MIPS Assembler Project Report

Yaodan Zhang

We want to translate the MIPS assembly language to its corresponding machine codes in this programming project. We classify the process into two phases. In phase 1, we process the file by removing its comments and empty lines. Also, we store the labels with their corresponding absolute address starting from 0x400000 along with their relative position (e.g., the first line is "0", the second line is "1", and so on). We store the processed file in an intermediate file called "FileProcessing.txt" which will be deleted after running the program. In phase 2, we scan this processed file line by line. This time, we match the line with its corresponding instruction type, get its correct machine code, and write the machine code into the output file called "output.txt". The whole process is done after we scan through all lines in the "FileProcessing.txt". After running the program, we will get an "output.txt" which stores the machine codes of the original MIPS file and lies in the same folder. We realize the whole process using 4 C++ files, "phase1.cpp", "phase2.cpp", "LabelTable.cpp", and "tester.cpp", and combine them using a Makefile.

Phase I

In phase 1, we scan the MIPS language file line by line for the first time in the "main" function in "tester.cpp". For each line, we remove the comments if there are any following a "#". The implementation is in the function "RemoveComments" in "phase1.cpp":

```
string RemoveComments(string line) {
   int found = line.find("#");
   if (found != -1) {
      return line.substr(0,found);
   }
   return line;
}
```

After removing the comments, we continue to search whether there is a label for this line. Note that labels are always followed by a colon. We store all labels in a map called "labelTable" declared in "LabelTable.cpp" as below:

```
std::map<string, vector<string>> labelTable;
```

More specifically, if there is a label in this line, we store the label's name as the key in the map and the pair (label's absolute address starting from 0x400000, label's relative position in the file) as the value in the map denoted as a string type vector. After that, we remove the label of the

line since we can already refer to it afterward using the “labelTable” map, and return the processed line to “tester.cpp”. The implementation is in the function “ProcessingLine” in “phase1.cpp”:

```
string ProcessingLine(string line, int n) {
    string tmp;tmp.clear();
    for(auto v:line){                                   //Here we remove all whitespaces and tabs
        if(v!=' '&&v!='\t')tmp.push_back(v);            //in the line first to make later
    }                                                   //implementation easier
    line = tmp;
    line = RemoveComments(line);            //Remove comments in the line as detailed above
    int found = line.find(":");             //Search if there is a label in this line
    if (found != -1) {
        string label = line.substr(0, found);
        labelTable[label]={addTwoBinarys("00000100000000000000000000",
        isBinaryNumber(n * 4)), to_string(n)};                  //Store the label’s addresses
        if (found + 1 == line.size()) {return "";}
        else {return line.substr(found + 1);}           //Remove the label and return the line
    }
    return line;
}
```

When receiving back the processed line, the “main” function in “tester.cpp” writes this line into the intermediate file “FileProcessing.txt” for the second time of the scan. The implementation of the “main” function related to phase 1 is in line 11 to line 31, as shown below:

```
fstream testFile;
testFile.open(argv[1], ios::in);            //This is the input MIPS file specified by the user
ofstream processedFile("FileProcessing.txt");                   //The intermediate file
if (testFile.is_open() && processedFile.is_open()) {
    string line;
    int n = 0;
    int start = 0;
    while (getline(testFile, line)) {                   //Scan through the MIPS file line by line
        if (line.find(".text") != -1) {start = 1; continue;}        //Read from “.text” section
        if (start == 1) {
            line = ProcessingLine(line, n);         //Process the line as above in “phase1.cpp”
            if (line == "") {continue;}
            else {
                processedFile << line << "\n";      //Write processed line into intermediate file
                ++n;                    //This number stores the relative position of a line (or a label)
            }
        }
    }
}
testFile.close();
processedFile.close();
```

Phase II

In phase 2, we scan through the file the second time by scanning the “FileProcessing.txt” created from phase 1. This time, without format distractions in the file, we get the machine code for each line that is realized in “phase2.cpp” and will be illustrated later, and write the machine code into “output.txt”. The related process in from line 32 to line 45 in “main” of “tester.cpp”:

```
ofstream outputFile("output.txt");    //Create the output file which contains machine codes
fstream sourceFile;
sourceFile.open("FileProcessing.txt", ios::in);                    //Read FileProcessing.txt
if (sourceFile.is_open() && outputFile.is_open()) {
        string line;
        int k = 0;
        while (getline(sourceFile, line)) {
          outputFile << GetMachineCode(line, k) << "\n";//Get machine code for this line
          ++ k;                                                    //and write it in output.txt
        }
}
sourceFile.close();
outputFile.close();
remove("FileProcessing.txt");                                    //Delete the intermediate file
```

