# Report for project 1

## 1. Overview

In project 1, we are required to implement a MIPS assembler. MIPS is and assembly language where each line of code corresponds to one machine instruction involving register manipulation, calculation, control flow, system call and so on. In a 32-bit MIPS architecture, the CPU actually runs 32-bit binary machine code. As a simple MIPS assembler, we should assemble the assembly code into the binary machine code.

In project 1, we only care about the .text section containing the instructions, labels and comments. Following the advice from the assignment description, I scan through the file for the first time (phase 1) to build the label table so that I can find  the corresponding address of each label. Then we scan through the file for the second time (phase 2) and converting each instruction into its corresponding binary machine code, while replacing the labels with their corresponding relative address or absolute address with the help of the label table.

## 2. Method and Implementation

### i. Preparatory work

To convert the MIPS code into binary code, we must know the mapping rules between MIPS code binary code, such as opcode corresponding to each type of instruction and the meaning of arguments for each instruction. If we represent the mapping logic in our program directly, the whole project will become a mess. Thus, I build the mapping tables for registers and instructions using the structure below, so that the program can be simplified.

```
// LabelTable.h

enum INST_TYPE { R_TYPE, I_TYPE, J_TYPE };

enum ARG_TYPE { RS, RT, RD, SA, IMM, LABEL, IMM_RS, NO_ARG };

struct InstTable {
    std::string instName;
    INST_TYPE instType;
    uint32 opCode;
    uint32 functCode;
    ARG_TYPE argList[3];
};
```

To simplify the program, `INST_TYPE` contains all possible types of instruction, `ARG_TYPE` contains all possible argument of instruction, such as rs, rt, rd, sa, immediate, label and immediate(rs). Then, we can describe the structure of each instruction using `InstTable` containing the name of instruction (add, addiu, etc.), the type of instruction, opcode, function code and the representation of its argument.

To build the whole mapping table, we can use code like below:

```
// LabelTable.cpp

InstTable instTable[INST_TYPE_CNT] = {
    /* R-type instructions */
    {"add", R_TYPE, 0, 0x20, {RD, RS, RT}},
    {"addu", R_TYPE, 0, 0x21, {RD, RS, RT}},
    {"and", R_TYPE, 0, 0x24, {RD, RS, RT}},
    ......
}
```

which means that the add instruction is R-type, its function code is 0, its opcode is 0x20, its following arguments represent rd, rs, rt in sequence.

To build the mapping of register, we can use code like below:

```
// LabelTable.cpp

std::string regList[REG_CNT] = {
    "$zero", "$at", "$v0", "$v1", "$a0", "$a1", "$a2", "$a3",
    "$t0", "$t1", "$t2", "$t3", "$t4", "$t5", "$t6", "$t7",
    "$s0", "$s1", "$s2", "$s3", "$s4", "$s5", "$s6", "$s7",
    "$t8", "$t9", "$k0", "$k1", "$gp", "$sp", "$fp", "$ra"
};
```

which gives the name of register in order, so we can find the corresponding identifier of each register.

## ii. Phase 1

In phase 1, we simply scan the file line by line, leaving out the comment, finding the start of `.text` section recording the number of actual instruction and find the corresponding line number of each label.

```
// phase1.cpp
    insCnt = 0;
    /* .text section begins */
    /* assuming that the input is valid, find all label */
    std::stringstream ss;
    std::string label;
    while (getline(inFile, str)) {
        ss.clear();
        ss.str(str);
        ss >> label;
        if (label.back() == ':') {
            labelMap[label.substr(0, label.length() - 1)] = insCnt;
        }

        if (trimToIns(str).length()) {
            insCnt++;
        }
    }
```

Using the `stringstream`, we can filter the label, and fix its position using the colon. After that, our label table is completed.

### iii. Phase 2

Using the label table and the mapping table, we can conveniently finish the phase 2. The whole process is like below:

```
// phase2.cpp
void LabelTable::pass2(std::string filename, std::string
outputFilename) {
    ......
    /* .text section begins */

    while (getline(inFile, str)) {
        str = trimToIns(str);
        if (str.length()) {
            outFile << parse(str) << std::endl;
            insCnt++;
        }
    }
}
```

That is to say, we leave out all the  comments and labels outside the instruction, and parse them into binary code.

While parser finds the name and the argument list of each instruction and applies different method according to its type:

```
    std::vector<std::string> argList;
    while (ss >> arg) {
        if (arg != ",") {
            if (arg.back() == ',') {
                arg = arg.substr(0, arg.length() - 1);
            }
            argList.push_back(arg);
        }
    }

    INST_TYPE instType = instMap[instName]->instType;
    switch (instType) {
        case R_TYPE:
            return parseR(instName, argList);
        case I_TYPE:
            return parseI(instName, argList);
        case J_TYPE:
            return parseJ(instName, argList);
    }
```

Then different type of parser do its work according to the mapping table.

Taking parser for R-type as an example, we can just join each part of the instruction code like below:

```
std::string LabelTable::parseR(std::string instName,
std::vector<std::string> argList) {
    std::string opCode = "000000",
                rs = "00000",
                rt = "00000",
                rd = "00000",
                sa = "00000",
                function = "000000";
    InstTable *instInfo = instMap[instName];
    function = numToBinStr(instInfo->functCode, 6);
    for (int i = 0; i < argList.size(); i++) {
        switch (instInfo->argList[i]) {
        case RD:
            rd = numToBinStr(regMap[argList[i]], 5);
            break;
        case RS:
            rs = numToBinStr(regMap[argList[i]], 5);
            break;
        case RT:
            rt = numToBinStr(regMap[argList[i]], 5);
            break;
        case SA:
            sa = numToBinStr(regMap[argList[i]], 5);
            break;
        }
    }
    return opCode + rs + rt + rd + sa + function;
}
```

In this way, we finish the whole process of assembling

## 3. Usage

### 1. Build

In my project, I use Makefile and g++ to build the project.

Use command below to build the project.

```
make test
# or
make
```

### 2. Test

```
./tester test_cases/testfile.asm output.txt
test_cases/expectedoutput.txt
```

To match the api the C++ code, I applied minor modifications to the `test.c` and turned it into `test.cpp`