```
 1: /*
 2: __/\\\\\\\\\\\\\\_____/\\\\\\\\\\\__/\\\\\\\\\\\\\____
 3:  _\/\\\/////////\\\__\/////\\\///__\/\\\/////////\\\__
 4:   _\/\\\_______\/\\\______\/\\\_____\/\\\_______\/\\\_
 5:    _\/\\\\\\\\\\\\\/_______\/\\\_____\/\\\_______\/\\\_
 6:     _\/\\\/////////_________\/\\\_____\/\\\_______\/\\\_
 7:      _\/\\\__________________\/\\\_____\/\\\_______\/\\\_
 8:       _\/\\\_____________/\\\___\/\\\_____\/\\\_______/\\\__
 9:        _\/\\\____________\//\\\\\\\\\_____\/\\\\\\\\\\\\\/___
10:         _\///______________\/////////______\/////////////_____
11:
12: -> Name:  main.cpp
13: -> Brief: Implementation for the main.cpp the code that runs executive commands
14: -> Date: May 15, 2017   (Created)
15: -> Author: Paul Duchesne (B00332119)
16: -> Contact: pl332718@dal.ca
17: */
18:
19: #include <iostream>
20: #include <stdio.h>
21: #include <string>
22: #include <cstring>
23: #include <fstream>
24: #include <cstdlib>
25:
26: #include "Include/main.h"
27: #include "Include/library.h"
28: #include "Include/symtbl.h"
29: #include "Include/inst_dir.h"
30: #include "Include/parser.h"
31: #include "Include/first_pass.h"
32: #include "Include/second_pass.h"
33: #include "Include/emitter.h"
34:
35: // Globals
36: std::string current_record = "";
37: std::string current_token  = "";
38: int err_cnt = 0;
39:
40: std::ifstream fin;
41: std::ofstream outfile;
42: std::ofstream srec_file;
43:
44: int main(int argc, char *argv[])
45: {
46:         // "Drag and drop" capability, used in command line personnaly
47:         if(argc < 2)
48:         {
49:                 std::cout << "ERROR: Missing input file" << std::endl;
50:                 getchar();
51:                 exit(0);
52:         }
53:
54:         fin.open(argv[1]);
55:
56:         if(!fin.is_open())
57:         {
58:                 std::cout << "ERROR READING FILE" << std::endl;
59:                 getchar();
60:                 exit(0);
61:         }
62:
63:         init_symtbl();
64:
65:         // For diagnostics
66:         outfile.open("diagnostics.LIS");
67:
68:         outfile << "FIRST PASS DIAGNOSTICS (Emitted Records are in ERROR)" << std::endl << std::endl;
69:
70:         // Runs the first pass
71:         first_pass();
72:
73:         std::cout << std::endl << "\tFirst Pass Completed with >>" << err_cnt
74:                   << "<< Errors (Not including unknowns)" << std::endl;
75:         outfile << std::endl << "\tFirst Pass Completed with >>" << err_cnt
76:                     << "<< Errors (Not including unknowns)" << std::endl;
77:
78:         // Check the symbol table for unresolved unknowns
79:         symtbl_unknown_check();
80:
81:         std::cout << std::endl << "\tFirst Pass Completed with >>" << err_cnt
82:                           << "<< Errors (Including unknowns)" << std::endl;
83:         outfile << std::endl << "\tFirst Pass Completed with >>" << err_cnt
```

```
   84:                     << "<< Errors (Including unknowns)" << std::endl;
   85:
   86:         // If there are no errors, rewind file and run second pass
   87:         if(err_cnt == 0)
   88:         {
   89:                 // Rewind file to beginning
   90:                 fin.clear();
   91:                 fin.seekg(0);
   92:
   93:                 // Run second pass
   94:                 outfile << std::endl << "SECOND PASS DIAGNOSTICS (All Records emitted with format shown bel
ow)";
   95:                 outfile << std::endl << std::endl;
   96:                 second_pass();
   97:         }
   98:
   99:         std::cout << std::endl;
  100:         outfile << std::endl;
  101:
  102:         output_symtbl();
  103:
  104:         outfile << std::endl << "END OF DIAGNOSTICS OUTFILE" << std::endl;
  105:
  106:         fin.close();
  107:         outfile.close(); // Note: srec_file is closed in the s9 function
  108:
  109:         return 0;
  110: }
```

```
    1: /*
    2: __/\\\\\\\\\\\\\\\_____/\\\\\\\\\\\__/\\\\\\\\\\\\\\\____
    3:  _\/\\\/////////\\\__\/////\\\///__\/\\\/////////\\\__
    4:   _\/\\\_______\/\\\______\/\\\_____\/\\\_______\/\\\_
    5:    _\/\\\\\\\\\\\\\/_______\/\\\_____\/\\\_______\/\\\_
    6:     _\/\\\/////////_________\/\\\_____\/\\\_______\/\\\_
    7:      _\/\\\__________________\/\\\_____\/\\\_______\/\\\_
    8:       _\/\\\_____________/\\\___\/\\\_____\/\\\_______/\\\__
    9:        _\/\\\____________\//\\\\\\\\\_____\/\\\\\\\\\\\\\/___
   10:         _\///______________\/////////______\/////////////_____
   11:
   12: -> Name:  s19_maker.cpp
   13: -> Brief: Function file for s19_maker.cpp
   14: -> Date: May 26, 2017   (Created)
   15: -> Author: Paul Duchesne (B00332119)
   16: -> Contact: pl332718@dal.ca
   17: -> Note: This section in particular is quite similar to
   18:       the example code provided by Dr Hughes
   19: */
   20:
   21: #include <iostream>
   22: #include <stdio.h>
   23: #include <string>
   24: #include <cstring>
   25: #include <fstream>
   26: #include <cstdlib>
   27: #include <iomanip>
   28:
   29: #include "Include/s19_maker.h"
   30:
   31: #define SREC_MAX_DATA_SIZE 32 // 32 bytes of data, 64 hex characters
   32:
   33: extern std::ofstream srec_file;
   34:
   35: unsigned short srec_buffer[SREC_MAX_DATA_SIZE];
   36: unsigned short srec_chksum;
   37: unsigned int srec_address;
   38: int srec_index;
   39: int test_cnt = 0;
   40:
   41: /*
   42:     Function: init_srec
   43:     Input: Address: The value to initialize the new S1 record with
   44:     Brief: This function is called to start a new S1 record, immediately
   45:                 after the srec_buffer is emitted. It resets all global
   46:                 variables and adds the address to the checksum.
   47: */
   48: void init_srec(unsigned int address)
   49: {
   50:         srec_index = 0;
   51:         srec_chksum = 0;
   52:         srec_address = address;
   53:         // Add the checksum first with the LSB and then with the MSB (Order doesn't matter)
   54:         srec_chksum += (srec_address >> 8) & 0xff;
   55:         srec_chksum += srec_address & 0xff;
   56: }
   57:
   58: /*
   59:     Function: output_srec_buffer
   60:     Brief: This function is called either by the write_srec_byte() function,
   61:                 or by BYTE, WORD, or STRING in the second pass. After being
   62:                 called, the contents of the srec_buffer is outputted as a
   63:                 complete S1 record with the correct count and address. After
   64:                 the data is outputted, the checksum is emitted to finish the
   65:                 record as a ones compliment.
   66: */
   67: void output_srec_buffer()
   68: {
   69:         if(srec_index != 0) // If the buffer is empty, don't print an empty S1. That would be silly.
   70:         {
   71:                 unsigned short count = 0;
   72:                 count = srec_index + 3; // Plus 3 for the CNT (1) and ADDRESS (2)
   73:
   74:                 // S1 header, Count, and Address
   75:                 srec_file << "S1" << std::right << std::setfill('0') << std::setw(2) << std::hex << count
   76:                         << std::right << std::setfill('0') << std::setw(4) << std::hex << srec_address;
   77:
   78:                 // DATA
   79:                 for(int i = 0; i < srec_index; i++)
   80:                 {
   81:                         srec_file << std::right << std::setfill('0') << std::setw(2) << std::hex << srec_bu
ffer[i];
   82:                 }
```

```
   83:                 // CHECKSUM
   84:                 srec_chksum += count;
   85:
   86:                 srec_chksum = (~srec_chksum) & 0xff;
   87:
   88:                 srec_file << std::right << std::setfill('0') << std::setw(2) << std::hex << srec_chksum <<
std::endl;
   89:
   90:                 // This may be overwritten if the new Srec is initialized by a directive that moves the LC
   91:                 srec_address += srec_index;
   92:
   93:                 init_srec(srec_address); // This may be overwritten if another emit() is called
   94:                                                                     // before the first byte is added to the b
uffer
   95:         }
   96: }
   97:
   98: /*
   99:     Function: write_srec_byte
  100:     Input: byte: The byte to add to the srec_buffer
  101:     Brief: Takes an input byte and adds it to the srec_buffer. If the
  102:         buffer is full, the buffer is first emitted before the byte
  103:         is added to the first place in the srec_buffer.
  104: */
  105: void write_srec_byte(unsigned char byte)
  106: {
  107:         if(srec_index == SREC_MAX_DATA_SIZE)
  108:         {
  109:                 output_srec_buffer();
  110:         }
  111:         else if(srec_index > SREC_MAX_DATA_SIZE)
  112:         {
  113:                 std::cout << "THIS SHOULD NEVER HAPPEN (Write SREC BYTE, BUFFER OVERFILLED)" << std::endl;
  114:                 getchar();
  115:         }
  116:
  117:         srec_buffer[srec_index++] = byte & 0xff;
  118:         srec_chksum += byte;
  119: }
  120:
  121: /*
  122:     Function: write_srec_word
  123:     Input: byte: The word to add to the srec_buffer
  124:     Brief: Takes an input word and calls write_srec_byte twice. First
  125:         the least significant byte is sent in, and then the most
  126:         significant byte is sent in. This is due to MSB 430 being a
  127:         Little-Endian system.
  128: */
  129: void write_srec_word(unsigned short word)
  130: {
  131:         write_srec_byte((unsigned char)(word&0xff));                    // Send in LSB first
  132:         write_srec_byte((unsigned char)((word >> 8)&0xff));            // Send in MSB second
  133: }
  134:
  135: /*
  136:     Function: write_S9
  137:     Brief: This function is called at the end of the file, or by
  138:         the end directive and it serves to add the closing S9 record
  139:         to the srec_file. The srec_file is also closed.
  140: */
  141: void write_S9(unsigned int s9_srec_address)
  142: {
  143:         // S9 and 03, 03 is the CNT, which is always 3 for the S9 record
  144:
  145:         // Emit the previous buffer
  146:         output_srec_buffer();
  147:
  148:         // Calculate S9 record checksum
  149:         srec_chksum = 0;
  150:         srec_chksum += (s9_srec_address >> 8) & 0xff;
  151:         srec_chksum += s9_srec_address & 0xff;
  152:         srec_chksum += 0x03; // CNT is always 3 for S9 records
  153:         srec_chksum = (~srec_chksum) & 0xff;
  154:
  155:         // Emit the final S9 record and close file
  156:         srec_file << "S903" << std::setfill('0') << std::setw(4) << std::hex << s9_srec_address << std::set
w(2) << srec_chksum;
  157:         srec_file.close();
  158: }
```

```
    1: /*
    2: __/\\\\\\\\\\\\\\_____/\\\\\\\\\\\__/\\\\\\\\\\\\\____
    3:  _\/\\\/////////\\\__\/////\\\///__\/\\\/////////\\\__
    4:   _\/\\\_______\/\\\______\/\\\_____\/\\\_______\/\\\_
    5:    _\/\\\\\\\\\\\\\/_______\/\\\_____\/\\\_______\/\\\_
    6:     _\/\\\/////////_________\/\\\_____\/\\\_______\/\\\_
    7:      _\/\\\__________________\/\\\_____\/\\\_______\/\\\_
    8:       _\/\\\_____________/\\\___\/\\\_____\/\\\_______/\\\__
    9:        _\/\\\____________\//\\\\\\\\\_____\/\\\\\\\\\\\\\/___
   10:         _\///______________\/////////______\/////////////_____
   11:
   12: -> Name:  emitter.cpp
   13: -> Brief: Function file for emitter.cpp
   14: -> Date: May 24, 2017   (Created)
   15: -> Author: Paul Duchesne (B00332119)
   16: -> Contact: pl332718@dal.ca
   17: */
   18:
   19: #include <iostream>
   20: #include <stdio.h>
   21: #include <string>
   22: #include <cstring>
   23: #include <fstream>
   24: #include <cstdlib>
   25: #include <iomanip>
   26:
   27: #include "Include/library.h"
   28: #include "Include/symtbl.h"
   29: #include "Include/inst_dir.h"
   30: #include "Include/parser.h"
   31: #include "Include/emitter.h"
   32: #include "Include/s19_maker.h"
   33:
   34: #define PC  0   // Program  Counter
   35: #define SR  2   // Status   Register
   36: #define CG1 2   // Constant Generator 1, used for -1, 0, 1, 2
   37: #define CG2 3   // Constant Generator 2, Used for 4 and 8
   38:
   39: // As values for setting As and Ad fields (First 4 Ad values equal first 4 Ad values)
   40: // Numbers correspond to enumeration declaration order in library.h
   41: int as_value[] = {0, 1, 1, 1, 2, 3, 3};
   42:
   43: // Addressing mode arrays for increasing location counter
   44: int addr_mode_LC_array_src[] = {0, 2, 2, 2, 0, 0, 2};
   45: int addr_mode_LC_array_dst[] = {0, 2, 2, 2, 0, 0, 0};
   46:
   47: /*
   48:         Function: emit
   49:         Input:  inst: string of instruction to carry out
   50:                         operand: string of operand(s)
   51:                         type: type of instruction
   52:                         outfile: file to print out to
   53:                         LC: Pointer to LC in the second pass for syncronization
   54:         Brief:The first pass performs error checking on the input .asm file
   55:                         and fills the symbol table with all appropriate values. If an error
   56:                         is found, it is recorded and will prevent the second pass from starting.
