CSC3050 Assignment1 report (120090211)

Overview:

This C++ program aims to translate MIPS instructions into its corresponding machine code. There are 3 inputs for the programs, including the '.asm file', 'output.txt', and 'expectedoutput.txt'. '.asm file' is the given MIPS instructions. 'output.txt' is the name of the output file, containing binary codes. 'expectedoutput.txt' is the file used to compare with the output file to see whether the program has performed correctly.

Processing Logic:

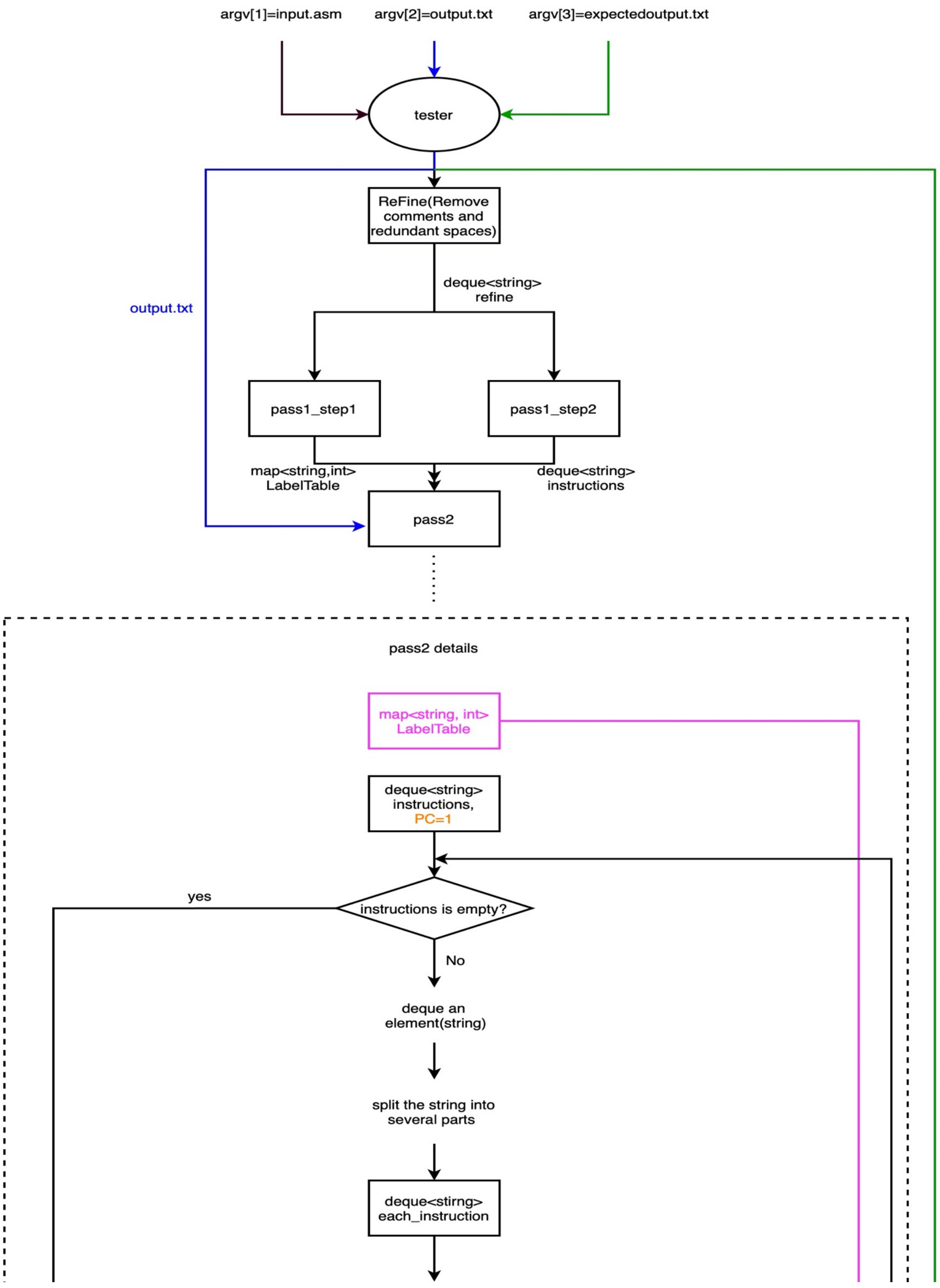
To solve the problems, I divide the whole program into 2 parts, and all the functions needed in these two parts are written in phase1.cpp and phase2.cpp respectively.

