Design

Yohandi (120040025)

Overview

This project is about designing and building a code that translates into a well-known assembly language, that is MIPS. As a matter of fact, the computer that we all know can not read the MIPS code directly. The code requires itself to be translated to a machine code containing 0-bit and 1-bit. After being translated, the computer will be able to read the machine code and execute the operation directly from the given task.

There are actually 55 MIPS instructions that are to be transliterated. The instructions are categorized into three types those are R-type, I-type, and J-type. Each type represents a different interpretation of the machine code. Each of the instructions falls into one category. Because of this reason, each of the instructions must be solved case by case. The thoughts and ideas will be explained later in the Processing Logic part.

Data Type and Data Structure

Boolean

In C++, boolean variables are defined by a true or false value. It is possible to use the integer type of variable to indicate 0 as a false value and 1 as a true value. However, the boolean variable saves up a lot of the allocated memories.

In this program, the boolean variable is used for many cases that only consider two possible choices. In its example, it is used to determine whether the current state is in the `.text` state or `.data` state.

Integer

In C++, an integer variable is a data type that stores numerical values. As explained in the boolean part, the integer variable can be used to store values such as 0, 1, 109, and possibly many numbers within some specific ranges. In its example, it is used to determine the current number line while doing the calculation of a label address.

Aside from the basic integer variable, int32_t is also used in this program. Mainly because the variable provides similar advantages as the normal integer type with 32 bits. This type is specifically used in storing the information of a label address.

Array

In C++, an array is a sequential value list. The array allows storing, changing, and accessing a specific element based on the index parameter in constant time. An array is used to obtain the given arguments in this program while running the executable program.

Bitset

Bitset is a container in C++ STL (Standard Template Library). Mainly, the use of bitsets creates simplicity in handling some variables that primarily use a bit. For example, while converting a label address from the int32_t type to binary, it can be easily done with this provided bitset from the STL.

String

A string is a collection of variable characters, which usually is used to store text. The string variable can also be considered as a data structure provided in C++ STL. This variable is a core in this program since all of the inputs and outputs use this string type. For example, each of the instructions can be saved to a string such as `add $t3 $t1 $t2`. Moreover, the code's core address needs to be stored inside a string variable for its label.

Map

In C++, creating an array-like type that has a custom key is not an easy task. That is why having a map data structure that supports any type of key is a must. Although using this map data structure costs more storage data by log multiplication in terms of space complexity, it eases the implementation job. In its example, a map variable is used to store the label as its key and its address as its value. In order to get the address, the value can be simply returned by calling the map directly. Moreover, the map is also used as a constant library to store opcode in I-type and J-type instructions, function code in R-type instructions, and the binary information of register numbers.

Vector

A vector, in its simplest way, is an array that is dynamic. The size of the allocated memory can be added and removed like a stack data structure bases. In this program, one of the uses of a vector is to store some strings compacted in one variable. Another thing, the vector that contains those strings can be stored again inside a vector, making it another dynamic array in a two-dimensional type.

