**Project Report**

**李华悦 118010138**

**1 understanding of this project**

The purpose of this project is to write an assembler that can translate a MIPS program into machine language. The C++ program needs to read all the MIPS instructions from an asm file and then translates them to machine code which will be stored in a txt file. Each MIPS instruction corresponds to a line of 32 binary numbers. There are label, register, number and instruction name in the asm file, so the key point of this project is to get them, translate them to corresponding machine code and finally store the machine code in a txt file.

**2 big picture idea**  
The project can be divided into three parts:

1. Read the asm file line by line, get the information of the instructions and labels in the asm file and store them through C++ code. Then store the important information of each line of instruction like register, instruction name and number. That’s all the preparation.
2. Process instruction line by line. Find the corresponding translation formula according to the instruction name of each line. The entire line of instruction is then translated into binary machine code based on the formula. Then the machine code is stored. This process is then repeated until the last line of instruction is completed.
3. Finally, write all the machine code into a txt file in order. The process of translation is over.

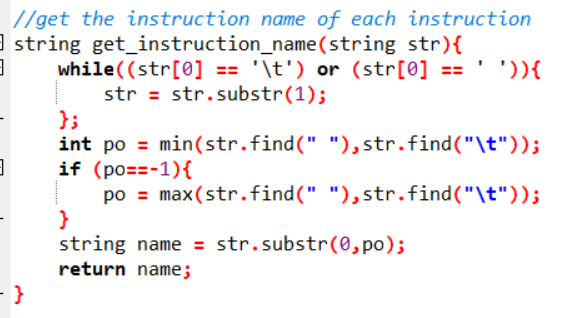
**3 how to store meaningful data**

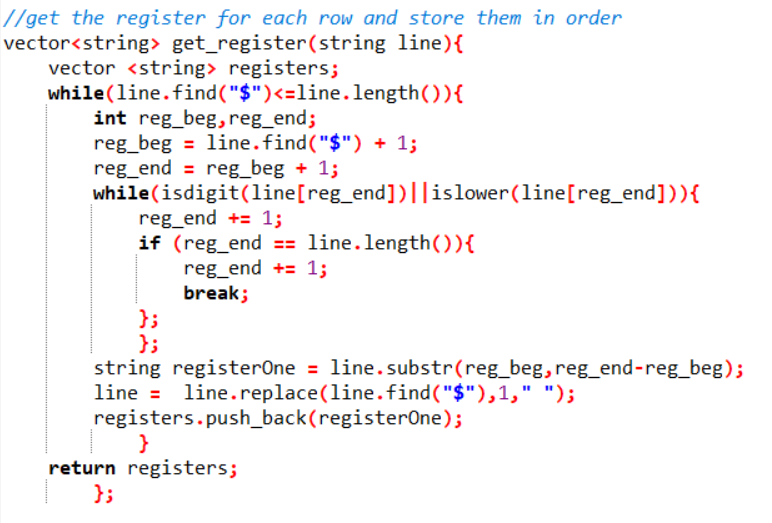
A vector is created and all registers are stored as elements in the form of strings.

A vector is created and each line in the ams file are stored as a string in the vector.

I take out all the labels according to the “:” and store them in a map whose key is the name of the label and whose value is the location of the label.

I use the self-created function to extract the instruction name of each line of instruction and all the registers, and stored the registers orderly in a vector.

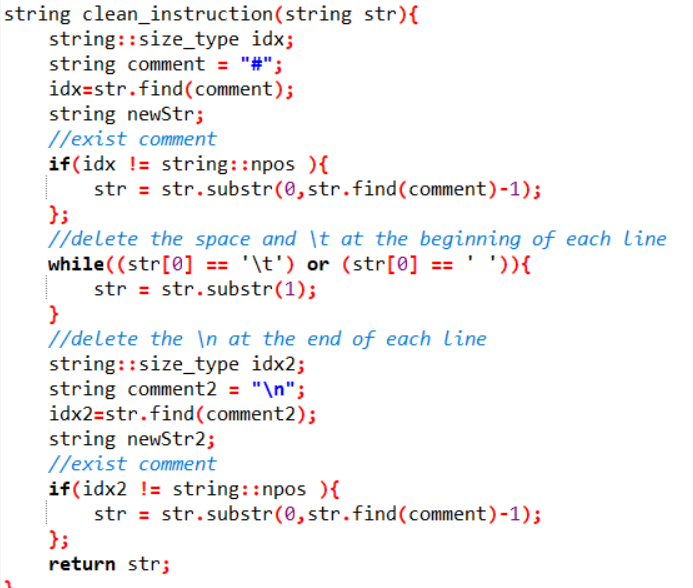




After each translation, I store the machine code as a string in a vector.