   57:                         The first pass works by utilizing a state machine that cycles through
   58:                         records individually, one token at a time. Please see the data flow diagram
   59:                         in the Diagrams folder for a general overview of the state transitions.
   60:
   61:         Note: The reason I used "std::string inst" instead of "inst_dir& inst_ptr" was
   62:                         to allow users to input the strings to emit. This is used in the case
   63:                         of the byte and word directives and was used heavily for debugging. This
   64:                         does however waste clock cycles because get_inst_dir() has to be called
   65:                         every time emit is called, but this is somewhat offset by the fact that
   66:                         it uses a binary search to find the instruction.
   67: */
   68: void emit(std::string inst, std::string operand, INST_TYPE type, int& LC)
   69: {
   70:
   71:         inst_dir* id_ptr = get_inst_dir(inst, I);
   72:         symtbl_entry* symtbl_ptr = NULL;
   73:         ADDR_MODE addr_mode0 = WRONG; // Used in general (For One operand, SRC in two operand, and jump ope
rand)
   74:         ADDR_MODE addr_mode1 = WRONG; // Used for DST in double operand
   75:
   76:         std::string src_string = "";
   77:         std::string dst_string = "";
   78:
   79:         single_overlay single;
   80:         double_overlay dbl;
   81:         jump_overlay jump;
   82:
```

```
   83:         // Used for JUMP, ONE, and the SRC of TWO operand instructions
   84:         int value0 = -1; // General value
   85:         int value1 = -1; // Used for register in indexed mode
   86:
   87:         // Used for DST of two operand instructions
   88:         int value0_dbl = -1; // General value
   89:         int value1_dbl = -1; // Used for register in indexed mode
   90:
   91:         // Flag used if the constant generator is used
   92:         bool constant_gen_flag = false;
   93:
   94:         // Set up output settings:
   95:         outfile << std::setfill('0') << std::right;
   96:
   97:         switch (type)
   98:         {
   99:                 case NONE: // Just RETI
  100:                         if(id_ptr->mnemonic == "RETI")
  101:                         {
  102:                                 outfile << "\t\t" << std::hex << std::setw(4) << LC << " " << 0x1300 << std
::endl;
  103:                                 write_srec_word(0x1300);
  104:
  105:                                 LC += 2;
  106:                         }
  107:                         else
  108:                         {
  109:                                 std::cout << "THIS SHOULD NEVER HAPPEN (Default case NONE emit)" << std::en
dl;
  110:                                 getchar();
  111:                         }
  112:                         break;
  113:                 case SINGLE:
  114:                         single.opcode = id_ptr->opcode/(128);  // Bit shift the opcode to the right 7 times
 (2^7)
  115:                         single.bw = id_ptr->b_w;
  116:
  117:                         addr_mode0 = parse(operand, value0, value1);
  118:
  119:                         single.as = as_value[addr_mode0];
  120:
  121:                         switch(addr_mode0) // DEAL WITH SOURCE
  122:                         {
  123:                                 case REG_DIRECT: // All 3 of these do the same thing in single operand mode
  124:                                 case INDIRECT:
  125:                                 case INDIRECT_AI:
  126:                                         single.reg = value0;
  127:                                         break;
  128:
  129:                                 case INDEXED:
  130:                                         single.reg = value1;
  131:                                         break;
  132:
  133:                                 case RELATIVE:
  134:                                         single.reg = PC;
  135:                                         value0 -= LC; // LC of the INSTRUCTION, not value
  136:                                         break;
  137:
  138:                                 case ABSOLUTE:
  139:                                         single.reg = SR;
  140:                                         break;
  141:
  142:                                 case IMMEDIATE:
  143:                                         single.reg = PC;
  144:                                         // Constant generator (CG) functionality: Tests if the value is on
the CG list
  145:                                         if(value0 == -1||value0 == 0||value0 == 1||value0 == 2||value0 == 4
||value0 == 8)
  146:                                         {
  147:                                                 operand.erase(0,1);
  148:                                                 symtbl_ptr = get_symbol(operand);
  149:                                                 if(symtbl_ptr != NULL) if(symtbl_ptr->line > line_num) brea
k;
  150:
  151:                                                 single.as = (value0 > 4) ? CG1 : CG2; // CG2 deals with -1,
 0, 1, and 2
  152:
          // CG1 deals with 4 and 8
  153:                                                 constant_gen_flag = true;
  154:                                                 // Then overwrite As for the specific value
  155:                                                 switch (value0)
  156:                                                 {
  157:                                                         case  0:
```

```
  158:                                                                    single.as = 0;
  159:                                                                    break;
  160:                                                            case  1:
  161:                                                                    single.as = 1;
  162:                                                                    break;
  163:                                                            case  2:
  164:                                                            case  4:
  165:                                                                    single.as = 2;
  166:                                                                    break;
  167:                                                            default:  // Note: for "case -1:" and "case 8:", si
ngle.as is
  168:                                                                              // already set to 3 from before t
he switch statement
  169:                                                                    break;
  170:                                                    }
  171:                                            }
  172:                                            break;
  173:
  174:                                    default:
  175:                                            std::cout << "This is an issue (WRONG addr_mode0 found)" << std::en
dl;
  176:                                            getchar();
  177:                                            break;
  178:                            }
  179:
  180:                            outfile << "\t\t" << std::hex << std::setw(4) << LC << " "
  181:                                                            << std::setw(4) << single.us_single << std::endl;
;
  182:                            write_srec_word(single.us_single);
  183:
  184:                            // Inrement LC for the instruction
  185:                            LC += 2;
  186:
  187:                            if(addr_mode_LC_array_src[addr_mode0] && !constant_gen_flag) // Emit SRC output if
needed
  188:                            {
  189:                                    outfile << "\t\t" << std::hex << std::setw(4) << LC << " "
  190:                                                                                            << std::setw(4) <
< (unsigned short)value0 << std::endl;
  191:                                    write_srec_word((unsigned short)value0);
  192:                                    LC += 2; // Because the addr_mode_LC_array_src is used to get into this sta
tement,
  193:                                                    // the LC is always increased if successful
  194:                            }
  195:
  196:                            break;
  197:
  198:                    case DOUBLE: // 2
  199:                            dbl.opcode = id_ptr->opcode/4096; // shift to the right 12 times before inputting (
2^12)
  200:
  201:                            dbl.bw = id_ptr->b_w;
  202:
  203:                            src_string = operand.substr(0, operand.find_first_of(","));
  204:                            dst_string = operand.substr(operand.find_first_of(",")+1);
  205:
  206:                            addr_mode0 = parse(src_string, value0, value1);
  207:                            addr_mode1 = parse(dst_string, value0_dbl, value1_dbl);
  208:
  209:                            // Sets As and Ad fields for structures
  210:                            dbl.as = as_value[addr_mode0];
  211:                            dbl.ad = as_value[addr_mode1];   // As and Ad are identical for first 4 addressing
modes
  212:
  213:                            switch(addr_mode0) // DEAL WITH SOURCE
  214:                            {
  215:                                    case REG_DIRECT:
  216:                                    case INDIRECT:
  217:                                    case INDIRECT_AI:
  218:                                            dbl.src = value0;
  219:                                            break;
  220:
  221:                                    case INDEXED:
  222:                                            dbl.src = value1;
  223:
  224:                                            break;
  225:
  226:                                    case RELATIVE:
  227:                                            dbl.src = PC;
  228:                                            value0 -= LC; // LC of the INSTRUCTION, not value
  229:
  230:                                            break;
  231:
```

```
  232:                                    case ABSOLUTE:
  233:                                            dbl.src = SR;
  234:
  235:                                            break;
  236:
  237:                                    case IMMEDIATE:
  238:                                            dbl.src = PC;
  239:                                            // Constant generator test
  240:                                            if(value0 == -1||value0 == 0||value0 == 1||value0 == 2||value0 == 4
||value0 == 8)
  241:                                            {
  242:                                                    src_string.erase(0,1);
  243:                                                    symtbl_ptr = get_symbol(src_string);
  244:                                                    if(symtbl_ptr != NULL)
  245:                                                    {
  246:                                                            {
  247:                                                                    if(symtbl_ptr->line > line_num) break;
  248:                                                            }
  249:                                                    }
  250:
  251:                                                    dbl.src = (value0 > 4) ? CG1 : CG2; // CG2 deals with -1, 0
, 1, and 2
  252:
        // CG1 deals with 4 and 8
  253:
  254:                                                    constant_gen_flag = true;
  255:                                                    // Then overwrite As for the specific value
  256:                                                    switch (value0)
  257:                                                    {
  258:                                                            case  0:
  259:                                                              dbl.as = 0;
  260:                                                              break;
  261:                                                            case  1:
  262:                                                              dbl.as = 1;
  263:                                                              break;
  264:                                                            case  2:
  265:                                                            case  4:
  266:                                                              dbl.as = 2;
  267:                                                              break;
  268:                                                            default:  // Note: for "case -1:" and "case 8:", db
l.as is already set to 3
  269:                                                              break;
  270:                                                    }
  271:                                            }
  272:
  273:                                            break;
  274:
  275:                                    default:
  276:                                            std::cout << "This should never happen (Double SRC switch case)" <<
 std::endl;
  277:                                            getchar();
  278:                                            break;
  279:
  280:                            }
  281:
  282:                            switch (addr_mode1)  // FOR DST
  283:                            {
  284:                                    case REG_DIRECT:
  285:                                            dbl.dst = value0_dbl;
  286:                                            break;
  287:
  288:                                    case INDEXED:
  289:                                            dbl.dst = value1_dbl;
  290:                                            break;
  291:
  292:                                    case RELATIVE:
  293:                                            dbl.dst = PC;
  294:                                            value0_dbl -= LC; // LC of the INSTRUCTION, not value
  295:                                            break;
  296:
  297:                                    case ABSOLUTE:
  298:                                            dbl.dst = SR;
  299:                                            break;
  300:
  301:                                    default:
  302:                                            std::cout << "This should never happen (Double DST switch case)" <<
 std::endl;
  303:                                            getchar();
  304:                                            break;
  305:                            }
  306:
  307:                            // Emit INST
  308:                            outfile << "\t\t" << std::hex << std::setw(4) << LC << " " << dbl.us_double << std:
```

```
:endl;;
  309:                          write_srec_word(dbl.us_double);
  310:
  311:                          // Increase LC for INST
  312:                          LC += 2;
  313:
  314:                          // Emit SRC output if needed (Not if constant generator is used
  315:                          if(addr_mode_LC_array_src[addr_mode0] && !constant_gen_flag)
  316:                          {
  317:                                  outfile << "\t\t" << std::hex << std::setw(4) << LC << " "
  318:                                                                                          <<  std::setw(4)
<< (unsigned short)value0 << std::endl;
  319:                                  write_srec_word((unsigned short)value0);
  320:                                  LC += 2; // Because the addr_mode_LC_array_src is used to get into this sta
tement,
  321:                                                  // the LC is always increased if successful
  322:                          }
  323:
  324:
  325:                          if(addr_mode_LC_array_dst[addr_mode1]) // Emit DST output if needed
  326:                          {
  327:                                  outfile << "\t\t" << std::hex << std::setw(4) << LC << " "
  328:                                                                          << std::setw(4) << (unsigned short)value0
_dbl << std::endl;
  329:                                  write_srec_word((unsigned short)value0_dbl);
  330:                                  LC += 2; // Because the addr_mode_LC_array_dst is used to get into this sta
tement,
  331:                                                  // the LC is always increased if successful
  332:                          }
  333:
  334:                          break;
  335:
  336:                  case JUMP:      // 3
  337:                          addr_mode0 = parse(operand, value0, value1);
  338:
  339:                          jump.opcode = id_ptr->opcode/1024; // Shift to the right 10 times (2^10)
  340:
  341:                          // Caluclating 10 bit offset for JUMP instruction.