Processing Logic

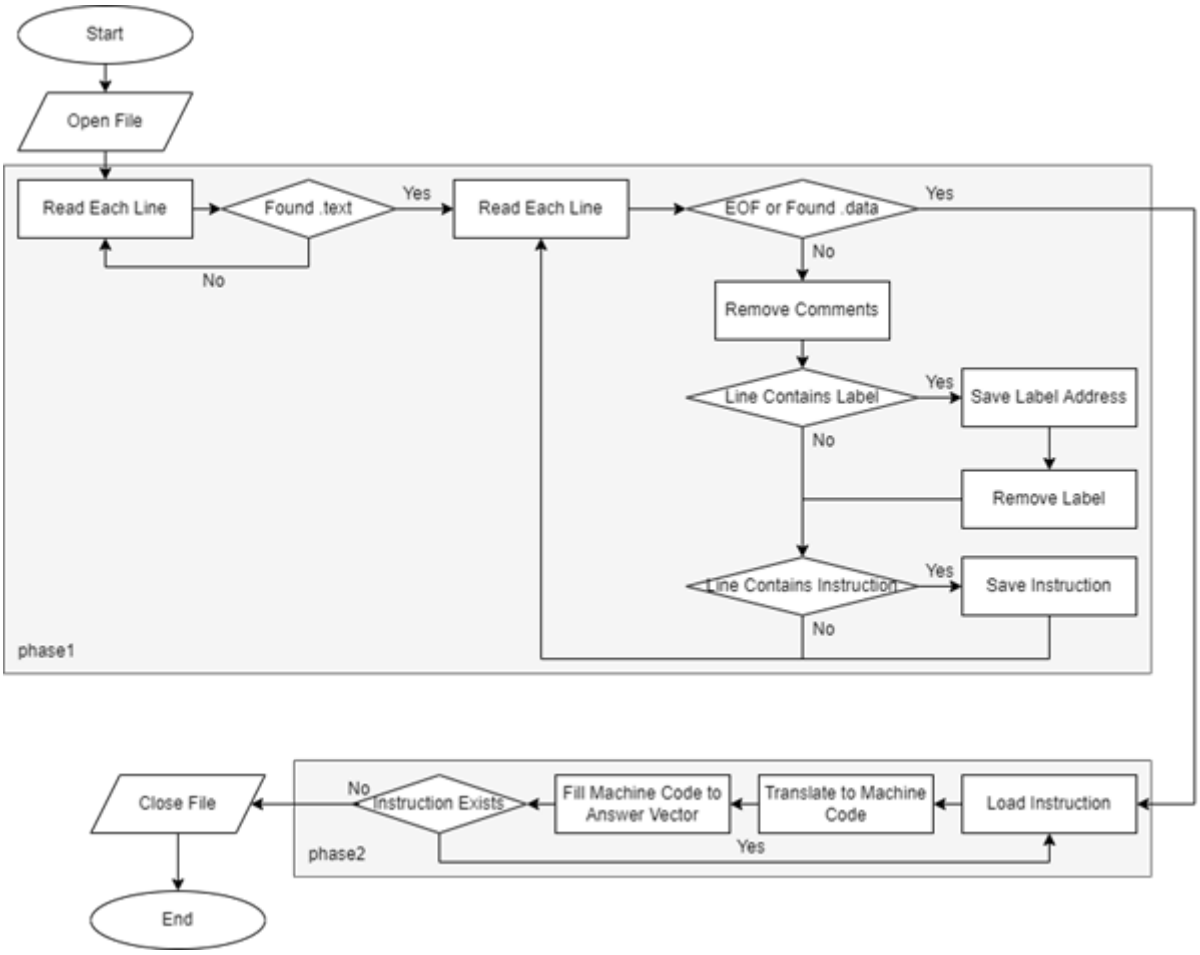
First, it must be noticed that the goal of this project is to translate every instructions found in the MIPS code into a list of machine codes. The MIPS code itself contains some sections such as `.data` and `.text`. The instructions that are mentioned before are mainly taken from the `.text` section. Other sections aside from the `.text` section are ignored in this project. Because of this reason, the work of separating the information will be done by the phase1 class. After that, it is required to save every specific label address. The label address will be stored in the labelTable class as a custom data structure. Subsequently, phase2 class works as the translator of the cleaned version of instructions. Lastly, the main source of this project will read the input file, write a new output file, and compare the new output file with the expected output file.

The class of phase1 is intentionally designed to deal with the `.text` section for assembling. After receiving the input file name from the main, the function `separated_text_type` will first determine the location of the section; after confirming it to be on the `.text` section, the function will begin to consider each of the lines input given. This work can be handled by maintaining a boolean expression that will be set true once a substring of `.text` is found in the line. Not to be left behind, the task of removing comments must also be worked on. This can be simply done by scanning each of the characters in the line, and once a character `#` is found, the current character and the after characters will be ignored. After that, each line can be concluded that it contains either label declaration, instruction, or both. For the line that contains some labels, the addresses of those labels need to be saved in the labelTable class, which later will be explained on how it works. For the line that contains an instruction, later on, it will be saved in the split format with the help of `split_string` function, and the information will be transferred back to the main source, which later will be used as a parameter in phase2 class.

