SRec &srec) {
698   srec._type = S0;
699   srec._addr = "0000";
700   unsigned short int len = f.size();
701   string data0 = f;
702   if (len > 28) { // file name is more than 28 bytes
703     len = 28;
704     data0 = f.substr(0, 28);
705   }
706   srec._data = data0;
707
708   len += ADDR_BYTES_SREC + CHKSUM_BYTE_SREC;
709   // get the hex string for calculated len
710   stringstream ss;
711   ss << std::uppercase << std::setfill('0') << std::setw(2) << std::hex <<
        len;
712   string t = ss.str();
713   srec._count = (t.length() > 2) ? t.substr(t.length() - 2, 2) : t;
714 }
715
716 void get_addr_opcode(const vector<string> &tokens, string addr, string
      rev_opcode) {}
717
718 /* generate S1 screcs from the s1str string */
```

```
719 void generateS1Recs(vector<string> &s1recs) {
720   vector<string> all_address = {};
721   vector<string> all_opcodes = {};
722   for (unsigned short int i = 0; i < s1str.size(); i += 8) { // 4 chars
          for address and 4 chars for opcode
723     string a = s1str.substr(i, 4);
724     all_address.push_back(a);
725     string o = s1str.substr(i + 4, 4);
726     all_opcodes.push_back(o);
727   }
728
729   unsigned short int index = 0;
730   SRec crec = {};
731   crec._type = S1;
732   crec._addr = all_address[index];
733   string data = "";
734   int curr_addr, next_addr;
735
736   while (index < all_address.size()) {
737     curr_addr = stoi(all_address[index], nullptr, 16);
738     if (index < all_address.size() - 1) {
739       next_addr = stoi(all_address[index + 1], nullptr, 16);
740     } else
741       next_addr = INVALID_INDEX;
742
743     data = data + all_opcodes[index].substr(2, 2) + all_opcodes[index].
            substr(0, 2);
744
745     if (abs(next_addr - curr_addr) > 2 || next_addr == INVALID_NUMBER) {
          // need to populate s1 rec and if req, create new s1
746       crec._data = data;
747       unsigned short int len = data.size() / 2 + ADDR_BYTES_SREC +
              CHKSUM_BYTE_SREC;
748       stringstream ss;
749       ss << std::uppercase << std::setfill('0') << std::setw(2) << std::
              hex << len;
750       string t = ss.str();
751       crec._count = (t.length() > 2) ? t.substr(t.length() - 2, 2) : t;
752       populateChecksum(crec);
753       // insert to vector
754       string s1RecStr = "S1" + crec._count + crec._addr + crec._data +
              crec._checksum;
755       s1recs.push_back(s1RecStr);
756       if (next_addr == INVALID_INDEX) { // no more S1 rec to produce
757         break;
758       } else {     // we processed the current record
759         crec = {}; // init
760         crec._type = S1;
761         index++;
762         crec._addr = all_address[index];
763         data = ""; // clear prev conten
764       }
765     } else {
766       index += 1;
```

```
767       }
768     }
769   }
770
771   void print_to_xme(string src_fname) {
772     string ofxme_name = src_fname.substr(0, src_fname.find_last_of('.')) + "
          .xme"; // xme file name
773     of_xme.open(ofxme_name);
774     // Write the S0 record
775     SRec srec0 = {};               // create s0 rec
776     populateS0(src_fname, srec0); // populate byte count,addr, and data
777     populateChecksum(srec0);      // calculate checksum based on count, addr
          and data
778     of_xme << "S0" << srec0._count << srec0._addr << srec0._data << srec0.
          _checksum << endl;
779     // process s1str to create s1 srec(s)
780     // of_xme<<"S1"<<s1str<<endl;
781     vector<string> s1recs;
782     generateS1Recs(s1recs);
783
784     for (unsigned short int i = 0; i < s1recs.size(); ++i)
785       of_xme << s1recs[i] << endl;
786     // generate S9 srecand print
787     SRec srec9 = {}; // create an s9 rec
788     srec9._type = S9;
789     if (s9str.empty()) {
790       srec9._addr = "0000";
791     } else {
792       srec9._addr = s9str;
793     }
794     // populate len/count
795     unsigned short int len = ADDR_BYTES_SREC + CHKSUM_BYTE_SREC; // data
          field is ignored in S9 srec
796     stringstream ss;
797     ss << std::uppercase << std::setfill('0') << std::setw(2) << std::hex <<
          len;
798     string t = ss.str();
799     srec9._count = (t.length() > 2) ? t.substr(t.length() - 2, 2) : t;
800     populateChecksum(srec9);
801     of_xme << "S9" << srec9._count << srec9._addr << srec9._checksum << endl
          ;
802
803     of_xme.close();
804   }
805
806   /* conducts pass1 */
807   bool Pass2(string src_fname) {
808     s1str = "";
809     s9str = "";
810     ofs.close();
811     string of_name = src_fname.substr(0, src_fname.find_last_of('.')) + ".
          lis"; // lis file name
812     ofs.open(of_name);
813
```