  342:                          value0 -= (unsigned)LC;         // Finds address relative to LC
  343:                          value0 = value0>>1;             // Bitshift to the right once
  344:                          value0 = value0 & 0x03FF;       // Only take the least 10 significant bits
  345:
  346:                          jump.offset = value0;
  347:
  348:                          // EMIT
  349:                          outfile << "\t\t" << std::hex << std::setw(4) << LC << " "
  350:                                                                          << std::setw(4) << (unsigned short)jump.u
s_jump << std::endl;;
  351:                          write_srec_word(jump.us_jump);
  352:
  353:                          // Increase LC for INST
  354:                          LC += 2;
  355:
  356:                          break;
  357:
  358:                  default:
  359:                          std::cout << "THIS SHOULD NEVER HAPPEN" << std::endl;
  360:                          getchar();
  361:                          break;
  362:          }
  363:          std::dec; // Resets output streams to print out decimals, not hex
  364: }
```

```
 1: /*
 2: __/\\\\\\\\\\\\\\_____/\\\\\\\\\\\__/\\\\\\\\\\\\\____
 3:  _\/\\\/////////\\\__\/////\\\///__\/\\\/////////\\\__
 4:   _\/\\\_______\/\\\______\/\\\_____\/\\\_______\/\\\_
 5:    _\/\\\\\\\\\\\\\/_______\/\\\_____\/\\\_______\/\\\_
 6:     _\/\\\/////////_________\/\\\_____\/\\\_______\/\\\_
 7:      _\/\\\__________________\/\\\_____\/\\\_______\/\\\_
 8:       _\/\\\_____________/\\\___\/\\\_____\/\\\_______/\\\__
 9:        _\/\\\____________\//\\\\\\\\\_____\/\\\\\\\\\\\\/___
10:         _\///______________\/////////______\////////////_____
11:
12: -> Name:  symtbl.cpp
13: -> Brief: Implements the symtbl with functions and such
14: -> Date: May 15, 2017   (Created)
15: -> Author: Paul Duchesne (B00332119)
16: -> Contact: pl332718@dal.ca
17: */
18:
19: #include <iostream>
20: #include <stdio.h>
21: #include <string>
22: #include <fstream>
23: #include <cstdlib>
24: #include <iomanip>
25:
26: #include "Include/symtbl.h"
27: #include "Include/inst_dir.h"
28:
29: #define MAX_SYM_LENGTH 31
30:
31: // Types as string, this order corresponds to the enumeration order in library.h
32: std::string types[] = {"REGISTER", "KNOWN", "UNKNOWN"};
33:
34: // Pointer to the start of the symbol table
35: symtbl_entry* symtbl_master = NULL;
36:
37: /*
38:     Function: init_symtbl
39:     Brief: This function takes initializes the symbol table by adding all the
40:                 registers and their aliases to the symbol table.
41: */
42: void init_symtbl()
43: {
44:         // Added r0-r15, R0-R15, plus aliases (and case values)
45:         add_symbol("R0",  0,  REG);
46:         add_symbol("R1",  1,  REG);
47:         add_symbol("R2",  2,  REG);
48:         add_symbol("R3",  3,  REG);
49:         add_symbol("R4",  4,  REG);
50:         add_symbol("R5",  5,  REG);
51:         add_symbol("R6",  6,  REG);
52:         add_symbol("R7",  7,  REG);
53:         add_symbol("R8",  8,  REG);
54:         add_symbol("R9",  9,  REG);
55:         add_symbol("R10", 10, REG);
56:         add_symbol("R11", 11, REG);
57:         add_symbol("R12", 12, REG);
58:         add_symbol("R13", 13, REG);
59:         add_symbol("R14", 14, REG);
60:         add_symbol("R15", 15, REG);
61:         add_symbol("r0",  0,  REG);
62:         add_symbol("r1",  1,  REG);
63:         add_symbol("r2",  2,  REG);
64:         add_symbol("r3",  3,  REG);
65:         add_symbol("r4",  4,  REG);
66:         add_symbol("r5",  5,  REG);
67:         add_symbol("r6",  6,  REG);
68:         add_symbol("r7",  7,  REG);
69:         add_symbol("r8",  8,  REG);
70:         add_symbol("r9",  9,  REG);
71:         add_symbol("r10", 10, REG);
72:         add_symbol("r11", 11, REG);
73:         add_symbol("r12", 12, REG);
74:         add_symbol("r13", 13, REG);
75:         add_symbol("r14", 14, REG);
76:         add_symbol("r15", 15, REG);
77:         add_symbol("PC",  0,  REG);
78:         add_symbol("Pc",  0,  REG);
79:         add_symbol("pC",  0,  REG);
80:         add_symbol("pc",  0,  REG);
81:         add_symbol("SP",  1,  REG);
82:         add_symbol("Sp",  1,  REG);
83:         add_symbol("sP",  1,  REG);
```

```
   84:         add_symbol("sp",  1,  REG);
   85:         add_symbol("SR",  2,  REG);
   86:         add_symbol("Sr",  2,  REG);
   87:         add_symbol("sR",  2,  REG);
   88:         add_symbol("sr",  2,  REG);
   89:         add_symbol("CG1", 2,  REG);
   90:         add_symbol("cG1", 2,  REG);
   91:         add_symbol("Cg1", 2,  REG);
   92:         add_symbol("cg1", 2,  REG);
   93:         add_symbol("CG2", 3,  REG);
   94:         add_symbol("Cg2", 3,  REG);
   95:         add_symbol("cG2", 3,  REG);
   96:         add_symbol("cg2", 3,  REG);
   97: }
   98:
   99: /*
  100:     Function: add_symbol
  101:     Input: label: input strint to add
  102:            value: value associated with the label
  103:            type: symbol table type for the symbol (Unknown, Known, or REG)
  104:     Brief: This function adds a symbol to the symbol table. It does no validity
  105:                 testing, that should be done (perhaps with "Valid_Symbol(X)") before
  106:                 calling this function.
  107: */
  108: void add_symbol(std::string label, int value, SYMTBLTYPE type)
  109: {
  110:         symtbl_entry* new_entry = new symtbl_entry();
  111:         new_entry->label = label;
  112:         new_entry->value = value;
  113:         new_entry->type = type;
  114:         new_entry->next = symtbl_master;
  115:         new_entry->line = line_num;
  116:         symtbl_master = new_entry;
  117: }
  118:
  119: /*
  120:     Function: output_symtbl()
  121:     Brief: This function outputs the symbol table when called by iterating
  122:                 through the table until the last result. Some aestetics were
  123:                 added for ease of readibility during debugging, these involve
  124:                 determining the maximum size for each column and setting the
  125:                 cout width to that value.
  126: */
  127: void output_symtbl()
  128: {
  129:         int temp_cnt = 0;
  130:         int temp_cnt2 = 0;
  131:
  132:     symtbl_entry* temp = symtbl_master;
  133:
  134:         // PURELY AESTECTIC PORTION (Formatting column width of output print so it looks nice)
  135:
  136:     int entry_no_length = 0;
  137:         int line_no_length = 0;
  138:
  139:         // To format, I iterate through the symbol table first, obtaining the number
  140:         // of entries, the maximum line number, and the maximum label length
  141:     int max_label_length = 0;   //
  142:         int max_symbol_line = 0;
  143:
  144:         while(temp->next != NULL)
  145:         {
  146:                 if(temp->label.length() > max_label_length) max_label_length = temp->label.length();
  147:                 if(temp->line > max_symbol_line) max_symbol_line = temp->line;
  148:                 temp = temp->next;
  149:                 temp_cnt++;
  150:         }
  151:
  152:         // Determining width of n in "ENTRY #n" column
  153:         while(temp_cnt >= 1)
  154:         {
  155:                 temp_cnt = temp_cnt/10;
  156:                 entry_no_length++;
  157:         }
  158:
  159:         // Determining width of n in "Line #n" column
  160:         while(max_symbol_line >= 1)
  161:         {
  162:                 max_symbol_line /= 10;
  163:                 line_no_length++;
  164:         }
  165:
  166:         // ACTUAL PRINTING
```

```
  167:         std::cout << "SYMBOL TABLE: (Starting With Most Recently Added Entry)" << std::endl << std::endl;
  168:         outfile << "SYMBOL TABLE: (Starting With Most Recently Added Entry)" << std::endl << std::endl;
  169:
  170:     // Iterate through points by using the "next" pointer on each value
  171:     temp_cnt = 0;
  172:         temp = symtbl_master;
  173:     while(temp->next != NULL)
  174:     {
  175:         // To terminal
  176:         std::cout << "\tEntry #"  << std::right << std::setfill('0')
  177:                           << std::setw(entry_no_length) << std::dec << temp_cnt;
  178:                 std::cout << " | Label: " << std::left << std::setfill(' ')
  179:                                   << std::setw(max_label_length) << temp->label;
  180:                 // Values of -1 (Unknowns) will appear as ffff (twos compliment output)
  181:                 std::cout << " | Value: " << std::right << std::setfill('0')
  182:                                   << std::setw(4) << std::hex << (unsigned short)temp->value;
  183:                 std::cout << " | Line #"  << std::right << std::setfill('0')
  184:                                   << std::setw(line_no_length) << std::dec << temp->line;
  185:                 std::cout << " | type: "  << types[temp->type] << std::endl;
  186:
  187:         // To diagnostics
  188:         outfile << "\tEntry #"    << std::right << std::setfill('0')
  189:                         << std::setw(entry_no_length) << std::dec << temp_cnt;
  190:                 outfile << " | Label: " << std::left << std::setfill(' ')
  191:                                   << std::setw(max_label_length) <<temp->label;
  192:                 // Values of -1 (Unknowns) will appear as ffff (twos compliment output)
  193:                 outfile << " | Value: " << std::right << std::setfill('0')
  194:                                   << std::setw(4) << std::hex << (unsigned short)temp->value;
  195:                 outfile << " | Line #"  << std::right << std::setfill('0')
  196:                                   << std::setw(line_no_length) << std::dec << temp->line;
  197:                 outfile << " | type: "  << types[temp->type] << std::endl;
  198:
  199:         temp = temp->next;
  200:         temp_cnt++;
  201:     }
  202:     std::cout << std::endl;
  203: }
  204:
  205: /*
  206:     Function: get_symbol
  207:     Input: label: The string symbol to search for.
  208:     Output: symtbl_entry*: Pointer to a symbol table entry, this is NULL
  209:                 if the symbol is not found.
  210:     Brief: This function linearly searches the symbol table starting with
  211:                 the most recently added symbol. When the correct symbol is
  212:                 found, it is returned.
  213: */
  214: symtbl_entry* get_symbol(std::string label)
  215: {
  216:         symtbl_entry* temp = symtbl_master;
  217:
  218:         while(temp->next != NULL)
  219:         {
  220:                 if(temp->label == label) return temp;
  221:                 temp=temp->next;
  222:         }
  223:         if(temp->label == label) return temp;
  224:         return NULL;
  225: }
  226:
  227: /*
  228:     Function: valid_symbol
  229:     Input: token: The string token to check the validity of
  230:     Output: bool: True means the symbol is valid, false means it is not
  231:     Brief: This function checks whether or not the given string is a valid
  232:                 symbol or not. This is used before calling "add_symbol" as an
  233:                 error checker.
  234: */
  235: bool valid_symbol(std::string token)
  236: {
  237:         inst_dir* id_ptr = get_inst_dir(token, I);
  238:         if(id_ptr != NULL) return false; // Symbol cannot be an instruction
  239:         id_ptr = get_inst_dir(token, D);
  240:         if(id_ptr != NULL) return false; // Symbol cannot be a directive
  241:
  242:         if(token.length() > MAX_SYM_LENGTH) return false; // Symbol cannot be longer than 31 characters
  243:
  244:         // First token must be alphabetic (A(65) to Z(90), a(97) to z(122), or _(95))
  245:         else if(((token[0] >= 65) && (token[0] <= 90))||((token[0] >= 97) && (token[0] <= 122))||(token[0]
== 95))
  246:         {
  247:                 int temp_cnt = 1;
  248:
```

```
  249:                 while(temp_cnt < token.length())
  250:                 {
  251:                         // Remaining tokens can be alphanumeric ('A'(65) to 'Z'(90), 'a'(97) to 'z'(122), '
0'(48) to '9'()57, or '_'(95))
  252:                         if(!((token[temp_cnt] >= 65 && token[temp_cnt] <= 90)||(token[temp_cnt] >= 97 && to
ken[temp_cnt] <= 122)||(token[temp_cnt] == 95)||((token[temp_cnt] >= 48)&&(token[temp_cnt] <= 57)))) break;
  253:                         temp_cnt++;
  254:                 }
  255:                 return (temp_cnt == token.length()) ? true : false;
  256:         }
  257:         else return false;
  258: }
  259:
  260: /*
  261:     Function: symtbl_unknown_check()
  262:     Brief: This function linearly scans through the symbol table increasining
  263:                 the error count for every UNKNOWN label found. This is called after
  264:                 the first pass is complete in order to ensure everything is in order
  265:                 before starting the second pass.