The class of phase2 is designed for the implementation of each of the MIPS instructions. Phase2 class mainly translates the instruction into a machine code. The file contains several dictionaries to keep the constant available without having to reconsider each key value. Those dictionaries are stored in the map data structure. The function `machine_code` in this file is called exactly as many as the instructions are given. In the

`machine_code` function, it first determines the operation given in the instruction and solves it case by case. Besides, other functions such as `r_instruction`, `i_instruction`, and `j_instruction` are also called in this function with the purpose of simplicity. Those functions concatenate the given modified information to become a line of machine code. Later on, the concatenated string will be pushed as an answer to the vector in the main source. Bitset, which is provided by C++ Standard Template Library, comes in handy for both `i_instruction` and `j_instruction` that require converting a label address into a binary string. The conversion can be done with `bitset<number_of_bits>(value)` property by bitset.

The class of labelTable is intentionally created as storage. The storage saves the address of all collected labels in a map data structure. The file contains three functions where one of the functions is to save the address, and the other two are to access the address value based on the label variable. The calculation of absolute address is `0x400000 + 4 x line_number`. The calculation of relative address is the value of absolute address - `0x400000 + 4 x line_number`. The function in labelTable will often be called in phase1 class to save label addresses. The function in labelTable will also often be called in phase2 class to get the label addresses information.



Function Specification

1. `string remove_specific_char(char c, string line);`

This function returns a new modified string, which does not contain a specific character anymore. The specific character refers to the character parameter c, and the pre-modified string refers to the string parameter line.

2. `void remove_comments(string &line);`

This procedure removes a substring from a line that contains comments—specifically, a substring where it starts from `#`. The line refers to the string parameter line. The use of the address in the parameter is to change the string inputted from outside of the function.

3. `vector<string> split_string(string divider, string str);`

This function returns a vector of strings where it is originally coming from string str. The split strings are divided based on the custom divider. This function takes two string as its parameters. Those are the pre-mentioned divider and str.

4. `vector<vector<string>> separated_text_type(string input_file);`

This function returns a vector of vectors of strings. It is a cleaned version of the inputted file where it does not contain comments, labels, and any additional spaces/tabs. Later on, the label address will be saved by this function.

5. `string r_instruction(string op, string rs, string rt, string rd, string shamt, string funct);`

This function returns a concatenated string of machine code based on the R-type instruction parameters. The op parameter refers to the opcode, and the shamt parameter refers to the shift amounts. In short, this function is mainly used for simplicity. This function is called to concatenate that information in parameters to become one string, machine code.

6. `string i_instruction(string op, string rs, string rt, string immediate);`

This function returns a concatenated string of machine code based on the I-type instruction parameters. Similar to r_instruction function, this function is mainly used for simplicity. This function is called to concatenate that information in parameters to become one string, machine code.

7. `string j_instruction(string op, string address);`

This function returns a concatenated string of machine code based on the J-type instruction parameters. Similar to both r_instruction and i_instruction functions, this function is mainly used for simplicity. This function is called to concatenate that information in parameters to become one string, machine code.

8. `void machine_code(vector<string> &ret, vector<string> argument, int line_number);`

This procedure fills the vector of strings with a machine code result based on the instruction argument.

9. `bool character_in_string(char c, string s);`

This function returns a boolean expression whether a character c is contained inside of a string s.

10. `string frag(string argument, string divider, int index);`

This function returns the index-th fragged string from the original argument. The frags are divided based on the custom divider. This function is mainly used to extract the information from `immediate(rs)` to `rs` and `immediate`.

11. `string compare_files(string output_file, string expected_output_file);`

Lastly, this function returns a string expression of the equality of two files by comparing each of the characters contained between two files. Though, this function is only used for the testing experience. However, this function provides simplicity for a tester to check the comparison between two files without any additional program.

