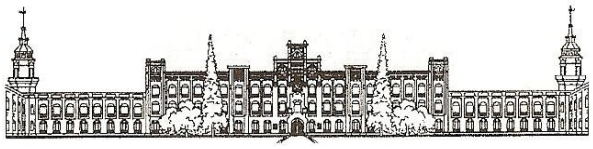

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ ТА СПЕЦІАЛІЗОВАНИХ КОМП’ЮТЕРНИХ СИСТЕМ

Лабораторна робота №3

з дисципліни «Архітектура для програмістів»

Виконав студент групи: КВ-22

ПІБ: Крутогуз Максим Ігорович

Перевірив: Молчанов О. А.

Київ 2025

Мета лабораторної роботи наступна:

- ознайомитись із елементами рівня архітектури системи команд;
- отримати практичний досвід декодування машинних команд.

Загальне завдання

Завдання на лабораторну роботу. Реалізувати програму мовою C або C++, що виконує: зчитування послідовності машинних команд (програми), що визначаються варіантом, з файлу; розбір зчитаних команд і вивід їхнього текстового представлення (асемблерного коду). Крім того необхідно реалізувати модульні тести для реалізованого(-их) модуля(-ів) програми.

Програма має містити наступні компоненти (модулі):

1. Модуль з реалізацією функцій зчитування і аналізу (розбору) машинних команд з текстового файлу.
2. Модуль тестування, що містить тестові утиліти і тести реалізованої програми.

Посилання:

- Github

12	1, 2, 3, 4, 5, 24, 25, 26, 28

1	MOV <reg1>, <reg2>	1A /reg1 /reg2 (приклад: 1A 12)	перемістити значення з регістру <reg1> у регістр <reg2>
2	MOV <reg>, <addr>	1B 0 /reg /addr (приклад: 1B 01 00000042)	перемістити значення з ОП за адресою <addr> у регістр <reg>
3	MOV <addr>, <reg>	1B 1 /reg /addr (приклад: 1B 11 00000042)	перемістити значення з регістру <reg> в ОП за адресою <addr>
4	ADD <reg1>, <reg2>	01 /reg1 /reg2 (приклад: 01 21)	додавання значення з регістру <reg1> до значення з регістру <reg2> і збереження результату в регістрі <reg1>
5	ADD <reg>, <addr>	02 0 /reg /addr (приклад: 02 01 00000042)	додавання значення з регістру <reg> до 4-байтового значення з ОП за адресою <addr> і збереження результату в регістрі <reg>

24	JG <shift>	94 /shift (приклад: 94 FA)	перехід за 1-байтовим відносним зміщенням <shift> у випадку, якщо ZF = 0 і SF = OF
25	JG <addr>	95 /addr (приклад: 95 00000042)	перехід за 4-байтовою адресою <addr> у випадку, якщо ZF = 0 і SF = OF
26	CMP <reg1>, <reg2>	80 /reg1 /reg2 (приклад: 80 12)	порівняння двох значень і встановлення відповідних прапорців

28	MOV <reg>, <lit16>	1C 1 /reg /lit16 (приклад: 1C 11 3344)	переміщення 2-байтового числа у регістра <reg>

Тестування

В режимі ручного тестування через файли:

```
1A 77
1B 01 00000042
1B 12 00000042

01 32
02 04 00aabb80

95 00004a5d
94 ea

80 12
1C 81 0020
```

```
1    1A 77             MOV R7, R7
2    1B 01 00000042    MOV R1, [0x00000042]
3    1B 12 00000042    MOV [0x00000042], R2
4    01 32             ADD R3, R2
5    02 04 00AABB80    ADD R4, [0x00AABB80]
6    95 5D 00004A5D    JG [0x00004A5D]
7    94 EA             JG -22
8    80 12             CMP R1, R2
9    1C 81 0020        MOV R1, 32
10
```

Модульне тестування із використанням рядків (файл test.cpp внизу або в репозиторії):

```
PS J:\Repositories\University\Computer_architecture\Lab3> make runtest
Compiling tests...
g++ -Wall -Wextra -std=c++17 src/ErrorManager.cpp src/Global.cpp src/Lexer.cpp src/Parser.cpp tests/test.cpp -o bin/TestApp.exe
bin/TestApp.exe
===============================================================================
All tests passed (79 assertions in 26 test cases)
```