  266: */
  267: void symtbl_unknown_check()
  268: {
  269:         symtbl_entry* se_ptr = symtbl_master;
  270:
  271:         int starting_err_cnt = err_cnt;
  272:
  273:         std::cout << std::endl << "\tChecking Symbol Table for Unresolved Unknowns:" << std::endl;
  274:
  275:         while(se_ptr->next != NULL)
  276:         {
  277:                 if(se_ptr->type == UNKNOWN) err_cnt++;
  278:                 se_ptr = se_ptr->next;
  279:         }
  280:
  281:         if(err_cnt == starting_err_cnt)
  282:         {
  283:                 std::cout << std::endl << "\t\tNo unknowns found in the symbol table" << std::endl;
  284:                 outfile << std::endl << "\t\tNo unknowns found in the symbol table" << std::endl;
  285:         }
  286:         else
  287:         {
  288:                 std::cout << std::endl << "\t\tTotal unknowns found on the symbol table: >>" << (err_cnt -
starting_err_cnt) << "<< (See symbol table below for UNKNOWN line numbers)" << std::endl;
  289:                 outfile << std::endl << "\t\tTotal unknowns found on the symbol table: >>" << (err_cnt - st
arting_err_cnt) << "<< (See symbol table below for UNKNOWN line numbers)" << std::endl;
  290:         }
  291:
  292: }
```

```
/*
__/\\\\\\\\\\\\\\_____/\\\\\\\\\\\__/\\\\\\\\\\\\\____
 _\/\\\/////////\\\__\/////\\\///__\/\\\/////////\\\__
  _\/\\\_______\/\\\______\/\\\_____\/\\\_______\/\\\_
   _\/\\\\\\\\\\\\\/_______\/\\\_____\/\\\_______\/\\\_
    _\/\\\/////////_________\/\\\_____\/\\\_______\/\\\_
     _\/\\\__________________\/\\\_____\/\\\_______\/\\\_
      _\/\\\_____________/\\\___\/\\\_____\/\\\_______/\\\__
       _\/\\\_____________\//\\\\\\\\\_____\/\\\\\\\\\\\\\/___
        _\///_______________\/////////______\/////////////_____

-> Name: inst_dir.cpp
-> Brief: Implementation file for the inst_dir file
-> Date: May 15, 2017   (Created)
-> Author: Paul Duchesne (B00332119)
-> Contact: pl332718@dal.ca
*/

#include <iostream>
#include <stdio.h>
#include <string>
#include <fstream>
#include <cstdlib>
#include <algorithm>

#include "Include/inst_dir.h"

/*
    Struct: inst_dir_array
    Brief: Array containing all instructions and directives in
            a tidy table for getting at later.
*/
inst_dir inst_dir_array[] = {
        {"ADD",  DOUBLE, 0x5000, WORD},           // Instructions on slots 0,60
                {"ADD.B",  DOUBLE, 0x5000, BYTE},
                {"ADD.W",  DOUBLE, 0x5000, WORD},
        {"ADDC", DOUBLE, 0x6000, WORD},
                {"ADDC.B", DOUBLE, 0x6000, BYTE},
                {"ADDC.W", DOUBLE, 0x6000, WORD},
        {"AND",  DOUBLE, 0xf000, WORD},
                {"AND.B",  DOUBLE, 0xf000, BYTE},
                {"AND.W",  DOUBLE, 0xf000, WORD},
        {"BIC",  DOUBLE, 0xc000, WORD},
                {"BIC.B",  DOUBLE, 0xc000, BYTE},
                {"BIC.W",  DOUBLE, 0xc000, WORD},
        {"BIS",  DOUBLE, 0xd000, WORD},
                {"BIS.B",  DOUBLE, 0xd000, BYTE},
                {"BIS.W",  DOUBLE, 0xd000, WORD},
        {"BIT",  DOUBLE, 0xb000, WORD},
                {"BIT.B", DOUBLE, 0xb000, BYTE},
                {"BIT.W", DOUBLE, 0xb000, WORD},
        {"CALL", SINGLE, 0x1280, WORD},
        {"CMP",  DOUBLE, 0x9000, WORD},
                {"CMP.B", DOUBLE, 0x9000, BYTE},
                {"CMP.W", DOUBLE, 0x9000, WORD},
        {"DADC", DOUBLE, 0xa000, WORD},
                {"DADC.B", DOUBLE, 0xa000, BYTE},
                {"DADC.W", DOUBLE, 0xa000, WORD},
        {"JC",   JUMP, 0x2c00, OFFSET},
        {"JEQ",  JUMP, 0x2400, OFFSET},
        {"JGE",  JUMP, 0x3400, OFFSET},
        {"JHS",  JUMP, 0x2c00, OFFSET},
        {"JL",   JUMP, 0x3800, OFFSET},
        {"JLO",  JUMP, 0x2800, OFFSET},
        {"JMP",  JUMP, 0x3c00, OFFSET},
        {"JN",   JUMP, 0x3000, OFFSET},
        {"JNC",  JUMP, 0x2800, OFFSET},
        {"JNE",  JUMP, 0x2000, OFFSET},
        {"JNZ",  JUMP, 0x2000, OFFSET},
        {"JZ",   JUMP, 0x2400, OFFSET},
        {"MOV",  DOUBLE, 0x4000, WORD},
                {"MOV.B",  DOUBLE, 0x4000, BYTE},
                {"MOV.W",  DOUBLE, 0x4000, WORD},
           {"PUSH", SINGLE, 0x1200, WORD},
                {"PUSH.B", SINGLE, 0x1200, BYTE},
                {"PUSH.W", SINGLE, 0x1200, WORD},
        {"RETI", NONE, 0x1300, WORD},
        {"RRA",  SINGLE, 0x1100, WORD},
                {"RRA.B", SINGLE, 0x1100, BYTE},
                {"RRA.W", SINGLE, 0x1100, WORD},
        {"RRC",  SINGLE, 0x1000, WORD},
                {"RRC.B", SINGLE, 0x1000, BYTE},
                {"RRC.W", SINGLE, 0x1000, WORD},
```

```
   84:         {"SUB",  DOUBLE, 0x8000, WORD},
   85:                 {"SUB.B",  DOUBLE, 0x8000, BYTE},
   86:                 {"SUB.W",  DOUBLE, 0x8000, WORD},
   87:         {"SUBC", DOUBLE, 0x7000, WORD},
   88:                 {"SUBC.B", DOUBLE, 0x7000, BYTE},
   89:                 {"SUBC.W", DOUBLE, 0x7000, WORD},
   90:         {"SWPB", SINGLE, 0x1080, WORD},
   91:         {"SXT",  SINGLE, 0x1180, WORD},
   92:         {"XOR",  DOUBLE, 0xe000, WORD},
   93:                 {"XOR.B",  DOUBLE, 0xe000, BYTE},
   94:                 {"XOR.W",  DOUBLE, 0xe000, WORD},
   95:         {"ALIGN",  NONE, 0xffff, WORD},  // Directives on slots 61,68
   96:         {"BSS",    NONE, 0xffff, WORD},
   97:         {"BYTE",   NONE, 0xffff, WORD},
   98:         {"END",    NONE, 0xffff, WORD},
   99:         {"EQU",    NONE, 0xffff, WORD},
  100:         {"ORG",    NONE, 0xffff, WORD},
  101:         {"STRING", NONE, 0xffff, WORD},
  102:         {"WORD"  , NONE, 0xffff, WORD},
  103: };
  104:
  105: /*
  106:     Function: get_inst_dir
  107:     Input: input: A string to search for; stype: the type to search for (Inst or DIR)
  108:     Output: inst_dir*: A pointer to the matching instruction or directive.
  109:     Brief: Performs a binary search that returns the value searched for or else a NULL
  110:             pointer. All input is transformed to uppercase because instructions and
  111:             directives are case insensitive.
  112: */
  113: inst_dir* get_inst_dir(std::string input, SEARCHTYPE stype)
  114: {
  115:         std::transform(input.begin(), input.end(), input.begin(), ::toupper);
  116:
  117:         int bottom = (stype == I) ? 0  : 61;   // The lower range of Instructions is 0, while the lower
  118:                                                //  range of Directives is 61
  119:         int top    = (stype == I) ? 60 : 68;   // The upper range of Instructions is 60, while the upper
  120:                                                // range of Directives is 68
  121:         int char_cnt = 0;
  122:         int cnt = 0;
  123:         if(stype != I && stype != D)
  124:         {
  125:                 std::cout << "ERROR: INVALID SEARCH TYPE" << std::endl;
  126:                 return NULL;
  127:         }
  128:     // Check top/bottom values
  129:     if(inst_dir_array[top].mnemonic == input)
  130:     {
  131:         return &inst_dir_array[top];
  132:     }
  133:     if(inst_dir_array[bottom].mnemonic == input)
  134:     {
  135:         return &inst_dir_array[bottom];
  136:     }
  137:     // Iterate through the list, slowly narrowing down the target search
  138:     // until the value is found or the search window gets small enough.
  139:     while(top-bottom != 1)
  140:     {
  141:         cnt = (top+bottom)/2;
  142:         if (inst_dir_array[cnt].mnemonic == input)
  143:         {
  144:             // std::string temp_str = (cnt <= 60) ? "[INST]" : "[DIR]";
  145:             return &inst_dir_array[cnt];
  146:         }
  147:
  148:         if(inst_dir_array[cnt].mnemonic > input) top = cnt;
  149:         else bottom = cnt;
  150:     }
  151:         return NULL;
  152: }
```

```
 1: /*
 2: __/\\\\\\\\\\\\\\_____/\\\\\\\\\\\__/\\\\\\\\\\\\\____
 3:  _\/\\\/////////\\\__\/////\\\///__\/\\\/////////\\\__
 4:   _\/\\\_______\/\\\______\/\\\_____\/\\\_______\/\\\_
 5:    _\/\\\\\\\\\\\\\/_______\/\\\_____\/\\\_______\/\\\_
 6:     _\/\\\/////////_________\/\\\_____\/\\\_______\/\\\_
 7:      _\/\\\__________________\/\\\_____\/\\\_______\/\\\_
 8:       _\/\\\_____________/\\\___\/\\\_____\/\\\_______/\\\__
 9:        _\/\\\____________\//\\\\\\\\\_____\/\\\\\\\\\\\\\/___
10:         _\///______________\/////////______\/////////////_____
11:
12: -> Name:  library.cpp
13: -> Brief: Implementation for the library.cpp, a helper library
14: -> Date: May 15, 2017   (Created)
15: -> Author: Paul Duchesne (B00332119)
16: -> Contact: pl332718@dal.ca
17: */
18:
19: #include <iostream>
20: #include <stdio.h>
21: #include <string>
22: #include <cstring>
23: #include <fstream>
24: #include <cstdlib>
25:
26: #include "Include/library.h"
27:
28: /*
29:         Function: fft (find first token)
30:         Output: std::string: The first token of the new record
31:         Input: fin: the input .asm input file open to the assembly code
32:         Brief: Fetches next the record and passes back the first token
33:                         after removing any comments from the line. If there is no
34:                         token on the line, "" is returned.
35: */
36: std::string fft(std::istream& fin)
37: {
38:         std::getline(fin, current_record);
39:
40:         std::string token;
41:
42:         // Remove comment from line
43:         current_record = current_record.substr(0, current_record.find_first_of(";"));
44:
45:         // Used to fix an error an error
46:         current_record = " " + current_record;
47:
48:         char* temp_crecord = new char[current_record.length()+1];
49:
50:         std::strcpy(temp_crecord, current_record.c_str()+1);
51:
52:         char* temp_ctoken = std::strtok(temp_crecord, " \t\n");
53:
54:         if (temp_ctoken == NULL) return "";
55:
56:         token.assign(temp_ctoken, strlen(temp_ctoken));
57:
58:         return token;
59: }
60:
61: /*
62:         Function: fnt (find next token)
63:         Output: std::string: The next token of the current record
64:         Brief: Using the internal storage of 'strtok', the next
65:                         token is returned without having to pass the file
66:                         pointer in. If there is no token on the line, ""
67:                         is returned.
68: */
69: std::string fnt()
70: {
71:         std::string token;
72:
73:         char* temp_ctoken = strtok(NULL, " \t\n");
74:
75:         if (temp_ctoken != NULL) token.assign(temp_ctoken, strlen(temp_ctoken));
76:         else return "";
77:
78:         return token;
79: }
80:
```

```
    1: /*
    2: __/\\\\\\\\\\\\\\_____/\\\\\\\\\\\__/\\\\\\\\\\\\\\____
    3:  _\/\\\/////////\\\__\/////\\\///__\/\\\/////////\\\__
    4:   _\/\\\_______\/\\\______\/\\\_____\/\\\_______\/\\\_
    5:    _\/\\\\\\\\\\\\\/_______\/\\\_____\/\\\_______\/\\\_
    6:     _\/\\\/////////_________\/\\\_____\/\\\_______\/\\\_
    7:      _\/\\\__________________\/\\\_____\/\\\_______\/\\\_
    8:       _\/\\\_____________/\\\___\/\\\_____\/\\\_______/\\\__
    9:        _\/\\\____________\//\\\\\\\\\_____\/\\\\\\\\\\\\\/___
   10:         _\///______________\/////////______\/////////////_____
   11:
   12: -> Name:  second_pass.cpp
   13: -> Brief: Function file for second_pass.cpp
   14: -> Date: May 24, 2017   (Created)
   15: -> Author: Paul Duchesne (B00332119)
   16: -> Contact: pl332718@dal.ca
   17: */
   18:
   19: #include <iostream>
   20: #include <stdio.h>
   21: #include <string>
   22: #include <cstring>
   23: #include <fstream>
   24: #include <cstdlib>
   25:
   26: #include "Include/second_pass.h"
   27: #include "Include/library.h"
   28: #include "Include/symtbl.h"
   29: #include "Include/inst_dir.h"
   30: #include "Include/parser.h"
   31: #include "Include/emitter.h"
   32: #include "Include/s19_maker.h"
   33:
   34: // Used to signify that the state machine should stop
   35: bool end_flag = false;
   36:
   37: /*
   38:         Function: second_pass
   39:         Input: fin: The input file to read records from.
