In this project, all functions and classed are declared in the “tools.h” and implemented in the “tools.cpp”. The “tester.cpp” is used to test this project. If the output file is the same as the expected output file, it will print “ALL PASSED! CONGRATS :)”, else it will print “YOU DID SOMETHING WRONG :(”.

Big Picture Thoughts and Ideas

First, the project classifies all instructions according to how they are input. Each kind of instructions corresponds to a way of converting into machine code. The project use maps to store instructions and their machine codes, the unknown parts are temporarily denoted by letters. Then, the project scans the input file twice. On the first scan, the project generates a map about the labels and their address. On the second scan, this project replaces the letters in the machine code of the corresponding instruction with the corresponding binary number based on the contents of each line, and then output it to the output file.

High Level Implementation Ideas

First, to solve the label jump problem, we scan the input file once and store all labels and their addresses. Second, to solve the problem of converting the entire file, we first consider what to do with one line of the input file, and then scan the input file line by line. One of the following situations can occur in each line of content: 1.only comments. 2.only a label. 3.only an instruction 4.label+comments. 5.label+instruction. 6.label+instruction+comments. 7.instruction+comments. 8.empty. We delete comments of the line, then the situations can be reduced: 1.only a label. 2.only an instruction. 3.label+instruction. 4.empty. For 1, just read next line. For 2 and 3, We judge by whether the first word contains “:” or not. And then, replace letters in the corresponding machine code according to instructions. For 4, we just skip. For how to ignore parts except “.text”, we import a flag. When the first word of the line is “.text”, we set flag from false to true, and then begin converting instructions. For instructions that contain jumps and branches, we import a counter to store the address of current instruction, and so with the help of the labels table, we can convert instructions that contain jumps and branches into machine code. To solve the problem of integers in instructions, we create a function that can converts a decimal number to a binary two’s complement.

Implementation Details

In this project, we use string to store binary numbers, so we create a function that can convert decimal(int) into binary 2’s complement(string). First, we use “map<string, string>” to store register names and their register number (e.g., regMap[“$zero”] = “00000”). Then, we classify instructions according to their form (e.g., R1->xxx rd rs rt, I2->xxx rt rs imm……). We use “const map<string, string>” to store them and their corresponding machine codes, temporarily unknown parts are replaced with letters. (e.g., R1[“add”] = “000000ssssstttttddddd00000100000”).

We use “map<string, int>” to store labels and their addresses, and we create a function to scan file and find labels. In the first scanning, we find “#” and delete contents behind it, we also set a flag so that we can skip other parts except “.text”. We replace the “,”, “(“, “)” with “ ” so that we can use “stringstream” to divide this line into five strings. When the first string has “:”, we record the label’s name and its address into the map.

We create a function named “assembler” to convert an instruction into machine code, it judges its class and replaces letters in the common form of instructions according to its input content. In the second scanning, we first define an “int” to record initial address, so that we can deal with jumps and branches. For each line, we find “#” and delete contents behind it, we also set a flag so that we can skip other parts except “.text”. We replace the “,”, “(“, “)” with “ “ so that we can use “stringstream” to divide this line into five strings. If the first string is a label and there is empty behind it, we skip. If the first string is a label and there is an instruction behind it, we put the last four strings into “assembler” function and get the machine code. If the first string is an instruction, we put the first four strings into “assembler” function and get the machine code. If the first string is empty, then this line is empty after deleting comments, so we just skip. Last, we stream the machine code to the output file.

**Appendix:**

map<string, string> createRegMap(): create the binary form of registers in map (e.g., regMap[“$zero”] = “00000”).

const map<string, string> R1, R2, R3, R4, R5, R6, R7, R8, I1, I2, I3, I4, I5, J: maps between assembly and machine code, classify by input form (e.g., R1[“add”] = “000000ssssstttttddddd00000100000”).

map<string, int> findLabels(ifstream& infile): find labels and record their address.

string decToTwoComplement(int num, int len): convert the number from decimal form to two's complement form with specific size.

void reline(string& line, const string& before, const string& after): replace “<”, “(“, “)” with “ ” in the instruction.

string assembler(string t1, string t2, string t3, string t4, map<string, int>& labels, int& curAddr): convert a line instruction into machine code.

void getOutFile(ifstream& infile, ofstream& outfile, map<string, int>& labels): main process, create the output file.

bool compareFiles(ifstream& f1, ifstream& f2): compare the outputfile with the expected.