Лістинг програми:

```cpp
// App.cpp
#include <iostream>
#include <sstream>
#include <string>

#include "Global.hpp"
#include "Lexer.hpp"
#include "Parser.hpp"

using namespace std;

int main() {
    string parserName = "Main";
    // Lexer lexer(lexerName, stringstream("1A\np2\t1A  12"));
    // Lexer lexer(lexerName, fstream("input1.txt"));

    // cout << lexer.nextToken().value << endl;
    // cout << lexer.nextToken().value << endl;
    // cout << lexer.nextToken().value << endl;
    // cout << lexer.nextToken().value << endl;

    // errorManager.outputLexicalErrors(lexerName);

    fstream input("input1.txt");
    fstream output("output1.txt", ios::out);
```

```
    Parser parser(parserName, move(output), move(input));

    parser.parse(CODE_ASM);

    errorManager.outputLexicalErrors(parserName);
    cout << endl;
    errorManager.outputSyntaxErrors(parserName);

    return 0;
}


// ErrorManager.cpp
#include "ErrorManager.hpp"

void ErrorManager::addLexicalError(int row, int column, const string& message,
string& lexerName) {
    _lexicalErrors[lexerName].push_back(Error({message, row, column}));
}

void ErrorManager::addSyntaxError(int row, int column, const string& message,
string& lexerName) {
    _syntaxErrors[lexerName].push_back(Error({message, row, column}));
}

int ErrorManager::getLexicalErrorCount(string& lexerName) {
    return _lexicalErrors[lexerName].size();
}

int ErrorManager::getSyntaxErrorCount(string& lexerName) {
    return _syntaxErrors[lexerName].size();
}

void ErrorManager::outputLexicalErrors(string lexerName) {
    auto lexicalErr = _lexicalErrors[lexerName];

    if (lexicalErr.size() == 0) {
        cout << "There are no lexical errors for " << lexerName << " lexer" <<
endl;
        return;
    }

    cout << "There are lexical errors for " << lexerName << " lexer:" << endl;
    for (auto err : lexicalErr) {
        cout << "\t" << err.row << ':' << err.col << ": " << err.message <<
endl;
    }
}

void ErrorManager::outputSyntaxErrors(string lexerName) {
    auto syntaxErr = _syntaxErrors[lexerName];

    if (syntaxErr.size() == 0) {
        cout << "There are no syntax errors for " << lexerName << " parser" <<
endl;
        return;
    }

    cout << "There are syntax errors for " << lexerName << " parser:" << endl;
    for (auto err : syntaxErr) {
```

```
        cout << "\t" << err.row << ':' << err.col << ": " << err.message <<
endl;
    }
}


// ErrorManager.hpp
#ifndef ERRORMANAGER_HPP
#define ERRORMANAGER_HPP

#include <iostream>
#include <vector>
#include <map>

using namespace std;

typedef struct {
    string message;
    int row;
    int col;
} Error;

class ErrorManager {
    public:
        void addLexicalError(int row, int column, const string& message, string&
lexerName);
        void addSyntaxError(int row, int column, const string& message, string&
lexerName);

        int getLexicalErrorCount(string& lexerName);
        int getSyntaxErrorCount(string& lexerName);

        void outputLexicalErrors(string lexerName);
        void outputSyntaxErrors(string lexerName);

    private:
        map<string, vector<Error>> _lexicalErrors;
        map<string, vector<Error>> _syntaxErrors;

};
#endif


// Global.cpp
#include "Global.hpp"

#include <sstream>
#include <iomanip>

int symbol_categories[127] = {
    PROHIBITED_CHARACTER, PROHIBITED_CHARACTER, PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER, PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    WHITESPACE,
    WHITESPACE,
    WHITESPACE,
    WHITESPACE,
    WHITESPACE,
```

```
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    WHITESPACE,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
```

```
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    HEX_DIGIT,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
};

ErrorManager errorManager;

unordered_map<int, string> registerMap = {
    {0b000, "R0"},
    {0b001, "R1"},
    {0b010, "R2"},
    {0b011, "R3"},
```

```
    {0b100, "R4"},
    {0b101, "R5"},
    {0b110, "R6"},
    {0b111, "R7"}

};

unordered_map<string, int> machineCodeMap = {
    {"1A", MOVREGREG},
    {"1B", MOVADDR},
    {"1C", MOVREGLIT16},