**4 how to translate the instruction into machine code**

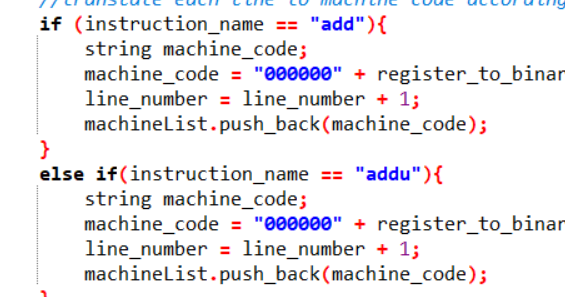
In order to get more useful information, I first delete the non-translation-related parts such as comments through a self-created function.



Then I try to find the corresponding formula according to the instruction name of each line.

Because there are 70 kinds of instruction name in total, I use “if-else if” structure to judge and then translate the instruction to machine code.

Example:

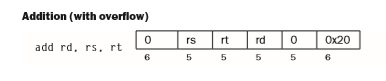


**5 the whole process of analyzing the project**

The length of machine code for every instruction is the same, which is 32. Because I am not familiar with C++ and the number of all the instructions from A51 to A71 is limited, I try to translate the instructions into machine code in an intuitive and obvious way. The number of all the instructions from A51 to A71 in the textbook is 70, so I use 70 “if-else if” structure to judge which instruction is on each line.

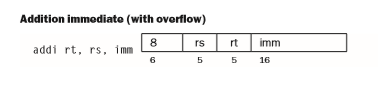
After observing the structure of all the instructions, I did not divide all the instructions into three (R, I, J) groups. Instead, I use my own classification method to distinguish. Here are all the types of structures I found. I will give an example for each type:

(1)



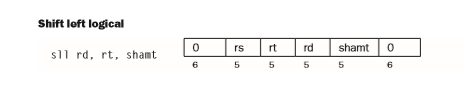
For this type, I just need to get all the registers and translate them into binary number and then put them in the machine code string in order. Then the translation for this type is over.

(2)



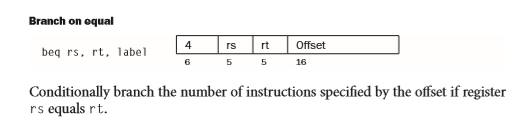
For this type, I just need to get all the registers and handle them by using the way I did in (1). For imm, it is a real number, so I need to get it and then translate it to binary number. Because imm can be negative and the binary for negative number should be 2’s complement, I need to add a judgement in my “dec\_to\_binary” function to judge whether the imm is negative or not. After that, I put all the binary string in the machine code string in order. Then the translation for this type is over.

(3)



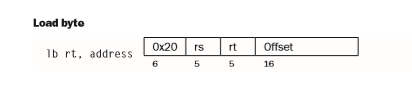
This type is similar to the type (2). The shamt in this type is similar to the imm in (2), so the specific operation methods can refer to the method in (2).

(4)



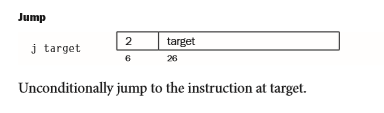
For this type, I have stored all the labels and their locations in a map before translation. What is more, I use a variable “line\_number” to store the line number for each instruction. Therefore, I can get the offset in this type, put all the substring in the machine code string in order and then the translation for this type is over.

(5)



For this type, because the structure of the offset is like number(register), so I get the number and register by identifying “(” and “)”. I will translate the number and all the registers to binary, put them in the machine code string and then the translation for this type is over.

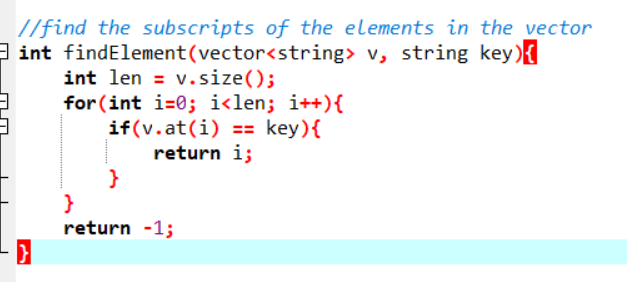
(6)



For this type, the target can be number or label, so I first get the target and judge it is number or label. If it is number, the method is similar to type (2). If it is label, the method is similar to type (4). After that I put the substring in the machine code string and then the translation for this type is over.

The machine code will translate all the instructions in the asm file line by line and will pass one line if there is no instruction in it. All the machine code will be stored in a vector and written into a txt file. For Mac, you can use diff to judge whether the txt file is the same as the expected txt file (to judge whether the translation is right or not). That is all for my project.