```
814   // read the src file line by line and process it
815   ifstream ifs;
816   ifs.open(src_fname);
817   string line;          // to hold the content of a line
818   short int n_line = 0; // corresponding line number in the src file
819   if (ifs.is_open()) {
820     while (getline(ifs, line)) {
821       n_line++;
822       if (line.empty()) {
823         ofs << "\t" << n_line << "\t" << line << endl;
824         continue;
825       } else {
826         // get tokens from the line
827         vector<string> tokens = {};
828         get_tokens(line, tokens);
829         if (tokens.size() > 0) {
830           proc_tokens(tokens, line, n_line);
831         } else {
832           ofs << "\t" << n_line << "\t" << line << endl;
833           continue;
834         }
835       }
836     }
837     ifs.close();
838
839     ofs << "\nSuccessful Completion of Assembly" << endl;
840     ofs << "\n ****  Symbol Table ***" << std::endl;
841     ofs << "Name\t\t\tType\tValue\tDecimal" << std::endl;
842
843     for (auto s : sym_tab) {
844       stringstream ss;
845       ss << std::uppercase << std::setfill('0') << std::setw(4) << std::
             hex << s.value;
846       string t = ss.str();
847       string hex_v = (t.length() > 4) ? t.substr(t.length() - 4, 4) : t;
848       ofs.width(20);
849       ofs << std::left << s.name;
850       ofs.width(8);
851       ofs << std::left << s.type;
852       ofs.width(8);
853       ofs << std::left << hex_v;
854       ofs << s.value << endl;
855     }
856
857     /* char* cwd = _getcwd(0,0); */
858     char *cwd = getcwd(0, 0);
859     string working_directory(cwd);
860     string fname = working_directory + "\\" + src_fname.substr(0,
           src_fname.find_last_of('.')) + ".xme"; // lis file name
861
862     ofs << "\n .XME file:  " << fname << endl;
863     ofs.close();
864     /* print to xme file */
865     //open of_xme to write xmmakina executable
```

```
866
867       print_to_xme(src_fname); // print to xme file
868       return true;
869     } else {
870       std::cout << "Could not open source file: " << src_fname << endl;
871       return false;
872     }
873   }
```

2 Test 1

This test file was provide for the people who have not done the first pass. This contains no errors. Below is a copy of the original asm, below that is a copy of the lis and xme that is expected.

Original 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
```

Expected Output

```
1  1 org $80
2  2 0080 0000 Label01 word #0
3  3 org $FF00
4  4 FF00 0001 Label02 word #1
5  5 org $100
6  6 0100 6C00 Label03 MOVLZ Label01,R0
7  7 0102 7001 MOVLS Label02,R1
8  8 0104 5000 LD R0,R0
9  9 0106 500A LD R1,R2
10 10 0108 4002 Label04 ADD R0,R2
11 11 010A 45AA CMP #16,R2
12 12 010C 27FD BNE Label04
13 13 010E 5411 ST R2,R1
14 14 0110 3FFF Done BRA Done
15 15 END Label03
16
17 Successful completion of assembly
18 ** Symbol table **
19 Name    Type    Value Decimal
20 Done    LBL     0110 272
21 Label04 LBL     0108 264
22 Label03 LBL     0100 256
23 Label02 LBL     FF00 -256
24 Label01 LBL     0080 128
25 R7      REG     0007 7
26 R6      REG     0006 6
27 R5      REG     0005 5
28 R4      REG     0004 4
29 R3      REG     0003 3
30 R2      REG     0002 2
31 R1      REG     0001 1
32 R0      REG     0000 0
```

Expected XME

```
S00D0000A2ex01.txtB3
S105008000007A
S105FF000100FA
S1150100006C017000500A500240AA45FD271154FF3F6A
S9030100FB
```

2.1 Results

After passing the assembly file through my assembler, these were the results:

LIS

```
1     org $80
2 0080  0000  Label01 word #0
3     org $FF00
4 FF00  0001  Label02 word #1
5     org $100
6 0100  6C00  Label03 MOVLZ Label01,R0
7 0102  7001  MOVLS Label02,R1
8 0104  5000  LD R0,R0
9 0106  500A  LD R1,R2
10  0108  4002  Label04 ADD R0,R2
11  010A  45AA  CMP #16,R2
12  010C  27FD  BNE Label04
13  010E  5411  ST R2,R1
14  0110  3FFF  Done BRA Done
15      END Label03
16
17 Successful Completion of Assembly
18
19  ****  Symbol Table ***
20 Name       Type  Value Decimal
21 Done                LBL     0110    272
22 Label04             LBL     0108    264
23 Label03             LBL     0100    256
24 Label02             LBL     FF00    -256
25 Label01             LBL     0080    128
26 R7                  REG     0007    7
27 R6                  REG     0006    6
28 R5                  REG     0005    5
29 R4                  REG     0004    4
30 R3                  REG     0003    3
31 R2                  REG     0002    2
32 R1                  REG     0001    1
33 R0                  REG     0000    0
34
35  .XME file:  /home/z/Documents/AaDS/bin/bin\a3.xme
```

XME

```
S0090000a3.asmF3
S105008000007A
S105FF000100FA
S1150100006C017000500A500240AA45FD271154FF3F6A
S9030100FB
```

The results match.

3 Test 2

A simple test I have written to quickly test my progress.

Original 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
```

3.1 Results

LIS

```
1      SIZE equ $26
2      CAP_A equ 'A'
3    3
4      org #80
5  5 0050  0000  BASE bss SIZE
6  6      org $1000
7    7
8  8 1000  6A08  Start movlz CAP_A,R0
9    9
10
11 Successful Completion of Assembly
12
13  ****  Symbol Table ***
14 Name       Type  Value Decimal
15 Start                LBL     1000    4096
16 BASE                 LBL     0050    80
17 CAP_A                LBL     0041    65
18 SIZE                 LBL     0026    38
19 R7                   REG     0007    7
20 R6                   REG     0006    6
21 R5                   REG     0005    5
22 R4                   REG     0004    4
23 R3                   REG     0003    3
24 R2                   REG     0002    2
25 R1                   REG     0001    1
26 R0                   REG     0000    0
27
28  .XME file:  /home/z/Documents/AaDS/bin/bin\example1.xme
```

XME

```
1  S00F0000example1.asm64
2  S10500500000AA
3  S1051000086A78
4  S9030000FC
```

We can assume this executable is correct as Test 1 was successful and the results matched.