    {"01", ADDREGREG},
    {"02", ADDREGADDR},
    {"95", JGADDR},
    {"94", JGSHIFT},
    {"80", CMP}
};

string mapRegister(int value) {
    if (registerMap.count(value) > 0) {
        return registerMap[value];
    }

    return "";
}

int mapMachineCode(string value) {
    if (machineCodeMap.count(value) > 0) {
        return machineCodeMap[value];
    }

    return PROHIBITED;
}


string mapAddress(string hexAddress) {
    if (hexAddress.size() != 8) {
        return "";
    }

    stringstream ss;

    ss << "[0x" << hexAddress << ']';

    return ss.str();
}

string mapValue(string hexValue) {
    if (hexValue.size() != 4) {
        return "";
    }

    int value;
    stringstream ss;

    ss << hex << hexValue;

    ss >> value;

    return to_string(value);
}
```

```
string mapShift(string shiftValue) {
    if (shiftValue.size() != 2) {
        return "";
    }

    stringstream ss;

    ss << hex << shiftValue;

    int value;

    ss >> value;

    if (value > 127) {
        value = -(256 - value);
    }

    return to_string(value);
}


// Global.hpp
#ifndef GLOBAL_HPP
#define GLOBAL_HPP

#include <string>
#include <unordered_map>

#include "ErrorManager.hpp"

using namespace std;

typedef struct {
    string value;
    int row;
    int column;
} Token;

enum SymbolCategory {
    WHITESPACE,
    HEX_DIGIT,
    PROHIBITED_CHARACTER,
};

enum CommandNames {
    MOVREGREG,
    MOVADDR,
    MOVREGADDR,
    MOVADDRREG,
    MOVREGLIT16,
    ADDREGREG,
    ADDREGADDR,
    JGADDR,
    JGSHIFT,
    CMP,
    PROHIBITED
};

enum ParseType {
    ASM,
```

```
    CODE_ASM
};

extern int symbol_categories[127];

extern ErrorManager errorManager;

string mapRegister(int value);

int mapMachineCode(string value);

string mapAddress(string hexAddress);

string mapValue(string hexValue);

string mapShift(string shiftValue);

#endif


// Lexer.cpp
#include "Lexer.hpp"
#include <fstream>

Lexer::Lexer(string lexerName, fstream&& fs) : _fs(new fstream(move(fs))),
_ss(nullptr), _row(1), _column(1), _lexerName(lexerName) {}

Lexer::Lexer(string lexerName, stringstream&& ss) : _fs(nullptr), _ss(new
stringstream(move(ss))), _row(1), _column(1), _lexerName(lexerName) {}

Lexer::~Lexer() {
    if (_fs) {
        _fs->close();
        delete _fs;
    }
    if (_ss) {
        _ss->clear();
        delete _ss;
    }
}


Token Lexer::nextToken() {
    int lexicalSymbol = 0;
    char symbol;
    stringstream ss;

    while (lexicalSymbol < 2) {
        symbol = nextSymbol();
        if (symbol == '\0') {
            if (lexicalSymbol == 0) {
                return Token{"", _row, _column};
            } else {
                errorManager.addLexicalError(_row, _column + lexicalSymbol,
string("Unexpected end"), _lexerName);
                break;
            }
        }
        switch (symbol_categories[(short)symbol]) {
            case HEX_DIGIT:
                ss << (char)toupper(symbol);
```

```
                lexicalSymbol++;
                _column += 1;
                break;
            case WHITESPACE:
                // cout << "Whitespace" << endl;
                // cout << (int)symbol << endl;
                if (symbol == '\n') {
                    // cout << _row << endl;
                    _row += 1;
                    // cout << _row << endl;
                    _column = 1;
                } else if (symbol == '\t') {
                    _column += 4;
                } else {
                    _column++;
                }
                break;
            // case PROHIBITED_CHARACTER:
            //     if (symbol == '\0') {
            //         return Token{"", _row, _column};
            //     } else {
            //         Token token = {ss.str(), _row, _column};
            //         ss.str("");
            //         return token;
            //     }
            // default:
            //     if (symbol == '\0') {
            //         return Token{"", _row, _column};
            //     } else {
            //         ss << symbol;
            //         _column++;
            //         symbol = nextSymbol();
            //     }
            //     break;
            default:
                string str = ss.str();
                errorManager.addLexicalError(_row, _column - (int)str.size(),
string("Used prohibitied character: ") + symbol, _lexerName);
                return Token{"", _row, _column - (int)str.size()};
        }
    }

    return Token{ss.str(), _row, _column - 2};
}


//Private

char Lexer::nextSymbol() {
    char symbol = '\0';
    if (_fs) {
        _fs->get(symbol);
        if (_fs->fail()) {
            return '\0';
        } else if (symbol == EOF) {
            return '\0';
        }
    } else if (_ss) {
        _ss->get(symbol);
        if (_ss->fail()) {
            return '\0';
```

```
        }
    }
    return symbol;
}


// Lexer.hpp
#ifndef LEXER_HPP
#define LEXER_HPP

#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

#include "Global.hpp"

using namespace std;

class Lexer {
    public:
        Lexer(string lexerName, fstream&& fs);
        Lexer(string lexerName, stringstream&& ss);
        ~Lexer();

        Token nextToken();

    private:
        fstream* _fs;
        stringstream* _ss;

        int _row;
        int _column;

        string _lexerName;

        char nextSymbol();
};

#endif


// Parser.cpp
#include "Parser.hpp"

Parser::Parser(string parserName, stringstream &&ssout, stringstream &&ssin) :
    _parserName(parserName),
    _fsout(nullptr),
    _ssout(new stringstream(move(ssout))),
    _lexer(new Lexer(parserName, move(ssin))) {
}

Parser::Parser(string parserName, fstream &&fsout, fstream &&fsin) :
    _parserName(parserName),
    _fsout(new fstream(move(fsout))),
    _ssout(nullptr),
    _lexer(new Lexer(parserName, move(fsin))) {
}

Parser::~Parser() {
    if (_fsout) {
```

```
        _fsout->close();
        delete _fsout;
    }
    if (_ssout) {
        _ssout->clear();
        delete _ssout;
    }
}

string Parser::getssout() {
    if (_ssout) {
        return _ssout->str();
    }
    return "";
}

void Parser::printLine(int parseType) {
    if (_fsout) {
        if (parseType == ASM) {
            *_fsout << _assemblyCodeStream.str() << endl;
        } else {
            *_fsout << setw(18) << left << _machineCodeStream.str() <<
_assemblyCodeStream.str() << endl;
        }
    }
    if (_ssout) {
        if (parseType == ASM) {
            *_ssout << _assemblyCodeStream.str() << endl;
        } else {
            *_ssout << setw(18) << left << _machineCodeStream.str() <<
_assemblyCodeStream.str() << endl;
        }
    }
    _assemblyCodeStream.str("");
    _machineCodeStream.str("");
}

void Parser::parse(int parseType) {
    _currentToken = _lexer->nextToken();

    while (_currentToken.value != "" &&
errorManager.getSyntaxErrorCount(_parserName) == 0 &&
errorManager.getLexicalErrorCount(_parserName) == 0) {
        int prevTokenValue = mapMachineCode(_currentToken.value);
        if (prevTokenValue != PROHIBITED) {
            _machineCodeStream << _currentToken.value;
            _currentToken = _lexer->nextToken();
        }

        switch (prevTokenValue) {
            case MOVREGREG:
                _assemblyCodeStream << "MOV";
                parseMovRegReg();
                break;
            case MOVADDR:
                _assemblyCodeStream << "MOV";
                parseMovAddr();
                break;
            case ADDREGREG:
                _assemblyCodeStream << "ADD";
                parseAddRegReg();
```

```
                break;
            case ADDREGADDR:
                _assemblyCodeStream << "ADD";
                parseAddRegAddr();
                break;
            case JGADDR:
                _assemblyCodeStream << "JG";
                parseJgAddr();
                break;
            case JGSHIFT:
                _assemblyCodeStream << "JG";
                parseJgShift();
                break;
            case CMP:
                _assemblyCodeStream << "CMP";
                parseCmp();
                break;
            case MOVREGLIT16:
                _assemblyCodeStream << "MOV";
                parseMovRegLit16();
                break;
            default:
                // cout << _currentToken.value << endl;
                errorManager.addSyntaxError(_currentToken.row,