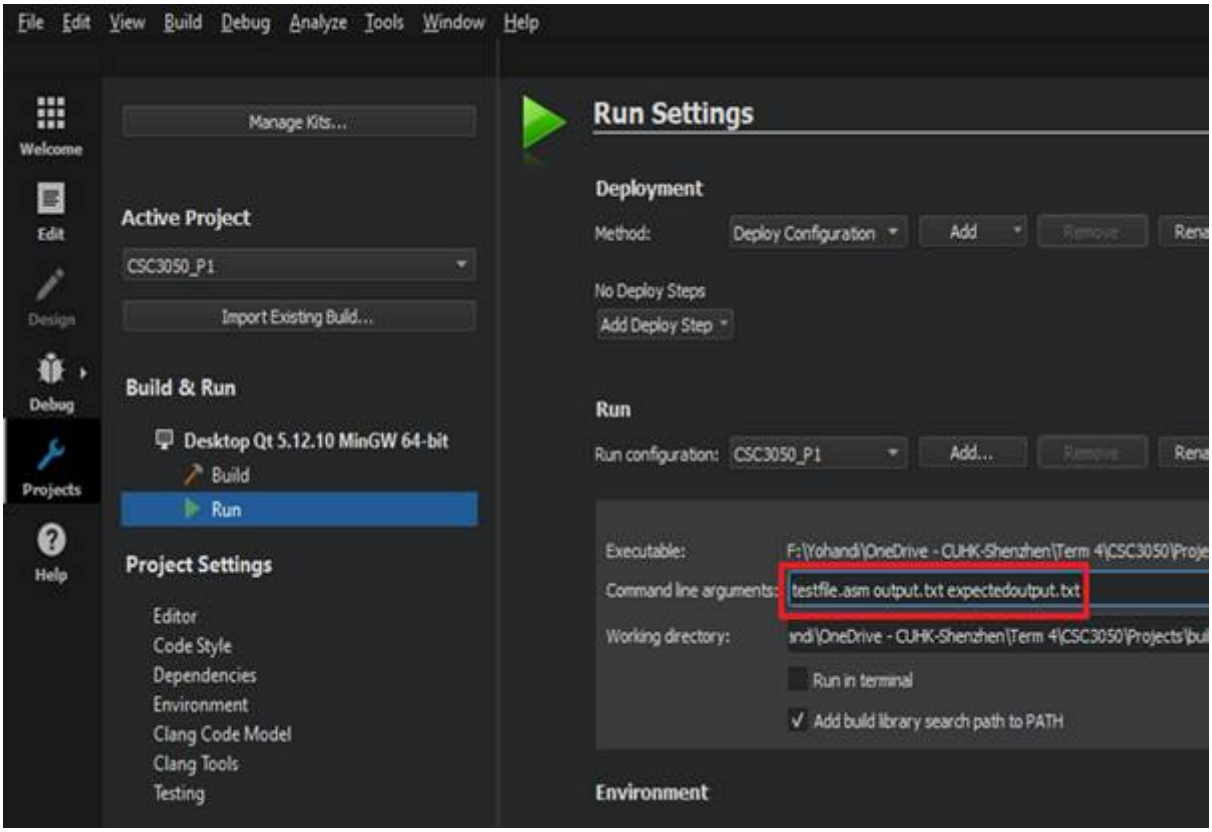
Output

Running the Project

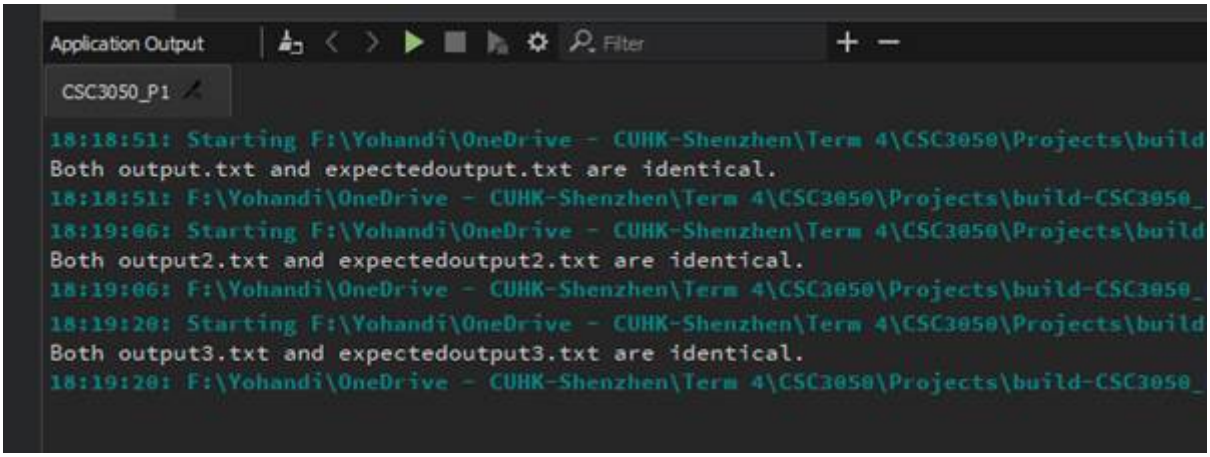
This project requires both the Qt Creator application and the CMake application to run. From the given .zip, there is a folder containing all required source codes and a CMakeLists.txt file to determine the main compile for execution.

After successfully compiling, another folder with a `build` name as its prefix will occur. Tester is required to employ the some considered `.asm` and `.txt` files as the test cases in the build folder.

Tester is required to specify three additional arguments after `.exe`, additional arguments after that are ignored. Those three arguments are the input file, output file, and expected output file. For window users, simply find the executable program in the build folder and run it in the command prompt with `CSC3050.exe testfile.asm output.txt expectedoutput.txt` command, where the `testfile.asm` refers to the MIPS code file, the `output.txt` refers to a new file that is to be printed, and the `expectedoutput.txt` refers to the expected output of machine code translated from `testfile.asm`. Similar to window users, linux users are also able to run it in the terminal with the same command along with the `./` as its prefix. However, some problems occur for windows users where the executable program launches an error message stating that it requires to be reinstalled. For this problem, please run it from the Qt Creator application directly. First, open the Qt Creator application and choose this project. Then, in the projects tab, find the run tab and fill the command line arguments starting from the input file until the expected output file (note that the name of the executable program should not be stated).



After successfully running the program, a console will pop up and will tell the information about the comparison between the output file and the expected output file.



The console results are obtained from the `tester.cpp` file. Though creating a custom tester file is not necessary; however, this is used to automatically compare between two files. If the tester is decided to use his/her own tester, it is definitely fine. Please note that the tester should take the obtained output results from the build folder. In short, the tester file will not interfere with any process inside. It also does not interfere with the output result.