Illustration now comes at how to get the machine code for a MIPS instruction line. The whole process is realized in “phase2.cpp”. Firstly, we identify whether this instruction is R type, I type, or J type. To do that, we define three type arrays, each constituting all the instructions under our consideration of this type:

```
const string R[27] =
{"add","addu","and","div","divu","jalr","jr","mfhi","mflo","mthi","mtlo","mult",
"multu","nor","or","sll","sllv","slt","sltu","sra","srav","srl","srlv","sub","subu","syscall","xor"};

const string I[26] =
{"addi","addiu","andi","beq","bgez","bgtz","blez","bltz","bne","lb","lbu","lh",
"lhu","lui","lw","ori","sb","slti","sltiu","sh","sw","xori","lwl","lwr","swl","swr"};

const string J[2] = {"j", "jal"};
```

Since a typical MIPS instruction is like “add $t0, $t1, $t2”, we first find the instruction name of this instruction (“add” in this case) and check which array this name is in and get its corresponding index in the array (0 in this case in R[27]). After that, we jump into corresponding functions, i.e., “RTypeCode”, “ITypeCode”, or “JTypeCode” to get its machine code. This part is implemented from line 353 to line 370 in “phase2.cpp”:

```
string GetMachineCode(string line, int k) {
```

```
    string operation;                                          //Find the operation name
    int ind = line.find("$");
    if (ind == -1){
       if (line.substr(0,3) == "jal") {operation = "jal";}
       else if (line.substr(0,1) == "j") {operation = "j";}
       else {operation = "syscall";}
    }
    else {operation = line.substr(0,ind);}
    if (isInArray(R, operation, 27) != -1) {                   //Check if it is R type
       return RTypeCode(isInArray(R, operation, 27), line);//If so process to "RTypeCode"
    } else if (isInArray(I, operation, 26) != -1) {            //Check if it is I type
       return ITypeCode(isInArray(I, operation, 26), line, k);//If so process to "ITypeCode"
    } else if (isInArray(J, operation, 2) != -1) {             //Check if it is J type
       return JTypeCode(isInArray(J, operation, 2), line);     //If so process to "JTypeCode"
    }
    return NULL;
}
```

After processing to the corresponding "XTypeCode (X = R, I, J)", the following implementations to find the machine code are a little tedious. Since this process is highly regular and repetitive, this report will only illustrate the above instruction, i.e., add $t0, $t1, $t2, as an example. We already know that this instruction is R type with array index 0. We thus proceed to "RTypeCode(0, line)" and find its corresponding "if" block. For an R-type instruction, the machine code format is operation code (6 digits) + rs (5 digits) + rt (5 digits) + rd (5 digits) + sa (5 digits) + function code (6 digits). Also, the operation code for all R instructions is 00000. Therefore, we only need to find rs, rt, rd, sa, and function code for this instruction:

```
string RTypeCode (int index, string line) {
   string opcode = "000000";
   string funcode, rd, rs, rt, sa, shift;
   if (index == 0||index == 1||index == 2||index == 13||index == 14||index == 17||
      index == 18||index == 23||index == 24||index == 26) {
      switch (index) {
         case 0: funcode = "100000"; break; //Index is 0, we find its function code is 100000
         case 1: funcode = "100001"; break;
         case 2: funcode = "100100"; break;
         case 13: funcode = "100111"; break;
         case 14: funcode = "100101"; break;
         case 17: funcode = "101010"; break;
         case 18: funcode = "101011"; break;
         case 23: funcode = "100010"; break;
         case 24: funcode = "100011"; break;
         case 26: funcode = "100110"; break;
         default: break;
```

```
}
int found = line.find("$") + 1;                 //Find the first operand, $t0 in this case
rd = findOperand(line, found);                  //Get the operand code for $t0
found = line.find("$", found) + 1;              //Find the second operand, $t1 in this case
rs = findOperand(line, found);                  //Get the operand code for $t1
found = line.find("$", found) + 1;              //Find the third operand, $t2 in this case
rt = findOperand(line, found);                  //Get the operand code for $t2
sa = "00000";                                   //sa is all 00000 for the "add" instruction
return opcode + rs + rt + rd + sa + funcode;    //Return the machine code for this MIPS line
```

We now successfully get the machine code for this MIPS instruction line. After that, we return the machine code to the "main" function in "tester.cpp". The "main" function will then write it into the output file "output.txt". After writing the machine codes for all lines, we close the output file and the intermediate file "FileProcessing.txt". Moreover, we delete the intermediate file since we don't want it as our program output.

How to Run and Test the Program

In this section, we introduce how to run the program and what the outputs' meanings are. To run the program, we first compile it by typing "make" in the terminal. After that, we use:

./tester MIPSFileName ExpectedOutputFileName

to run and test the program. For example, if the MIPS file we want to assemble is *testfile.asm* and the expected correct machine code file is *expectedoutput.txt*, we shall type the following command in the terminal:

./tester *testfile.asm expectedoutput.txt*

After running the program, you will see an *output.txt* in the same folder which contains the program's output machine codes for the *testfile.asm*. Also, you will either see

"Congrats output is correct !!" or "Output is incorrect !"

displayed in the terminal. The first case means our output file *output.txt* completely matches the expected output file *expectedoutput.txt* by assembling all machine codes correctly for the *testfile.asm*. The second case means there are some errors in the *output.txt* which make its machine codes mismatch the correct machine codes in *expectedoutput.txt*.