_currentToken.column, "Prohibited command", _parserName);
                _machineCodeStream.str("");
                _assemblyCodeStream.str("");
                break;
        }
        if (errorManager.getSyntaxErrorCount(_parserName) == 0) {
            printLine(parseType);
        }
    }
}

void Parser::parseMovRegReg() {
    stringstream ss;
    ss << hex << _currentToken.value;

    int value;
    ss >> value;

    int reg1 = (value & 0b11110000) >> 4;
    int reg2 = value & 0b00001111;

    string regStr1 = mapRegister(reg1);
    string regStr2 = mapRegister(reg2);

    if (regStr1 == "" || regStr2 == "") {
        errorManager.addSyntaxError(_currentToken.row, _currentToken.column,
"Invalid register", _parserName);
        return;
    }

    _machineCodeStream << ' ' << _currentToken.value;
    _assemblyCodeStream << ' ' << regStr1 << ", " << regStr2;

    _currentToken = _lexer->nextToken();
}

void Parser::parseMovAddr() {
```

```
    stringstream ss;
    ss << hex << _currentToken.value;

    int value;
    ss >> value;

    bool isAddressFirst = false;

    int reg = value & 0b00001111;

    if ((value & 0b00010000) != 0) {
        isAddressFirst = true;
    }

    string regStr = mapRegister(reg);

    if (regStr == "") {
        errorManager.addSyntaxError(_currentToken.row, _currentToken.column,
"Invalid register", _parserName);
        return;
    } else if ((value & 0b11100000) != 0) {
        errorManager.addSyntaxError(_currentToken.row, _currentToken.column,
"Invalid code expression", _parserName);
        return;
    }

    _machineCodeStream << ' ' << _currentToken.value << ' ';
    _assemblyCodeStream << ' ';

    int row = _currentToken.row;
    int col = _currentToken.column;

    stringstream address;

    for (int i = 0; i < 4; i++) {
        _currentToken = _lexer->nextToken();
        address << _currentToken.value;
    }

    string addressString = mapAddress(address.str());

    if (addressString == "") {
        errorManager.addSyntaxError(row, col, "Invalid address", _parserName);
        return;
    }

    if (isAddressFirst) {
        _assemblyCodeStream << addressString << ", " << regStr;
        _machineCodeStream << address.str();
    } else {
        _assemblyCodeStream << regStr << ", " << addressString;
        _machineCodeStream << address.str();
    }

    _currentToken = _lexer->nextToken();
}

void Parser::parseAddRegReg() {
    stringstream ss;
    ss << hex << _currentToken.value;
```

```
    int value;
    ss >> value;

    int reg1 = (value & 0b11110000) >> 4;
    int reg2 = value & 0b00001111;

    string regStr1 = mapRegister(reg1);
    string regStr2 = mapRegister(reg2);

    if (regStr1 == "" || regStr2 == "") {
        errorManager.addSyntaxError(_currentToken.row, _currentToken.column,
"Invalid register", _parserName);
        return;
    }

    _machineCodeStream << ' ' << _currentToken.value;
    _assemblyCodeStream << ' ' << regStr1 << ", " << regStr2;

    _currentToken = _lexer->nextToken();
}

void Parser::parseAddRegAddr() {
    stringstream ss;
    ss << hex << _currentToken.value;

    int value;
    ss >> value;

    int reg = value & 0b00001111;

    string regStr = mapRegister(reg);

    if (regStr == "") {
        errorManager.addSyntaxError(_currentToken.row, _currentToken.column,
"Invalid register", _parserName);
        return;
    } else if ((value & 0b11110000) != 0) {
        errorManager.addSyntaxError(_currentToken.row, _currentToken.column,
"Invalid code expression", _parserName);
        return;
    }

    _machineCodeStream << ' ' << _currentToken.value << ' ';
    _assemblyCodeStream << ' ';

    int row = _currentToken.row;
    int col = _currentToken.column;

    stringstream address;

    for (int i = 0; i < 4; i++) {
        _currentToken = _lexer->nextToken();
        address << _currentToken.value;
    }

    string addressString = mapAddress(address.str());

    if (addressString == "") {
        errorManager.addSyntaxError(row, col, "Invalid address", _parserName);
        return;
    }
```

```
    _assemblyCodeStream << regStr << ", " << addressString;
    _machineCodeStream << address.str();