   40:         Brief: The second pass state machine functions similarly to the first
   41:                         pass state machine, but somewhat simpler. There are far fewew
   42:                         error checks, and the check operand states no longer exist.
   43:                         In fact, most of the effort for the second pass is within the
   44:                         emit function that is called from the second pass. Effectively,
   45:                         the second pass parses the input token until it finds an inst, in
   46:                         which case it calls emit and goes to the next line, or until it
   47:                         finds a directive, in which case it performs the directive action.
   48: */
   49: void second_pass()
   50: {
   51:         // NOTE: This state machine is very similar to the first pass state
   52:         // machine, thus it is far less commented
   53:
   54:         // Open s19 record file
   55:         srec_file.open("srec_output.s19");
   56:
   57:         STATE next_state = CHK_FIRST_TOKEN;
   58:
   59:         // Local LC variable
   60:         int LC = 0;
   61:
   62:         bool directive_error_flag = false;
   63:
   64:         ADDR_MODE addr_mode = WRONG;
   65:
   66:         inst_dir* id_ptr = NULL;
   67:
   68:         symtbl_entry* symtbl_ptr = NULL;
   69:
   70:         int string_cnt = 0;
   71:         int string_esc_cnt = 0; // Used to keep track of number of bytes saved by using ^H instead of \t (o
r similarly escaped character)
   72:
   73:         int value0 = -1;
   74:         int value1 = -1;
   75:
   76:         std::string last_label = ""; // Used by EQU
   77:
   78:         init_srec(0);
   79:
   80:         // Output the diagnostics format to diagnostics.LIS file
   81:         outfile << "\tRecord #n: >>Input Record<<" << std::endl
   82:                 << "\t\tMemloc INST" << std::endl
```

```
   83:                 << "\t\tMemloc [SRC Address] (If it exists)" << std::endl
   84:                 << "\t\tMemloc [DST Address] (If it exists)" << std::endl;
   85:
   86:         while(!fin.eof() && !end_flag)
   87:         {
   88:                 switch (next_state)
   89:                 {
   90:                         case CHK_FIRST_TOKEN:
   91:                                 line_num++;
   92:                                 current_token = fft(fin);
   93:
   94:                                 outfile << std::endl << "\tRecord #" << line_num
   95:                                         << ": >>" << current_record << "<<" << std::endl;
   96:
   97:                                 // If there is an empty line, move on to next line
   98:                                 if(current_token == "")
   99:                                 {
  100:                                         next_state = CHK_FIRST_TOKEN;
  101:                                         break;
  102:                                 }
  103:
  104:                                 // Check if token is an instruction
  105:                                 id_ptr = get_inst_dir(current_token, I);
  106:                                 if(id_ptr != NULL)
  107:                                 {
  108:                                         next_state = INST;
  109:                                         break;
  110:                                 }
  111:
  112:                                 // Check if token is a directive
  113:                                 id_ptr = get_inst_dir(current_token, D);
  114:                                 if(id_ptr != NULL)
  115:                                 {
  116:                                         next_state = DIRECT;
  117:                                         break;
  118:                                 }
  119:
  120:                                 // Otherwise it is a label, in which case it is skipped
  121:                                 next_state = CHK_NEXT_TOKEN;
  122:
  123:                                         break;
  124:                         case CHK_NEXT_TOKEN:
  125:                                 // If there is an empty token, move on to next line
  126:                                 current_token = fnt();
  127:                                 if(current_token == "") // Line only had a label (and maybe a comment)
  128:                                 {
  129:                                         next_state = CHK_FIRST_TOKEN;
  130:                                         break;
  131:                                 }
  132:
  133:                                 // Check if token is an instruction
  134:                                 id_ptr = get_inst_dir(current_token, I);  // Check if it is a valid INST
  135:                                 if(id_ptr != NULL)
  136:                                 {
  137:                                         next_state = INST;
  138:                                         break;
  139:                                 }
  140:
  141:                                 // Check if token is a directive
  142:                                 id_ptr = get_inst_dir(current_token, D);  // Check if it is a valid DIRECTI
VE
  143:                                 if(id_ptr != NULL)
  144:                                 {
  145:                                         next_state = DIRECT;
  146:                                         break;
  147:                                 }
  148:
  149:                                 // This should never happen, all such errors would be caught in the first p
ass
  150:                                 next_state = CHK_FIRST_TOKEN;
  151:
  152:                                 break;
  153:                         case INST:
  154:                                 next_state = CHK_FIRST_TOKEN;
  155:
  156:                                 switch(id_ptr->type)
  157:                                 {
  158:                                         case NONE:
  159:                                                 emit(id_ptr->mnemonic, "", NONE, LC);
  160:                                                 break;
  161:                                         case SINGLE:
  162:                                                 emit(id_ptr->mnemonic, fnt(), SINGLE, LC);
  163:                                                 break;
```

```
  164:                                    case DOUBLE:
  165:                                            emit(id_ptr->mnemonic, fnt(), DOUBLE, LC);
  166:                                            break;
  167:                                    case JUMP:
  168:                                            emit(id_ptr->mnemonic, fnt(), JUMP, LC);
  169:                                            break;
  170:                                    default:
  171:                                            std::cout << "[INST] THIS SHOULD NEVER TRIGGER" << std::end
l;
  172:                                            getchar();
  173:                                            break;
  174:                            }
  175:
  176:                            break;
  177:
  178:                    case DIRECT: // id_ptr should already point to the correct DIR
  179:                            // No matter the outcome of the directive (error or not),
  180:                            // the next state is CHK_FIRST_TOKEN
  181:                            next_state = CHK_FIRST_TOKEN;
  182:
  183:                            // The directive is assumed to have caused an error until proven otherwise
  184:                            directive_error_flag = true;
  185:
  186:                            // If the type is not ALIGN, END, or STRING, the value needs to be parsed
  187:                            if(id_ptr->mnemonic[1] != 'L' && id_ptr->mnemonic[1] != 'N' && id_ptr->mnem
onic[1] != 'T')
  188:                            {
  189:                                    current_token = fnt();  // Find next token
  190:
  191:                                    current_token = "#" + current_token; // tell parser to process as I
MMEDIATE
  192:                                    addr_mode = parse(current_token, value0, value1);
  193:
  194:                                    if(addr_mode == IMMEDIATE) directive_error_flag = false;
  195:                                    else error_detected_no_cnt("Directive: Unknown Label after DIRECTIV
E (Value0 parsing)");
  196:                            }
  197:
  198:                            switch (id_ptr->mnemonic[1])
  199:                            {
  200:                                    case 'L':  // Align
  201:                                            if(LC%2)
  202:                                            {
  203:                                                    LC++;
  204:                                                    // Output the buffer in order to avoid filling the
skipped memory location
  205:                                                    output_srec_buffer();
  206:                                                    init_srec(LC);
  207:                                            }
  208:                                            break;
  209:
  210:                                    case 'S':  // BSS
  211:                                            if(!directive_error_flag)
  212:                                            {
  213:                                                    if(value0 >= 0)
  214:                                                    {
  215:                                                            LC += value0; // Bounds checked in first pa
ss
  216:                                                            output_srec_buffer();
  217:                                                            init_srec(LC);
  218:                                                    }
  219:                                                    else error_detected_no_cnt("Directive: Negative val
ue for BSS");
  220:                                            }
  221:                                            break;
  222:
  223:                                    case 'Y':  // BYTE
  224:                                            if(!directive_error_flag)
  225:                                            {
  226:                                                    if(value0 >= MINBYTE && value0 <= MAXBYTE)
  227:                                                    {
  228:                                                            write_srec_byte((unsigned char)value0);
  229:                                                            LC += 1;
  230:                                                    }
  231:                                                    else error_detected_no_cnt("Directive: Value too la
rge for BYTE directive");
  232:                                            }
  233:
  234:                                            break;
  235:
  236:                                    case 'N':  // END
  237:                                            current_token = fnt();  // Find next token
  238:                                            symtbl_ptr = get_symbol(current_token);  // Get symbol or N
```

```
ULL
  239:                                                  if(symtbl_ptr != NULL)
  240:                                                  {
  241:                                                          write_S9(symtbl_ptr->value); // Error checking done
 in first pass
  242:                                                          std::cout << "END VALUE >>" << symtbl_ptr->value <<
"<<"<<std::endl;
  243:                                                          std::cout << "END VALUE >>" << symtbl_ptr->label <<
"<<"<<std::endl;
  244:                                                  }
  245:                                                  else write_S9(0);
  246:                                                  end_flag = true;
  247:                                                  break;
  248:
  249:                                          case 'Q':  // EQU (Handled in the first pass)
  250:                                                  break;
  251:
  252:                                          case 'R':  // ORG
  253:                                                  if(!directive_error_flag)
  254:                                                  {
  255:                                                          LC = value0;
  256:                                                          output_srec_buffer();
  257:                                                          init_srec(LC);
  258:                                                  }
  259:                                                  break;
  260:
  261:                                          case 'T':  // STRING
  262:                                                  current_token = fnt();
  263:
  264:                                                  // Error detecting for this was done in the first pass
  265:                                                  current_token.erase(0,1); // Removes Opening Quote
  266:
  267:                                                  // ITERATE THROUGH LOOKING FOR ESCAPE CHARACTERS
  268:                                                  for(string_cnt = 0; string_cnt < current_token.length(); st
ring_cnt++)
  269:                                                  {
  270:                                                          if(current_token[string_cnt] == '\\')
  271:                                                          {
  272:                                                                  string_esc_cnt++;
  273:                                                                  string_cnt++;
  274:                                                                  // USED IN SECOND PASS TO SWITCH TO ESCAPE
SEQUENCES
  275:                                                                  switch (current_token[string_cnt])
  276:                                                                  {
  277:                                                                          case 't': // TAB
  278:                                                                                  write_srec_byte(0x09); // 0
x09 = ^H, which is the escape sequence for \t
  279:                                                                                  break;
  280:                                                                          case '0': // NULL
  281:                                                                                  write_srec_byte(0x00); // 0
x00 = ^@, which is the escape sequence for \0
  282:                                                                                  break;
  283:                                                                          case 'r': // RETURN/ENTER
  284:                                                                                  write_srec_byte(0x0d); // 0
x0d = ^M, which is the escape sequence for \r
  285:                                                                                  break;
  286:                                                                          case 'n': // NEW LINE
  287:                                                                                  write_srec_byte(0x0a); // 0
x0a = ^J, which is the escape sequence for \n
  288:                                                                                  break;
  289:                                                                          // Other sequences could be impleme
nted here
  290:                                                                          default: // Therefore the character
 is something else (Such as double quotes)
  291:                                                                                  write_srec_byte(current_tok
en[string_cnt]);   // Stores backslash
  292:                                                                                  break;
  293:                                                                  }
  294:                                                          }
  295:                                                          else if(current_token[string_cnt] != '\"') // Looki
ng for end quote
  296:                                                          {
  297:                                                                  write_srec_byte(current_token[string_cnt]);
  298:                                                          }
  299:                                                  }
  300:
  301:                                                  LC += current_token.length();
  302:                                                  LC -= string_esc_cnt; // For every esc character converted
from \t to ^H, a byte is saved
  303:
  304:                                                  string_esc_cnt = 0; // Reset escape count
  305:
  306:                                                  break;
```

```
  307:
  308:                                        case 'O':  // WORD
  309:                                                if(!directive_error_flag)
  310:                                                {
  311:                                                        if(LC%2) LC += 0x01;  // Align LC first
  312:                                                        if(value0 >= MINWORD && value0 <= MAXWORD)
  313:                                                        {
  314:                                                                write_srec_word(value0);
  315:                                                                LC += 2;
  316:                                                        }
  317:                                                        else error_detected_no_cnt("Directive: Value too la
rge for WORD directive");
  318:                                                }
  319:                                                break;
  320:
  321:                                        default:
  322:                                                std::cout << "\t\t[Directive] DEFAULT ERROR" << std::endl;
  323:                                                getchar();  // This should never happen, getchar will stop
the runtime
  324:                                                                        // and let the user know there is a
 serious assembler flaw
  325:                                                break;
  326:                                }
  327:
  328:                                break;
  329:                        default:
  330:                                std::cout << "\t\t[Second Pass] DEFAULT ERROR" << std::endl;
  331:                                getchar();  // This should never happen, getchar will stop the runtime
  332:                                                // and let the user know there is a serious assembl
er flaw
  333:                                break;
  334:                }
  335:
  336:         }
  337:
  338:         // Output any bytes still in the buffer and then close with an S9
  339:         if(!end_flag) write_S9(0);
  340: }
  341:
  342: /*
  343: Function: error_detected_no_cnt
  344: Input: error_msg: Error msg to display if triggered
  345: Brief: Because there should be 0 errors in the second pass, this function
  346: will actually stop the program with a getchar() to prompt the user
  347: that a catastrophic failure has occurred. This is primarily for
  348: testing and debugging
  349:  */
  350: void error_detected_no_cnt(std::string error_msg)
  351: {
  352:         std::cout << "\t\t[ERROR MSG - SECOND PASS] " << error_msg << std::endl;
  353:         outfile << "\t\t[ERROR MSG - SECOND PASS] " << error_msg << std::endl;
  354:         getchar();
  355:         end_flag = true; // Ends run because an error is found on second pass
  356:                                         // This is done due to the fact that this is more of an issue
  357:                                         // on the second pass given that all errors should be caught on
  358:                                         // the first pass
  359: }
```

```
    1: /*
    2: __/\\\\\\\\\\\\\\\_____/\\\\\\\\\\\__/\\\\\\\\\\\\\\\____
    3:  _\/\\\/////////\\\__\/////\\\///__\/\\\/////////\\\__
    4:   _\/\\\_______\/\\\______\/\\\_____\/\\\_______\/\\\_
    5:    _\/\\\\\\\\\\\\\/_______\/\\\_____\/\\\_______\/\\\_
    6:     _\/\\\/////////_________\/\\\_____\/\\\_______\/\\\_
    7:      _\/\\\__________________\/\\\_____\/\\\_______\/\\\_
    8:       _\/\\\_____________/\\\___\/\\\_____\/\\\_______/\\\__
    9:        _\/\\\____________\//\\\\\\\\\_____\/\\\\\\\\\\\\\/___
   10:         _\///______________\/////////______\/////////////_____
   11:
   12: -> Name:  parser.h
   13: -> Brief: Header file for parser.h.cpp
   14: -> Date: May 18, 2017
   15: -> Author: Paul Duchesne (B00332119)
   16: -> Contact: pl332718@dal.ca
   17: */
   18:
   19: #include <iostream>
   20: #include <stdio.h>
   21: #include <string>
   22: #include <cstring>
   23: #include <fstream>
   24: #include <cstdlib>
   25:
   26: #include "Include/parser.h"
   27: #include "Include/symtbl.h"
   28: #include "Include/library.h"
   29:
   30: /*
   31:     Function: parse
   32:     Input: op: String containing the operand,
   33:            value0: Pointer to the general return value for most addressing modes
   34:            value1: Pointer to the typically unused value storage of the register in indexed mode
   35:     Output: ADDR_MODE: An enumeration of the addressing mode associated with the given input
   36:     Brief: This function takes in an operand string and parses it into one of the 7 addressing
   37:                 modes, or into an error (Denoted by the "WRONG" addressing mode). This is done
   38:                 through a switch statement and many many checks.