Expected Error and Solution

Some common errors/mistakes from the users could be prevented in this project. Specific custom information will be displayed in the error console based on the mistakes in the interface usage. The existence of the custom information means the author (me) expects the error to be coming from the user. The problems are handleable, and the solutions can be found in this part. At least for the custom error information, there is an explanation of what possible things trigger a specific case.

Prior to the explanations, please note that errors that are not displayed in the error console, which means aside from the cerr format, are not expected from the author (me). The most probable cause is the inconsistency of using MIPS instruction. In addition to that, failing to install the CMake application or the Qt Creator application correctly might also be the cause.

[Error] lack of input validation

This means the command line arguments when running the code are not correctly stated. Please make sure that there are three arguments stated in the command line arguments. Those three arguments refer to the input file name, the output file name, and the expected output file name. Refer to the “running project” part on finding where to state the arguments.

[Error] invalid operation occurs

This means the given operation in the input file is not consistent with the given MIPS instruction list. Please make sure that there are not any stated labels in the middle of the instructions.

[Error] unknown operation type occurs, found: [instruction argument]

This means the operation given in the input file is not stated in the MIPS instruction list. Please make sure that the operation is contained and stated in the MIPS instruction list. Else, this could be an author’s (my) fault for missing some specific operations.

As a side note, any mistake in the input file might result in some unexpected errors such as index access errors and might not be handleable. Please make sure that every instruction given is in the correct format according to the MIPS instruction list.