    _currentToken = _lexer->nextToken();
}

void Parser::parseJgAddr() {
    int row = _currentToken.row;
    int col = _currentToken.column;

    stringstream address;

    address << _currentToken.value;
    for (int i = 0; i < 3; i++) {
        _currentToken = _lexer->nextToken();
        address << _currentToken.value;
    }

    string addressString = mapAddress(address.str());

    if (addressString == "") {
        errorManager.addSyntaxError(row, col, "Invalid address", _parserName);
        return;
    }

    _machineCodeStream << ' ' << _currentToken.value << ' ' << address.str();
    _assemblyCodeStream << ' ' << addressString;

    _currentToken = _lexer->nextToken();
}

void Parser::parseJgShift() {
    string shiftStr = mapShift(_currentToken.value);

    if (shiftStr == "") {
        errorManager.addSyntaxError(_currentToken.row, _currentToken.column,
"Invalid shift", _parserName);
        return;
    }

    _machineCodeStream << ' ' << _currentToken.value;
    _assemblyCodeStream << ' ' << shiftStr;

    _currentToken = _lexer->nextToken();
}

void Parser::parseCmp() {
    stringstream ss;
    ss << hex << _currentToken.value;

    int value;
    ss >> value;

    int reg1 = (value & 0b11110000) >> 4;
    int reg2 = value & 0b00001111;

    string regStr1 = mapRegister(reg1);
    string regStr2 = mapRegister(reg2);

    if (regStr1 == "" || regStr2 == "") {
```

```
        errorManager.addSyntaxError(_currentToken.row, _currentToken.column,
"Invalid register", _parserName);
        return;
    }

    _machineCodeStream << ' ' << _currentToken.value;
    _assemblyCodeStream << ' ' << regStr1 << ", " << regStr2;

    _currentToken = _lexer->nextToken();
}

void Parser::parseMovRegLit16() {
    stringstream ss;
    ss << hex << _currentToken.value;

    int value;
    ss >> value;

    int reg = value & 0b00001111;

    string regStr = mapRegister(reg);

    if (regStr == "") {
        errorManager.addSyntaxError(_currentToken.row, _currentToken.column,
"Invalid register", _parserName);
        return;
    } else if ((value & 0b11110000) != 0b10000000) {
        errorManager.addSyntaxError(_currentToken.row, _currentToken.column,
"Invalid code expression", _parserName);
        return;
    }

    _machineCodeStream << ' ' << _currentToken.value << ' ';
    _assemblyCodeStream << ' ';

    int row = _currentToken.row;
    int col = _currentToken.column;

    stringstream lit16;

    for (int i = 0; i < 2; i++) {
        _currentToken = _lexer->nextToken();
        lit16 << _currentToken.value;
    }

    string lit16Str = mapValue(lit16.str());

    if (lit16Str == "") {
        errorManager.addSyntaxError(row, col, "Invalid imdediate value",
_parserName);
        return;
    }

    _assemblyCodeStream << regStr << ", " << lit16Str;
    _machineCodeStream << lit16.str();

    _currentToken = _lexer->nextToken();
}


// Parser.hpp
```

```cpp
#ifndef PARSER_HPP
#define PARSER_HPP

#include <fstream>
#include <sstream>
#include <map>
#include <iomanip>

#include "Global.hpp"
#include "Lexer.hpp"

using namespace std;

class Parser {
    public:
    Parser(string parserName, stringstream &&ssout, stringstream &&ssin);
    Parser(string parserName, fstream &&fsout, fstream &&fsin);
    ~Parser();

    void parse(int parseType);

    string getssout();
    // string getfsout();

    private:
    string _parserName;
    fstream* _fsout;
    stringstream* _ssout;
    Lexer *_lexer;
    Token _currentToken;
    stringstream _machineCodeStream;
    stringstream _assemblyCodeStream;

    void printLine(int parseType);

    void parseMovRegReg();
    void parseMovAddr();
    void parseMovRegLit16();

    void parseAddRegReg();
    void parseAddRegAddr();

    void parseJgAddr();
    void parseJgShift();

    void parseCmp();
};

#endif


// test.cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

#include <sstream>

#include "../src/Global.hpp"
#include "../src/Lexer.hpp"
#include "../src/Parser.hpp"
```

```
using namespace std;

// TEST_CASE("Lexer nextToken true test") {
//     Lexer lexer(stringstream("1A\n12\t1A  12"));

//     SECTION("First token") {