   39: */
   40: ADDR_MODE parse(std::string op, int& value0, int& value1)
   41: {
   42:         std::string operand = op;   // Make a copy of the operand in case the original is needed
   43:         std::string temp_indexed;       // Used to store the register in indexed mode
   44:
   45:         bool hex_flag = false;
   46:
   47:         int temp_cnt = 0;
   48:
   49:         // Setting the return values to a default -1
   50:         value0 = -1;  // The general return value
   51:         value1 = -1;  // The return value of the REGISTER in INDEXED mode
   52:
   53:         // General use pointer to a symtbl entry
   54:         symtbl_entry* se_ptr = NULL;
   55:
   56:         // Flag for auto increment mode
   57:         bool auto_flag = false;
   58:
   59:         switch (operand[0])     // Checking first character of operand
   60:         {
   61:                 case '&': // Absolute Mode (Or Error)
   62:                         /*      BRIEF STATE SUMMARY:
   63:                                 This state handles the absolute addressing mode by checking
   64:                                 the validity of the value. To be valid, the operand must either
   65:                                 already be on the symbol table (backwards reference) or be a
   66:                                 valid symbol and then added to the symbol table (forward reference)
   67:                         */
   68:                         // Remove '&' from operand for future parsing
   69:                         operand.erase(0, 1);
   70:
   71:                         // The input '&' is an error
   72:                         if(operand == "") return WRONG;
   73:
   74:                         se_ptr = get_symbol(operand);
   75:
   76:                         // If symbol not in table, and the operand is a valid label, add to symbol table
   77:                         if(se_ptr == NULL && valid_symbol(operand))
   78:                         {
   79:                                 add_symbol(operand, -1, UNKNOWN);  // This creates the forward reference
   80:                                 if(value0 < MINWORD || value0 > MAXWORD) return WRONG; // Value0 cannot be
larger than a word
   81:                                 else return ABSOLUTE;
   82:                         }
```

```
   83:                     else if(se_ptr != NULL)
   84:                     {
   85:                             // The se_ptr came back with something, ensure it is not a REG
   86:                             if(se_ptr->type != REG)
   87:                             {
   88:                                     value0 = se_ptr->value;
   89:                                     return ABSOLUTE;
   90:                             }
   91:                             else return WRONG; // Registers cannot be used in absolute mode
   92:                     }
   93:                     else return WRONG; // Therefore not in symbol table and is also not valid symbol
   94:                     break;
   95:             case '@':       // (64) Indirect or Indirect auto-increment (OR BUST)
   96:                     /*      BRIEF STATE SUMMARY:
   97:                             This state handles the indirect and indirect auto-increment
   98:                             addressing mode of registers. This is done by searching the
   99:                             symbol table for the input operand (minus the beginning @), and
  100:                             then checking if the last character is a "+". If a plus sign is
  101:                             found in the correct spot, the auto-increment flag is set and
  102:                             the plus is removed. The remaining string is searched for in the
  103:                             symbol table and the result is checked to be a register or not.
  104:                     */
  105:                     // Remove '@' from operand for future parsing
  106:                     operand.erase(0, 1);
  107:
  108:                     // "@" is an invalid operand
  109:                     if(operand == "") return WRONG;
  110:
  111:                     if(operand.find_last_of("+") == operand.length()-1)
  112:                     {
  113:                             operand.erase(operand.length()-1, operand.length());
  114:                             // "@+" is in an invalid operand
  115:                             if(operand == "") return WRONG;
  116:                             auto_flag = true;
  117:                     }
  118:
  119:                     // Check symbol table for the remaining operand, it must be a REG
  120:                     se_ptr = get_symbol(operand);
  121:                     if (se_ptr == NULL) return WRONG;                       // Invalid symbol from ind
irect or indirect+
  122:                     else if (se_ptr->type !=REG) return WRONG;   // Invalid symbol type from indirect o
r indirect+
  123:                     else value0 = se_ptr->value;                            // Register value, will be
 between 0 and 15
  124:                     return (auto_flag ? INDIRECT_AI : INDIRECT);
  125:                     break;
  126:
  127:             case '#':
  128:                     /*      BRIEF STATE SUMMARY:
  129:                             This state handles the immediate addressing mode, including
  130:                             using labels as immediates. First the # symbol is erased, then
  131:                             the symbol table is checked for the current operand. If it is
  132:                             in the symbol table, it must not be a register. If it is not in
  133:                             the symbol table, yet is a valid symbol, it is added as a forward
  134:                             reference. Otherwise, it is parsed as a Decimal or Hex number
  135:                             depending on the first character being $ or not.
  136:                     */
  137:
  138:                     operand.erase(0, 1);
  139:
  140:                     // "#" is in invalid operand
  141:                     if(operand == "") return WRONG; // This means the input was "#" alone
  142:
  143:                     se_ptr = get_symbol(operand);
  144:
  145:                     if(se_ptr == NULL && valid_symbol(operand))
  146:                     {
  147:                             // Forward reference of label within immediate
  148:                             add_symbol(operand, -1, UNKNOWN);
  149:                             value0 = -1;
  150:                             return IMMEDIATE;
  151:                     }
  152:                     if(se_ptr != NULL && se_ptr->type != REG)
  153:                     {
  154:                             // Therefore, constant is the value from the label
  155:                             value0 = se_ptr->value;
  156:                             return IMMEDIATE;
  157:                     }
  158:                     else
  159:                     {
  160:                             // Therefore, value is a HEX number or DECIMAL number
  161:                             if(operand[0] == '$')
  162:                             {
```

```
  163:                                    operand.erase(0, 1);
  164:                                    // "#$" is an invalid operand
  165:                                    if(operand == "") return WRONG;
  166:                                    hex_flag = true;
  167:                            }
  168:                            else if(operand[0] == '-' && operand.length() == 1) return WRONG; // "#-" i
s invalid
  169:                            // Delete preceding 0s
  170:                            while(operand[0] == '0' && operand.length() > 1) operand.erase(0, 1);
  171:
  172:                            if(operand.length() > 8 && hex_flag) return WRONG;              // TOO LONG
 FOR STOL (Hex)
  173:                            if(operand.length() > 10 && !hex_flag) return WRONG;    // TOO LONG FOR STO
L (Decimal)
  174:
  175:                            // Check that all remaining characters are numeric
  176:                            if(hex_flag && operand.find_first_not_of("0123456789abcdefABCDEF") != std::
string::npos) return WRONG;
  177:                            else if(!hex_flag && operand.find_first_not_of("-0123456789") != std::strin
g::npos) return WRONG;
  178:
  179:                            // THEREFORE: "Operand" is numeric and contains a number
  180:                            value0 = std::stol(operand, nullptr, hex_flag ? 16 : 10);
  181:
  182:                            // Value0 cannot be larger than a word
  183:                            if(value0 < MINWORD || value0 > MAXWORD) return WRONG;
  184:                            else return IMMEDIATE;
  185:                    }
  186:                    break;
  187:            default:        // Reg, Indexed, Relative
  188:                    /*      BRIEF STATE SUMMARY:
  189:                            This state handles the remaining addressing modes, register
  190:                            direct, indexed, and relative. First indexed is checked for
  191:                            by searching for "(" and ")" in the token. Next the symbol
  192:                            table is searched to see if it is register direct or relative.
  193:                            Both are handled quite similarly after checking the type of
  194:                            the returned symbol table erntry pointer.
  195:                    */
  196:
  197:                     // If either is not found, 'find_first_of' returns n_pos, which is equal to -1
  198:                    if(operand.find_first_of("(") != -1 && operand.find_first_of(")") != -1)
  199:                    {
  200:                            // Therefore the operand is indexed mode
  201:                            // (Below, if wrong) Invalid INDEXED OPERAND
  202:                            // (Closing bracket just after opening bracket)
  203:                            if(operand.find_first_of("(") + 1 == operand.find_first_of(")")) return WRO
NG;
  204:
  205:                            // (Below, if wrong) Invalid INDEXED OPERAND
  206:                            // (Closing bracket appears before opening bracket)
  207:                            if(operand.find_first_of("(") > operand.find_first_of(")")) return WRONG;
  208:
  209:                            // Obtains x from x(Rn)
  210:                            while(operand[0] != '(')
  211:                            {
  212:                                    temp_indexed += operand[0]; // temp_indexed is the x in x(Rn)
  213:                                    operand.erase(0,1);
  214:                            }
  215:
  216:                            operand.erase(0,1); // Erases the "("
  217:
  218:                            // (Below, if wrong) Invalid closing bracket position (Not last character)
  219:                            if(operand.find_first_of(")") != operand.length()-1) return WRONG;
  220:                            operand.pop_back(); // Removes last character of the string, ")"
  221:
  222:                            // temp_indexed is the x in x(Rn)
  223:                            // operand is now the Rn in x(Rn)
  224:                            // Now check validity of both
  225:
  226:                            // Check validity of X in X(Rn)
  227:                            se_ptr = get_symbol(temp_indexed);
  228:                            // If the symbol is not in the symbol table, and is a valid symbol
  229:                            // add it as a forward reference
  230:                            if(se_ptr == NULL)
  231:                            {
  232:                                    if(valid_symbol(temp_indexed))
  233:                                    {
  234:                                            add_symbol(temp_indexed, -1, UNKNOWN);
  235:                                            value0 = -1;
  236:                                    }
  237:                                    else return WRONG; // Symbol is not in symbol table, and is also in
valid
  238:                            }
```

```
  239:                                    else if(se_ptr->type == REG) return WRONG; // X in x(Rn) cannot be a regist
er
  240:                                    else // X is KNOWN or UNKNOWN, which is valid
  241:                                    {
  242:                                            value0 = se_ptr->value;
  243:                                    }
  244:                                    // Else KNOWN or UNKNOWN, therefore set value0 to se_ptr->value
  245:
  246:                                    // Check validity of Rn in x(Rn)
  247:                                    se_ptr = get_symbol(operand);
  248:                                    if(se_ptr != NULL)
  249:                                    {
  250:                                            if(se_ptr->type == REG) value1 = se_ptr->value;
  251:                                            else return WRONG; // Rn in x(Rn) must be REG type
  252:                                    }
  253:                                    else return WRONG; // Therefore it must be REG type
  254:
  255:                                    return INDEXED;
  256:                            }
  257:                            else if(get_symbol(operand) == NULL && valid_symbol(operand))
  258:                            {
  259:                                    add_symbol(operand, -1, UNKNOWN);
  260:                                    value0 = -1;
  261:                                    // Value0 cannot be larger than a word
  262:                                    if(value0 < MINWORD || value0 > MAXWORD) return WRONG;
  263:                                    else return RELATIVE;
  264:                            }
  265:                            else if(get_symbol(operand) != NULL)
  266:                            {
  267:                                    // Therefore the return is either register direct or relative
  268:                                    se_ptr = get_symbol(operand);
  269:                                    value0 = se_ptr->value;
  270:                                    return (se_ptr->type == REG) ? REG_DIRECT : RELATIVE;
  271:                            }
  272:                            break;
  273:                    }
  274:                    // If the operand was none of the modes above, it is clearly an error.
