**Name: Paritosh Kabra**

**Enrollment Number: 20114066**

**Subbatch: O3**

**SICXE Assembler**

**Q1: How to start the assembler?**

**Ans:** Download the zip file and enter into the folder

Copy the input to the assembler in the file input.txt

Then run:

g++ .\transpile.cpp .\parser.cpp .\opTab.cpp .\symTab.cpp .\utils.cpp .\Pass1.cpp

.\Pass2.cpp -o Assembler

And then:

./Assembler.exe

**NOTE:**

1. ORG was not implemented

\*\*\*Intermediate File\*\*\*

0000 SUM START 0

0000 FIRST LDX *#0*

0003  LDA *#0*

0006  LDB *#TABLE2*

        BASE TABLE2

000a LOOP ADD TABLE X

000d  ADD TABLE2 X

0010  TIX COUNT

0013  JLT LOOP

0016  STA TOTAL

001a  RSUB

001d COUNT RESW 1

0020 TABLE RESW 2000

1790 TABLE2 RESW 2000

2f00 TOTAL RESW 1

END 2f03

COUNT 1d

FIRST 0

LOOP a

TABLE 20

TABLE2 1790

TOTAL 2f00

DEFAULT DEFAULT 2f03 0 0 0

\*\*\*Listing File\*\*\*

SUM START 0

0000 FIRST LDX *#0   050000*

0003  LDA *#0   010000*

0006  +LDB *#TABLE2   6a101790*

000a  BASE TABLE2

000a LOOP ADD TABLE X  1ba013

000d  ADD TABLE2 X  1bc000

0010  TIX COUNT   2f200a

0013  JLT LOOP   3b2ff4

0016  +STA TOTAL   0f102f00

001a  RSUB    4f0000

001d COUNT RESW 1

0020 TABLE RESW 2000

1790 TABLE2 RESW 2000

2f00 TOTAL RESW 1

END 2f03

\*\*\*Object Program\*\*\*

HTEST\_\_000000

T0000001D0500000100006A1017901BA0131BC0002F200A3B2FF40F102F004F0000

E002F03

M00000705

M00001705

The program will run for input file **file.in** and print all amongst the **intermediate, listing and object program in a single file (**ifstream was not working properly on separate files**), as shown below:**

\*\*\*Intermediate File\*\*\*

0000 SUM START 0

0000 FIRST LDX *#0*

0003  LDA *#0*

0006  LDB *#TABLE2*

        BASE TABLE2

000a LOOP ADD TABLE X

000d  ADD TABLE2 X

0010  TIX COUNT

0013  JLT LOOP

0016  STA TOTAL

001a  RSUB

001d COUNT RESW 1

0020 TABLE RESW 2000

1790 TABLE2 RESW 2000

2f00 TOTAL RESW 1

END 2f03

COUNT 1d

FIRST 0

LOOP a

TABLE 20

TABLE2 1790

TOTAL 2f00

DEFAULT DEFAULT 2f03 0 0 0

\*\*\*Listing File\*\*\*

SUM START 0

0000 FIRST LDX *#0   050000*

0003  LDA *#0   010000*

0006  +LDB *#TABLE2   6a101790*

000a  BASE TABLE2

000a LOOP ADD TABLE X  1ba013

000d  ADD TABLE2 X  1bc000

0010  TIX COUNT   2f200a

0013  JLT LOOP   3b2ff4

0016  +STA TOTAL   0f102f00

001a  RSUB    4f0000

001d COUNT RESW 1

0020 TABLE RESW 2000

1790 TABLE2 RESW 2000

2f00 TOTAL RESW 1

END 2f03

\*\*\*Object Program\*\*\*

HTEST\_\_000000

T0000001D0500000100006A1017901BA0131BC0002F200A3B2FF40F102F004F0000

E002F03

M00000705

M00001705

Any error generated in the program will be listed in the file.out file.

**Assembler Design:**

The assembler is divided in the following big modules:

**transpiler.cpp:** The main transpiler module of the assembler. It includes all the functions to generate tables like **opTab, regMap**, generate all directives and then initializes the regFile and  the first program block. It then takes the input from the file line by line and then then stores it in the vector vec. It then calls the parse() function which divides the input line into various fields.

**structs.h:**

structs.h:   
#ifndef STRUCT\_H

#define STRUCT\_H

#include <bits/stdc++.h>

using namespace std;

const unsigned int FORMAT\_1 = 1;

const unsigned int FORMAT\_2 = 2;

const unsigned int FORMAT\_3\_4 = 3;

struct OpCodeStruct

{

    std::string mnemonic;

    int opcode;

    int possibleFormat;

};

struct ObjCode

{

    bool isData = false;

    bool hasReg = false;

    bool isOnlyOpcode = false;

    int isWord = -13371337;

    int opcode;

    int displacement;

    int reg1;

    int reg2;

    int ni;

    int xbpe;

    unsigned int format;

    string data = "";

};

struct parsedLine

{

    long long int location;

    string label;

    string opcode;

    string op1;

    string op2;

    string err;

    bool isFormat4 = false;

    ObjCode objCode;

};

struct BlockTable

{

    string name = "DEFAULT";

    int number = 0;

    long long int locCtr = 0;

    long long int startingAddress = 0;

    long long int blockLength;

};

struct SymStruct

{

    string label;

    int location = -1;

    BlockTable block;

    string flags = "R";

};

struct LiteralStruct

{

    string value;

    long long int address;

    BlockTable block;

    int size;

    bool dumped = false;