//         Token token = lexer.nextToken();
//         REQUIRE(token.value == string("1A"));
//         REQUIRE(token.row == 1);
//         REQUIRE(token.column == 1);
//     }
//     SECTION("Second token") {
//         Token token = lexer.nextToken();
//         REQUIRE(token.value == "12");
//         REQUIRE(token.row == 2);
//         REQUIRE(token.column == 1);
//     }
//     SECTION("Third token") {
//         Token token = lexer.nextToken();
//         REQUIRE(token.value == "1A");
//         REQUIRE(token.row == 2);
//         REQUIRE(token.column == 7);
//     }
//     SECTION("Fourth token") {
//         Token token = lexer.nextToken();
//         REQUIRE(token.value == "12");
//         REQUIRE(token.row == 2);
//         REQUIRE(token.column == 11);
//     }
// }

TEST_CASE("Symbol category test") {

    REQUIRE(symbol_categories['\t'] == WHITESPACE);
    REQUIRE(symbol_categories['\n'] == WHITESPACE);
    REQUIRE(symbol_categories[11] == WHITESPACE);
    REQUIRE(symbol_categories[12] == WHITESPACE);
    REQUIRE(symbol_categories[13] == WHITESPACE);
    REQUIRE(symbol_categories[' '] == WHITESPACE);

    REQUIRE(symbol_categories['0'] == HEX_DIGIT);
    REQUIRE(symbol_categories['1'] == HEX_DIGIT);
    REQUIRE(symbol_categories['2'] == HEX_DIGIT);
    REQUIRE(symbol_categories['3'] == HEX_DIGIT);
    REQUIRE(symbol_categories['4'] == HEX_DIGIT);
    REQUIRE(symbol_categories['5'] == HEX_DIGIT);
    REQUIRE(symbol_categories['6'] == HEX_DIGIT);
    REQUIRE(symbol_categories['7'] == HEX_DIGIT);
    REQUIRE(symbol_categories['8'] == HEX_DIGIT);
    REQUIRE(symbol_categories['9'] == HEX_DIGIT);
    REQUIRE(symbol_categories['A'] == HEX_DIGIT);
    REQUIRE(symbol_categories['B'] == HEX_DIGIT);
    REQUIRE(symbol_categories['C'] == HEX_DIGIT);
    REQUIRE(symbol_categories['D'] == HEX_DIGIT);
    REQUIRE(symbol_categories['E'] == HEX_DIGIT);
    REQUIRE(symbol_categories['F'] == HEX_DIGIT);
    REQUIRE(symbol_categories['a'] == HEX_DIGIT);
    REQUIRE(symbol_categories['b'] == HEX_DIGIT);
    REQUIRE(symbol_categories['c'] == HEX_DIGIT);
    REQUIRE(symbol_categories['d'] == HEX_DIGIT);
    REQUIRE(symbol_categories['e'] == HEX_DIGIT);
```

```
    REQUIRE(symbol_categories['f'] == HEX_DIGIT);
}

TEST_CASE("Error manager test") {

    string lexerName = "test1";

    REQUIRE(errorManager.getLexicalErrorCount(lexerName) == 0);
    errorManager.addLexicalError(1, 2, "Something wrong", lexerName);
    REQUIRE(errorManager.getLexicalErrorCount(lexerName) == 1);

    REQUIRE(errorManager.getSyntaxErrorCount(lexerName) == 0);
    errorManager.addSyntaxError(1, 2, "Something wrong again", lexerName);
    REQUIRE(errorManager.getSyntaxErrorCount(lexerName) == 1);
}

TEST_CASE("Register map test") {
    REQUIRE(mapRegister(0b001) == "R1");
    REQUIRE(mapRegister(0b101) == "R5");
    REQUIRE(mapRegister(0b1001) == "");
}

TEST_CASE("Machine code test") {
    REQUIRE(mapMachineCode("01") == ADDREGREG);
    REQUIRE(mapMachineCode("10") == PROHIBITED);
}


TEST_CASE("Address map test") {
    REQUIRE(mapAddress("00AD0043") == "[0x00AD0043]");
    REQUIRE(mapAddress("00AD00") == "");
}

TEST_CASE("Value map test") {
    REQUIRE(mapValue("3344") == "13124");
    REQUIRE(mapValue("334") == "");
}

TEST_CASE("Shift map test") {
    REQUIRE(mapShift("FA") == "-6");
    REQUIRE(mapShift("0A") == "10");
    REQUIRE(mapShift("FAA") == "");
}


TEST_CASE("Lexer token sequence true test") {
    string lexerName = "test2";
    Lexer lexer(lexerName, stringstream("1A\n12\t1A  12"));

    // First token
    Token token1 = lexer.nextToken();
    REQUIRE(token1.value == "1A");
    REQUIRE(token1.row == 1);
    REQUIRE(token1.column == 1);

    // Second token
    Token token2 = lexer.nextToken();
    REQUIRE(token2.value == "12");
    REQUIRE(token2.row == 2);
    REQUIRE(token2.column == 1);
```

```
    // Third token
    Token token3 = lexer.nextToken();
    REQUIRE(token3.value == "1A");
    REQUIRE(token3.row == 2);
    REQUIRE(token3.column == 7);

    // Fourth token
    Token token4 = lexer.nextToken();
    REQUIRE(token4.value == "12");
    REQUIRE(token4.row == 2);
    REQUIRE(token4.column == 11);
}

TEST_CASE("Lexer token file test") {