  275:                    return WRONG;
  276: }
  277:
```

```
    1: /*
    2: __/\\\\\\\\\\\\\\_____/\\\\\\\\\\\__/\\\\\\\\\\\\\____
    3:  _\/\\\/////////\\\__\/////\\\///__\/\\\/////////\\\__
    4:   _\/\\\_______\/\\\______\/\\\_____\/\\\_______\/\\\_
    5:    _\/\\\\\\\\\\\\\/_______\/\\\_____\/\\\_______\/\\\_
    6:     _\/\\\/////////_________\/\\\_____\/\\\_______\/\\\_
    7:      _\/\\\__________________\/\\\_____\/\\\_______\/\\\_
    8:       _\/\\\_____________/\\\___\/\\\_____\/\\\_______/\\\__
    9:        _\/\\\____________\//\\\\\\\\\_____\/\\\\\\\\\\\\\/___
   10:         _\///______________\/////////______\/////////////_____
   11:
   12: -> Name:  first_pass.cpp
   13: -> Brief: Function file for first_pass.cpp
   14: -> Date: May 21, 2017   (Created)
   15: -> Author: Paul Duchesne (B00332119)
   16: -> Contact: pl332718@dal.ca
   17: */
   18:
   19: #include <iostream>
   20: #include <stdio.h>
   21: #include <string>
   22: #include <cstring>
   23: #include <fstream>
   24: #include <cstdlib>
   25:
   26: #include "Include/first_pass.h"
   27: #include "Include/library.h"
   28: #include "Include/symtbl.h"
   29: #include "Include/inst_dir.h"
   30: #include "Include/parser.h"
   31:
   32:
   33: int addr_mode_LC_array[] = {0, 2, 2, 2, 0, 0, 2};
   34:
   35: // A global line number to keep track of which line of the file is being processed
   36: int line_num = 0;
   37:
   38: /*
   39:         Function: first_pass
   40:         Input: fin: The input file to read records from.
   41:         Brief:The first pass performs error checking on the input .asm file
   42:                         and fills the symbol table with all appropriate values. If an error
   43:                         is found, it is recorded and will prevent the second pass from starting.
   44:                         The first pass works by utilizing a state machine that cycles through
   45:                         records individually, one token at a time. Please see the data flow diagram
   46:                         in the Diagrams folder for a general overview of the state transitions.
   47: */
   48: void first_pass()
   49: {
   50:         // Enumeration of the state machine states, initialized to the first state
   51:         STATE next_state = CHK_FIRST_TOKEN;
   52:
   53:         // Location Counter variable
   54:         int LC = 0;
   55:
   56:         // Various flags used within the state machine
   57:         bool end_flag = false;
   58:         bool two_op_flag = false;
   59:         bool directive_error_flag = false;
   60:
   61:         // Enumeration of the addressing mode, used for the return value of the parse function
   62:         ADDR_MODE addr_mode = WRONG;
   63:
   64:         // General use pointer to an instruction/directive table entry
   65:         inst_dir* id_ptr = NULL;
   66:
   67:         // General use pointer to a symbol table entry
   68:         symtbl_entry* symtbl_ptr = NULL;
   69:
   70:         // Return values for the Parse function (defaulted to error of -1)
   71:         int value0 = -1;
   72:         int value1 = -1;
   73:
   74:         // Used in the state machine
   75:         std::string src_operand = "";
   76:         std::string dst_operand = "";
   77:         std::string jmp_operand = "";
   78:
   79:         // Used to iterate through STRING input to search for escaped character
   80:         int string_cnt = 0;
   81:         int string_esc_cnt = 0; // Used to keep track of number of bytes saved by using ^H instead of \t (o
r similarly escaped character)
   82:
```

```
   83:         std::string last_label = "";
   84:
   85:         while(!fin.eof() && !end_flag)
   86:         {
   87:                 switch (next_state)
   88:                 {
   89:                         case CHK_FIRST_TOKEN:
   90:                                 /*      BRIEF STATE SUMMARY:
   91:                                         This state gets the first token (using fft()),
   92:                                         checks whether it in an instruction, directive,
   93:                                         or label respectively, and then sets the next
   94:                                         state to the appropriate state.
   95:                                 */
   96:
   97:                                 line_num++;
   98:
   99:                                 current_token = fft(fin);       // This fetches the next record and it's fi
rst token
  100:
  101:                                 // If there is an empty line, move on to next line
  102:                                 if(current_token == "")
  103:                                 {
  104:                                         next_state = CHK_FIRST_TOKEN;
  105:                                         break;
  106:                                 }
  107:
  108:                                 // Check if token is an instruction
  109:                                 id_ptr = get_inst_dir(current_token, I);
  110:                                 if(id_ptr != NULL)
  111:                                 {
  112:                                         next_state = INST;
  113:                                         break;
  114:                                 }
  115:
  116:                                 // Check if token is a directive
  117:                                 id_ptr = get_inst_dir(current_token, D);
  118:                                 if(id_ptr != NULL)
  119:                                 {
  120:                                         next_state = DIRECT;
  121:                                         break;
  122:                                 }
  123:
  124:                                 // Check if token is is a symbol
  125:                                 symtbl_ptr = get_symbol(current_token);
  126:                                 if(symtbl_ptr == NULL && valid_symbol(current_token))   // Therefore this i
s a new symbol
  127:                                 {
  128:                                         add_symbol(current_token, LC, KNOWN);
  129:                                         last_label = current_token;                             // Used for
 the EQU directive
  130:                                         next_state = CHK_NEXT_TOKEN;
  131:                                         break;
  132:                                 }
  133:                                 else if(symtbl_ptr != NULL)
  134:                                 {
  135:                                         if(symtbl_ptr->type == UNKNOWN)                         // Fill in
any forward references
  136:                                         {
  137:                                                 symtbl_ptr->value = LC;
  138:                                                 symtbl_ptr->type = KNOWN;
  139:                                                 symtbl_ptr->line = line_num;                    // Put in l
ine at which the label is made known
  140:                                                 last_label = symtbl_ptr->label;                 // Used for
 the EQU directive
  141:                                                 next_state = CHK_NEXT_TOKEN;
  142:                                                 break;
  143:                                         }
  144:                                         else
  145:                                         {
  146:                                                 error_detected("Chk_First_Token: Duplicate token");
  147:                                                 next_state = CHK_FIRST_TOKEN;
  148:                                                 break;
  149:                                         }
  150:                                 }
  151:                                 else
  152:                                 {
  153:                                         error_detected("Chk_First_Token: Invalid token");
  154:                                         next_state = CHK_FIRST_TOKEN;
  155:                                         break;
  156:                                 }
  157:
  158:                                 break;
  159:                         case CHK_NEXT_TOKEN:
```

```
  160:                                    /*      BRIEF STATE SUMMARY:
  161:                                            This state occurs if the first token was a valid label.
  162:                                            At this point the token is either empty, an instruction,
  163:                                            or a directive. Otherwise it is an error. This state is
  164:                                            quite similar to the previous state in functionality and
  165:                                            thus has less explanatory comments.
  166:                                    */
  167:
  168:                                    current_token = fnt();  // This fetches the next token from the current rec
ord
  169:
  170:                                    if(current_token == "")
  171:                                    {
  172:                                            next_state = CHK_FIRST_TOKEN;
  173:                                            break;
  174:                                    }
  175:
  176:                                    id_ptr = get_inst_dir(current_token, I);
  177:                                    if(id_ptr != NULL)
  178:                                    {
  179:                                            next_state = INST;
  180:                                            break;
  181:                                    }
  182:
  183:                            id_ptr = get_inst_dir(current_token, D);
  184:                            if(id_ptr != NULL)
  185:                            {
  186:                                    next_state = DIRECT;
  187:                                    break;
  188:                            }
  189:
  190:                                    // If it is not empty, an instruction, or a directive, it is an error
  191:                                    error_detected("Chk_Next_Token: Token is not empty, INST, or DIR");
  192:                            next_state = CHK_FIRST_TOKEN;
  193:
  194:                            break;
  195:                                    case INST:
  196:                                    /*      BRIEF STATE SUMMARY:
  197:                                            This state parses the current instruction from the provided
  198:                                            id_ptr (which is set in the previous state). The location counter
  199:                                            is incremented and the next state is set based on the type of instr
uction.
  200:
  201:                                            The operand for the instruction is also found and set up for the
                                                  following state.
  202:                                    */
  203:                                    LC += 2;
  204:
  205:                                    current_token = fnt();  // This fetches the next token from the current rec
ord
  206:
  207:                                    // If the next state is not set to check operands (i.e. if there is an erro
r),
  208:                                    // the next state is set to CHK_FIRST_TOKEN by default
  209:                                    next_state = CHK_FIRST_TOKEN;
  210:
  211:                                    switch(id_ptr->type)
  212:                                    {
  213:                                            case NONE:
  214:                                                    if(current_token != "") error_detected("Found Operand on NO
NE INST");
  215:
  216:                                                    break;
  217:                                            case SINGLE:
  218:                                                    if(current_token == "") error_detected("INST: Found No Oper
and on SINGLE INST");
  219:                                                    else
  220:                                                    {
  221:                                                            src_operand = current_token;
  222:                                                            next_state = CHK_SRC_OP;
  223:                                                    }
  224:                                                    break;
  225:                                            case JUMP:
  226:                                                    if(current_token == "") error_detected("INST: Found No Oper
and on JMP INST");
  227:                                                    else
  228:                                                    {
  229:                                                            jmp_operand = current_token;
  230:                                                            next_state = CHK_JMP_OP;
  231:                                                    }
  232:                                                    break;
  233:                                            case DOUBLE:
  234:                                                    two_op_flag = true;
  235:
```

```
  236:                                                    if(current_token.find_first_of(",") != std::string::npos
  237:                                                            && current_token.find_first_of(",") != current_toke
n.length()-1)
  238:                                                    {
  239:                                                            src_operand = current_token.substr(0, current_token
.find_first_of(","));
  240:                                                            dst_operand = current_token.substr(current_token.fi
nd_first_of(",")+1);
  241:                                                            next_state = CHK_DST_OP; // CHK_DST_OP sets the nex
t state to CHK_SRC_OP after
  242:                                                    }
  243:                                                    else error_detected("INST: Found Non-Double operand DOUBLE
INST");
  244:
  245:                                                    break;
  246:                                            default:
  247:                                                    std::cout << "[INST] THIS SHOULD NEVER TRIGGER" << std::end
l;
  248:                                                    getchar();  // This should never happen, getchar will stop
the runtime
  249:                                                                                    // and let the user know there is a
 serious assembler flaw
  250:                                                    break;
  251:                                    }
  252:
  253:                                    break;
  254:                            case DIRECT:
  255:                                    /*      BRIEF STATE SUMMARY:
  256:                                    This state parses the current directive from the provided id_ptr
  257:                                    (which is set in the previous state). The directives are performed
  258:                                    through a switch case that has different operations for each direct
ive.
  259:                                    Instead of writing an enumeration for the directives, I simply noti
ced
  260:                                    that all directives inthe MSB-430 set have a unique 2nd character a
nd
  261:                                    performed a switch case off it. (Note: This was discussed with a cl
assmate,
  262:                                    Tom Smith) See below for the translates.