1. **Special points of my project**
2. To calculate the relative address for offset and target, I create a variable named line\_number. Because only the line where there is an instruction is meaningful, every time a translation is over, the line\_number will plus 1, then I will get the address of each instruction.
3. To get the register of each line, I first find the “$” in each line and then take the substring backwards until it is no longer an alphabet or a number. Repeat this process until there is no “$”. Finally, I will get all the registers and store them in order.
4. Considering negative number should be translated to 2’s complement, I create a function to distinguish positive and negative decimal to binary conversions.
5. Because C++ cannot find the index of element in vector directly and I need the index of element in the registerList vector to get the corresponding number of the register, I create a function to implement.



**7. how to run my code (in Mac)**

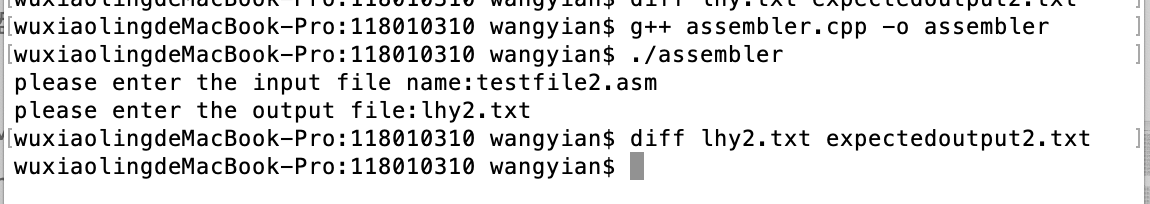
(1). Run the C++ code

(2) Enter the input asm file name in the terminal

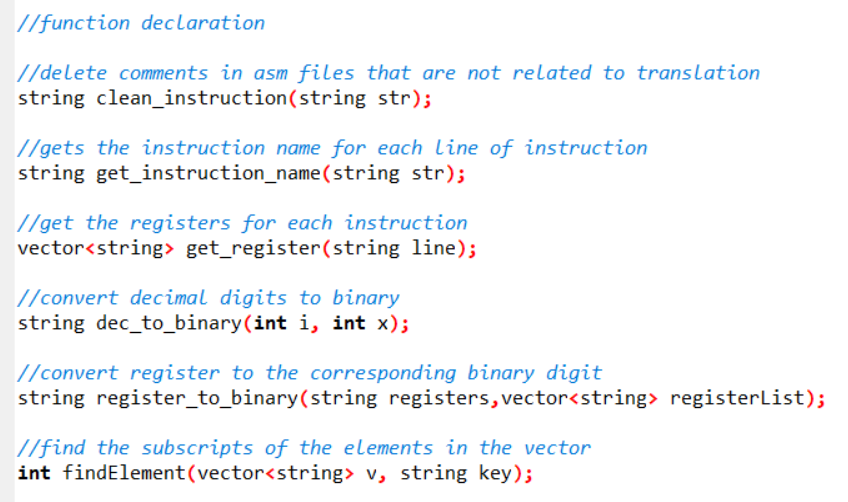
(3) Enter the output txt file name in the terminal

(4) Then you will get an output.txt file in the same folder, you can use diff to check it with the expected.txt file.

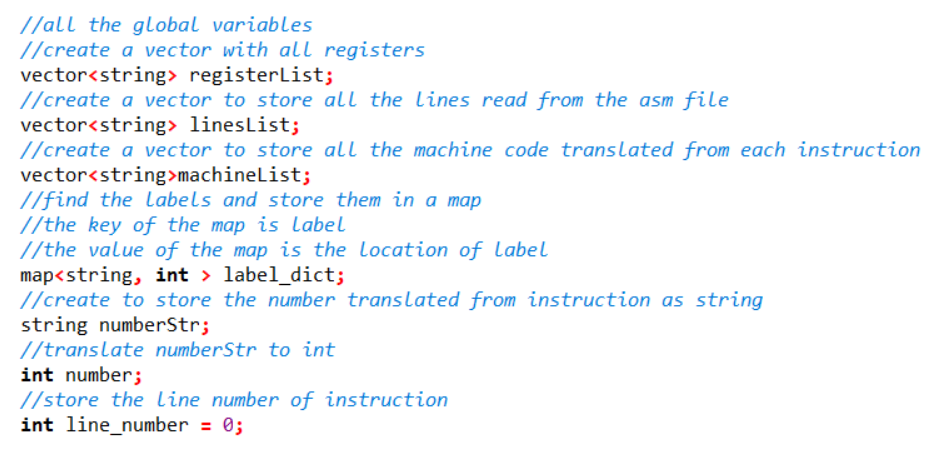
(5) Exit.



**8. all the functions**



**9 all the global variables**



**10. Summary**

This project has tested our carefulness, patience and ability to summarize. Although I am not good at C++, I really try to learn some functions in C++ by myself to write this project as well as possible in limitation of my ability. Although my code may not be as good as other students, I really try my best. It was an unforgettable experience for me.