    string lexerName = "file1";
    Lexer lexer(lexerName, fstream("input1.txt"));
    REQUIRE(lexer.nextToken().value == "1A");
}

TEST_CASE("Lexer token sequence false test") {
    string lexerName = "test3";
    Lexer lexer(lexerName, stringstream("1A\n1G\t1A  12"));

    // First token
    Token token1 = lexer.nextToken();
    REQUIRE(token1.value == "1A");
    REQUIRE(token1.row == 1);
    REQUIRE(token1.column == 1);

    // Second token
    Token token2 = lexer.nextToken();
    REQUIRE(token2.value == "");
    REQUIRE(token2.row == 2);
    REQUIRE(token2.column == 1);
}

TEST_CASE("Paser prohibited command") {
    string lexerName = "test4";

    stringstream input("13\t77");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(errorManager.getSyntaxErrorCount(lexerName) == 1);
}

TEST_CASE("Parser phohibited register") {
    string lexerName = "test5";

    stringstream input("1A\nF1");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(errorManager.getSyntaxErrorCount(lexerName) == 1);
}
```

```
TEST_CASE("Parser MovRegReg true test1") {
    string lexerName = "test6";

    stringstream input("1A\t77");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(parser.getssout() == "MOV R7, R7\n");
}

TEST_CASE("Parser MovRegAddr true test1") {
    string lexerName = "test7";

    stringstream input("1B  02 00010432");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(parser.getssout() == "MOV R2, [0x00010432]\n");
}

TEST_CASE("Parser MovRegAddr false test1") {
    string lexerName = "test8";

    stringstream input("1B  22 00010432");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(errorManager.getSyntaxErrorCount(lexerName) == 1);
}

TEST_CASE("Parser AddRegReg true test1") {
    string lexerName = "test9";

    stringstream input("01 72");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(parser.getssout() == "ADD R7, R2\n");
}

TEST_CASE("Parser AddRegAddr true test1") {
    string lexerName = "test10";

    stringstream input("02 01 00001234");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));
```

```
    parser.parse(ASM);

    REQUIRE(parser.getssout() == "ADD R1, [0x00001234]\n");
}

TEST_CASE("Parser AddRegAddr false test1") {
    string lexerName = "test11";

    stringstream input("02 81 00001234");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(errorManager.getSyntaxErrorCount(lexerName) == 1);
}

TEST_CASE("Parser JgAddr true test1") {
    string lexerName = "test12";

    stringstream input("95 ff00aabb");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(parser.getssout() == "JG [0xFF00AABB]\n");
}

TEST_CASE("Parser JgAddr false test1") {
    string lexerName = "test13";

    stringstream input("95 ff00aab");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(errorManager.getSyntaxErrorCount(lexerName) == 1);
}

TEST_CASE("Parser JgShift true test1") {
    string lexerName = "test14";

    stringstream input("94 BA");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(parser.getssout() == "JG -70\n");
}

TEST_CASE("Parser JgShift false test1") {
    string lexerName = "test15";

    stringstream input("95 B");
```

```
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(errorManager.getSyntaxErrorCount(lexerName) == 1);
}

TEST_CASE("Parser Cmp true test1") {
    string lexerName = "test16";

    stringstream input("80 12");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(parser.getssout() == "CMP R1, R2\n");
}

TEST_CASE("Parser Cmp false test1") {
    string lexerName = "test17";

    stringstream input("80 82");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(errorManager.getSyntaxErrorCount(lexerName) == 1);
}

TEST_CASE("Parser MovRegLit16 true test1") {
    string lexerName = "test18";

    stringstream input("1c 86 0011");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(parser.getssout() == "MOV R6, 17\n");
}

TEST_CASE("Parser MovRegLit16 false test1") {
    string lexerName = "test19";

    stringstream input("1c 96 0011");
    stringstream output("");

    Parser parser(lexerName, move(output), move(input));

    parser.parse(ASM);

    REQUIRE(errorManager.getSyntaxErrorCount(lexerName) == 1);
}
```