  263:
  264:                                    > ALIGN, BSS, BYTE, END, EQU, ORG, STRING, WORD
  265:                                    >  L      S    Y     N    Q    R    T       O
  266:                                    */
  267:
  268:                                    // No matter the outcome of the directive (error or not), next state will b
e CHK_FIRST_TOKEN
  269:                                    next_state = CHK_FIRST_TOKEN;
  270:
  271:                                    // The directive is assumed to have caused an error until proven otherwise
  272:                                    directive_error_flag = true;
  273:
  274:                                    // If the type is not ALIGN or STRING, the value needs to be parsed
  275:                                    if(id_ptr->mnemonic[1] != 'L' && id_ptr->mnemonic[1] != 'N' && id_ptr->mnem
onic[1] != 'T')
  276:                                    {
  277:                                            current_token = fnt();                                          //
Find next token
  278:                                            symtbl_ptr = get_symbol(current_token);         // Check symtbl for
 that token
  279:
  280:                                            // "No forward referecing for directives" ~ TA Gary, 2017
  281:                                            if(symtbl_ptr == NULL && valid_symbol(current_token))
  282:                                            {
  283:                                                    error_detected("Directive: UNKNOWN label after DIRECTIVE (V
alue0 parsing, #1)");
  284:                                            }
  285:                                            else
  286:                                            {
  287:                                                    // The operand parser expects immediates to start with "#",
 so this is added
  288:                                                    current_token = "#" + current_token;
  289:                                                    addr_mode = parse(current_token, value0, value1);
  290:                                                    if(addr_mode == IMMEDIATE) directive_error_flag = false;
  291:                                                    else error_detected("Directive: Unknown Label after DIRECTI
VE (Value0 parsing, #2)");
  292:                                                    if(!is_last_token()) error_detected("Directive: Found token
 after DIRECTIVE");
  293:                                            }
  294:                                    }
  295:
  296:                                    switch (id_ptr->mnemonic[1])
  297:                                    {
  298:                                            case 'L':  // Align
```

```
  299:                                                    if(LC%2) LC++;
  300:                                                    if(!is_last_token()) error_detected("Directive: Found token
 after ALIGN directive");
  301:                                                    break;
  302:
  303:                                            case 'S':  // BSS
  304:                                                    if(!directive_error_flag)
  305:                                                    {
  306:                                                            // BSS cannot put LC above max LC value (0xffff)
  307:                                                            if(value0 >= 0 && value0 < MAXWORD-LC) LC += value0
;
  308:                                                            else error_detected("Directive: Invalid value for B
SS (Negative or too small)");
  309:                                                    }
  310:                                                    break;
  311:
  312:                                            case 'Y':  // BYTE
  313:                                                    if(!directive_error_flag)
  314:                                                    {
  315:                                                            if(value0 >= MINBYTE && value0 <= MAXBYTE) LC += 1;
 // Bounds for signed byte
  316:                                                            else error_detected("Directive: Value too large for
 BYTE directive");
  317:                                                    }
  318:
  319:                                                    break;
  320:
  321:                                            case 'N':  // END
  322:                                                    end_flag = true;
  323:                                                    current_token = fnt();                          // Find nex
t token
  324:                                                    symtbl_ptr = get_symbol(current_token);         // Check sy
mtbl for that token
  325:
  326:                                                    if(symtbl_ptr == NULL)
  327:                                                    {
  328:                                                            if(current_token != "") error_detected("Directive:
Invalid label after END directive (Undeclared or invalid symbol");
  329:                                                    }
  330:                                                    else if(symtbl_ptr->type != KNOWN) error_detected("Directiv
e: REG or UNKNOWN label found after END directive");
  331:                                                    if(!is_last_token()) // After the label, there must be no o
ther token
  332:                                                    {
  333:                                                            error_detected("Directive: Found Unknown token afte
r END");
  334:                                                            end_flag = false;
  335:                                                    }
  336:                                                    break;
  337:
  338:                                            case 'Q':  // EQU
  339:                                                    if(!directive_error_flag)
  340:                                                    {
  341:                                                            // EQU requires a label, therefore the 'last_label'
 is kept track of
  342:                                                            // If that symbol's line number matches the current
 line number, EQU can proceed
  343:                                                            current_token = fnt();
  344:
  345:                                                            symtbl_ptr = get_symbol(last_label);
  346:
  347:                                                            if(symtbl_ptr == NULL) error_detected("Directive: N
o label for EQU directive (Case 1)");
  348:                                                            else if(symtbl_ptr->type == KNOWN && symtbl_ptr->li
ne == line_num)
  349:                                                            {
  350:                                                                    // Therefore there is a label preceding EQU
  351:                                                                    // EQU cannot be negative, that would allow
 jumping to negative values
  352:                                                                    if(value0 >= 0 && value0 <= MAXWORD)
  353:                                                                    {
  354:                                                                            symtbl_ptr->value = value0;
  355:                                                                            symtbl_ptr->line = line_num;
  356:
  357:                                                                            if(!is_last_token()) error_detected
("Directive: Found Unknown Label after EQU value");
  358:                                                                    }
  359:                                                                    else error_detected("Directive: Value negat
ive or too large for EQU directive");
  360:                                                            }
  361:                                                            else error_detected("Directive: No label for EQU di
rective (Case 2)");
  362:                                                    }
```

```
  363:                                                    break;
  364:
  365:                                            case 'R':  // ORG
  366:                                                    if(!directive_error_flag)
  367:                                                    {
  368:                                                            // LC cannot be negative or  greater than MAXWORD a
t any point
  369:                                                            if(value0 >= 0 && value0 < MAXWORD)
  370:                                                            {
  371:                                                                    LC = value0;
  372:                                                                    if(!is_last_token()) error_detected("Direct
ive: Found Unknown Label after ORG value");
  373:                                                            }
  374:                                                            else error_detected("Directive: Value negative or t
oo large for ORG directive");
  375:                                                    }
  376:
  377:                                                    break;
  378:
  379:                                            case 'T':  // STRING
  380:                                                    current_token = fnt();
  381:                                                    // String max value has been (Arbitrarily) set to 80 charac
ters
  382:                                                    // (Punch card width), plus two for the quotation marks
  383:                                                    if(current_token.length() <= 82)
  384:                                                    {
  385:                                                            if(current_token[0] != '"') error_detected("Directi
ve: Missing OPENING quote for STRING");
  386:                                                            else
  387:                                                            {
  388:                                                                    current_token.erase(0,1); // Removes Openin
g Quote
  389:                                                                    // ITERATE THROUGH LOOKING FOR ESCAPE CHARA
CTERS
  390:                                                                    for(string_cnt = 0; string_cnt < current_to
ken.length(); string_cnt++)
  391:                                                                    {
  392:                                                                            if(current_token[string_cnt] == '\\
')
  393:                                                                            {
  394:                                                                                    string_esc_cnt++;
  395:                                                                                    string_cnt++;
  396:                                                                                    if(string_cnt == current_to
ken.length()-1) error_detected("Directive: Escaping final character of STRING");
  397:                                                                            }
  398:                                                                            else if(current_token[string_cnt] =
= '\"') // Looking for end quote
  399:                                                                            {
  400:                                                                                    // Unescaped double quote m
ust be the last characer of the token
  401:                                                                                    // meaning the string_cnt m
ust be 1 less than the string length
  402:                                                                                    if(current_token.length() -
 string_cnt != 1) error_detected("Directive: STRING error, premature quote");
  403:                                                                            }
  404:                                                                    }
  405:
  406:                                                                    current_token.pop_back(); // Removes Closin
g Quote
  407:
  408:                                                                    LC += current_token.length();
  409:                                                                    LC -= string_esc_cnt;
  410:
  411:                                                                    outfile << "ESC CNT >>" << string_esc_cnt <
<"<<"<<std::endl;
  412:
  413:                                                                    string_esc_cnt = 0;
  414:                                                                    if(!is_last_token()) error_detected("Direct
ive: Found Unknown Label after STRING value");
  415:                                                            }
  416:                                                    }
  417:                                                    else error_detected("Directive: Value too large for ORG dir
ective");
  418:
  419:                                                    break;
  420:
  421:                                            case 'O':  // WORD
  422:                                                    if(!directive_error_flag)
  423:                                                    {
  424:                                                            if(value0 >= MINWORD && value0 <= MAXWORD) LC += 2;
  425:                                                            else error_detected("Directive: Value too large for
 WORD directive");
  426:                                                    }
```

```
  427:                                                    break;
  428:
  429:                                            default:
  430:                                                    std::cout << "\t\t[Directive] DEFAULT ERROR" << std::endl;
// This should literally never happen
  431:                                                    getchar();  // This should never happen, getchar will stop
the runtime
  432:                                                                                    // and let the user know there is a
 serious assembler flaw
  433:                                                    break;
  434:                                    }
  435:
  436:                                    break;
  437:                            case CHK_SRC_OP:
  438:                                    /*      BRIEF STATE SUMMARY:
  439:                                            This state parses the source operand for either one or two
  440:                                            operand instructions. Most of the work here is done by the
  441:                                            operand parser. Note: The constant generator check is also
  442:                                            performed here, it cannot use forward references and takes
  443:                                            measures to avoid it. If the constant generator is used,
  444:                                            the location counter is not incremented for the immediate
  445:                                            addressing mode.
  446:                                    */
  447:
  448:                                    next_state = CHK_FIRST_TOKEN;
  449:
  450:                                    addr_mode = parse(src_operand, value0, value1);
  451:
  452:                                    if(addr_mode == WRONG) error_detected("CHK_SRC_OP: Invalid SRC Operand Pars
ing");
  453:                                    else
  454:                                    {
  455:                                            LC += addr_mode_LC_array[addr_mode];
  456:
  457:                                            // Constant generator check, must also avoid forward references
  458:                                            if(addr_mode == IMMEDIATE && (value0 == -1 || value0 == 0 || value0
 == 1 || value0 == 2 || value0 == 4 || value0 == 8))
  459:                                            {
  460:                                                    symtbl_ptr = get_symbol(src_operand);
  461:                                                    if(symtbl_ptr != NULL)
  462:                                                    {
  463:                                                            if(symtbl_ptr->type == UNKNOWN) break; // Breaks be
fore undoing the LC increment
  464:                                                            else LC -= 2; // Undo the LC increment from earlier
  465:                                                    }
  466:                                                    else LC -= 2; // Undo the LC increment from earlier
  467:                                            }
  468:                                            if(!is_last_token()) error_detected("Directive: Found Unknown Label
 after SRC operand");
  469:                                    }
  470:
  471:                                    src_operand = "";
  472:                                    break;
  473:                            case CHK_DST_OP:
  474:                                    /*      BRIEF STATE SUMMARY:
  475:                                            This state parses the destination operand for two operand
  476:                                            instructions. Most of the work here is done by the
  477:                                            operand parser. Only the first 4 addressing modes are valid,
  478:                                            so if the parser returns an invalid addressing mode, an error
  479:                                            is triggered.
  480:                                    */
  481:
  482:                                    next_state = CHK_FIRST_TOKEN;
  483:
  484:                                    addr_mode = parse(dst_operand, value0, value1);
  485:
  486:                                    // If the addressing mode is INDIRECT (4), INDIRECT_AI (5), IMMEDIATE (6),
or WRONG (7),
  487:                                    //  there is an error. See enumerations in library.h for declaration that s
hows this
  488:                                    if(addr_mode >= 4) error_detected("CHK_DST_OP: Invalid addressing mode or p
arsing for DST operand");
  489:                                    else
  490:                                    {
  491:                                            LC += addr_mode_LC_array[addr_mode];
  492:                                            // Must ensure that this is the last token in the record
  493:                                            dst_operand = fnt();
  494:                                            if(dst_operand == "") next_state = (two_op_flag) ? CHK_SRC_OP : CHK
_NEXT_TOKEN;
  495:                                            else error_detected("CHK_DST_OP: Invalid extra token after operand
token");
  496:                                    }
  497:                                    dst_operand = "";
```

```
  498:                            break;
  499:                          case CHK_JMP_OP:
  500:                            /*      BRIEF STATE SUMMARY:
  501:                                    This state parses the jump operand for jump instructions.
  502:                                    Again, most of the work here is done by the operand
  503:                                    parser. The only addressing mode allowed by jump instructions
  504:                                    is relative.
  505:                            */
  506:
  507:                            next_state = CHK_FIRST_TOKEN;
  508:
  509:                            addr_mode = parse(jmp_operand, value0, value1);
  510:
  511:                            // JUMP instructions must have the relative addressing mode1
  512:                            if(addr_mode == RELATIVE)
  513:                            {
  514:                                    LC += addr_mode_LC_array[addr_mode];
  515:                                    if(!is_last_token()) error_detected("Directive: Found Unknown Label
 after JMP operand");
  516:                            }
  517:                            else error_detected("CHK_JMP_OP: Invalid addressing mode or parsing for JMP
 operand");
  518:
  519:                            jmp_operand = "";
  520:                            break;
  521:                          default:
  522:                            std::cout << "\t\t[First Pass] DEFAULT ERROR" << std::endl;
  523:                            getchar();  // This should never happen, getchar will stop the runtime
  524:                                                    // and let the user know there is a serious assembl
er flaw
  525:                            break;
  526:                  }
  527:          }
  528:          line_num = 0;
  529: }
  530:
  531: /*
  532:          Function: error_detected
  533:          Output: bool: True means this is the last token on the line, false means this is not the last token
  534:          Brief: Fetches next token and checks if it is empty. Returns true or false accordingly.
  535: */
  536: bool is_last_token()
  537: {
  538:          current_token = fnt();
  539:          if(current_token != "") return false;
  540:          else return true;
  541: }
  542:
  543: /*
  544:          Function: error_detected
  545:          Input: error_msg: String containing the error message to print out to diagnostics
  546:          Brief: Increments the error counter and performs diagnostics output
  547: */
  548: void error_detected(std::string error_msg)
  549: {
  550:          std::cout << std::dec << "\tRECORD #" << line_num << ": >>"<< current_record << "<<" <<std::endl;
  551:          std::cout << "\t\t[ERROR MSG - FIRST PASS] " << error_msg << std::endl << std::endl;
  552:
  553:          outfile << std::dec << "\tRECORD #" << line_num << ": >>"<< current_record << "<<" <<std::endl;
  554:          outfile << "\t\t[ERROR MSG - FIRST PASS] " << error_msg << std::endl << std::endl;
  555:
  556:          err_cnt++;
  557: }
```