};

struct TextRecord

{

    int startingAddress;

    int size;

    string text;

    string label;

};

struct ModicationRecord

{

    char label = 'M';

    int start;

    int modified;

};

#endif

**Pass1.h :** In this an intermediate file is generated which gives the locCtr, symTab, litTab, and BlockTable for debugging while this assembler was created. **Pass2.h:** This generates a listing file, which includes the **locCtr, objCode, for each parsedLine. We iterate over the** *parsedLine* **vector** and create *objectCode* accordingly, that is, for two operand instructions (which is the case with registers only), similarly for single operand and FORMAT\_3\_4 instructions are handled the similar manner, as in the below code snippet.

 if (pl.isFormat4)

            {

                locCtr += 4;

            }

            else

            {

                locCtr += op.possibleFormat;

            }

            ll pcRel = active.startingAddress + locCtr;

            if (op.possibleFormat == FORMAT\_2)

            {

                createObjectCodeWithRegisters(pl, opTab, regs);

            }

            else if (op.possibleFormat == FORMAT\_1)

            {

                createObjectCodeWithOnlyOpcode(pl, opTab);

            }

            else

            {

                createObjectCodeForInstruction(pl, opTab, symTab, litTab, pcRel, modifications);

            }

**Below is the way how listing file is printed using directives.**

tvoid printParsedLineListing(parsedLine pl)

{

    auto p = genObjcode(pl.objCode, pl);

    cout << setfill('0') << setw(4) << right << hex << pl.location << " " << pl.label << " " << printFormat4(pl) << pl.opcode << " " << pl.op1 << " " << pl.op2 << " ";

    cout << pl.err << " " << setfill('0') << setw(p.second \* 2) << right << hex << p.first << "\n";

}

void printParsedLineWithoutObjCode(parsedLine pl)

{

    cout << setfill('0') << setw(4) << right << hex << pl.location << " " << pl.label << " " << pl.opcode << " " << pl.op1 << " " << pl.op2 << " " << pl.err << "\n";

}

**utils.cpp:**  Below file holds several methods which are used all over the program for modularity and reuse.

#include "utils.h"

string toUpperCase(string su)

{

    transform(su.begin(), su.end(), su.begin(), ::toupper);

    return su;

}

string getString(char x)

{

*// string class has a constructor*

*// that allows us to specify size of*

*// string as first parameter and character*

*// to be filled in given size as second*

*// parameter.*

    string s(1, x);

    return s;

}

std::string string\_to\_hex(const std::string &input)

{

    static const char hex\_digits[] = "0123456789ABCDEF";

*// trouble with quotes in c++*

    std::string output;

    output.reserve((input.length() - 1) \* 2);

    for (int i = 0; i < input.length() - 1; i++)

    {

        unsigned char c = input[i];

        output.push\_back(hex\_digits[c >> 4]);

        output.push\_back(hex\_digits[c & 15]);

    }

    return output;

}

bool isNumeric(string s)

{

    if (s.length() == 0)

        return false;

    for (unsigned int i = 0; i < s.length(); i++)

    {

        if (s.at(i) - '0' < 0 || s.at(i) - '0' > 9)

            return false;

    }

    return true;

}

bool validf3(int disp)

{

    if (-2048 <= disp && disp <= 2047)

    {

        return true;

    }

    return false;

}

bool validf4(int disp)

{

    if (-524288 <= disp && disp <= 524287)

    {

        return true;

    }

    return false;

}

string print\_hex\_from\_bin(int str)

{

    char buf[2];

    sprintf(buf, "%0x", str);

*// cout << "in print hex: " + string(buf) << endl;*

    return string(buf);

}

pair<int, int> genObjcode(ObjCode obj, parsedLine &pl)

{

    if (obj.isWord != -13371337)

    {

        return {obj.isWord, 3};

    }

    int op1 = obj.ni + obj.opcode;

    if (obj.isOnlyOpcode)

    {

        return {obj.opcode, 1};

    }

    if (obj.isData)

    {

        return {stoi(obj.data, 0, 16), obj.data.length() / 2};

    }

    if (obj.hasReg)

    {

        int ans = (op1 << 8) + (obj.reg1 << 4);

        if (obj.reg2 == -1)

        {

            return {ans, 2};

        }

        else

        {

            ans += obj.reg2;

            return {ans, 2};

        }

    }

    int op2 = obj.xbpe;

    int op3 = obj.displacement;

    if (pl.isFormat4)

    {

        return {(op1 << 24) + (op2 << 20) + (op3 & 0xffff), 4};

    }

    return {(op1 << 16) + (op2 << 12) + (op3 & 0xfff), 3};

}

string getProgramName(string label)

{

    if (label.length() > 6)

    {

        return label.substr(0, 6);

    }

    else

    {

        string ans = label;

        while (ans.length() < 6)

        {

            ans.push\_back('\_');

        }

        return ans;

    }

}

**genObjCode(): This method generates object code based on different formats, viz.**

**WORD, BYTE , LITERALS are based on obj.isData and obj.Data.**

**Normal instructions are checked for FORMAT\_3\_4.  
Based on reg1 and reg2, based on FORMAT\_2**

**FORMAT\_1 is checked from `needNoOperands` during parsing (two distinguish between opcode and operand vs label and opcode).**

**ObjectProgram.cpp:** This holds the program for writing various records in the object program.

writeHeaderRecord:

void writeHeaderRecord(string name, int startingAddress)

{

    cout << 'H' << getProgramName(name) << setfill('0') << setw(6) << right << hex << startingAddress << '\n';

}