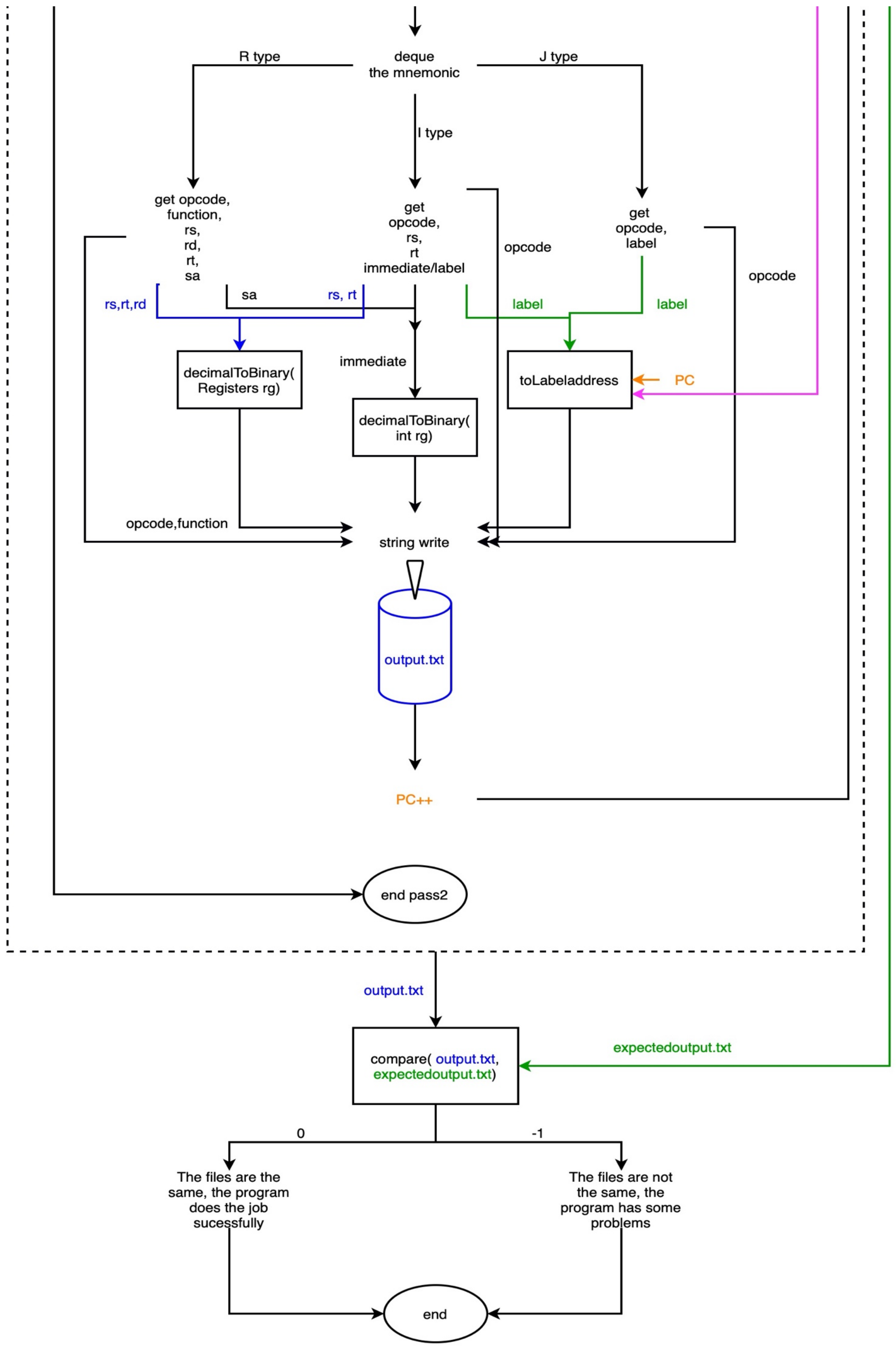
1) phase1.cpp:
   In this part, the goal is to refine the input file and generate the label table. The program will read the information in the .text part, discarding all complements as well as redundant indentation and spaces in the input file. After refinement, only labels and instructions are retained. Next, the program will distinguish labels from the remaining instructions and generate a LabelTable, which will be used in phase2.cpp.

2) LabelTable:
   LabelTable is generated in the last step of phase1.cpp. It records the order of occurrence (denoted as x) of the instruction targeted by the labels based on a map<string, int>. For example, x=0 represents the first instruction of the whole program. In I type, (PC+1-x)*4+0x400000 will be used to calculate the immediate number. In J type, 4*x+0x400000 will be used to represent the address of labels.

3) phase2.cpp:
   In this part, the program will read the instructions one by one with the label table generated from phase1.cpp to generate the corresponding machine code. Firstly, a line of instruction is picked. Secondly, it will be split into several parts. By recognizing the mnemonic of each instruction, the program will decide how it will combine the remaining parts to generate the binary codes. Particularly, if a label appears in the instructions, the LabelTable will be used to find the wanted address.

4) flow chart (next page)


argv[1]=input.asm
argv[2]=output.txt
argv[3]=expectedoutput.txt
tester
ReFine(Remove comments and redundant spaces)
output.txt
deque<string> refine
pass1_step1
pass1_step2
map<string,int> LabelTable
deque<string> instructions
pass2
pass2 details
map<string, int> LabelTable
deque<string> instructions, PC=1
yes
instructions is empty?
No
deque an element(string)
split the string into several parts
deque<stirng> each_instruction



deque the mnemonic
R type
I type
J type
get opcode, function, rs, rd, rt, sa
get opcode, rs, rt immediate/label
get opcode, label
opcode
opcode
rs,rt,rd
sa
rs, rt
label
label
decimalToBinary(Registers rg)
immediate
decimalToBinary(int rg)
toLabeladdress
PC
opcode,function
string write
output.txt
PC++
end pass2
output.txt
compare(output.txt, expectedoutput.txt)
expectedoutput.txt
0
-1
The files are the same, the program does the job sucessfully
The files are not the same, the program has some problems
end

Implementation details:

- Header file:

mainlib.h:

C++ STL library; all function prototypes, and a class used to define Registers.

- Functions:

tester.cpp:

1. int compare_files(string filename1, string filename2)

input: filename1—the output file
 filename2—the expected output file
output: 0—If two files are the same
 -1—If two files are not the same

2. map<string, Registers> setRegmap()

The setRegmap function initiates a <string, Registers> map when initializing the whole program, to translate a register written in string type into its value, which will be used to calculate its binary expression in phase2.

3. int main(int argc, char* argv[])

open the input files, output files, and expected output files.
Examine whether they are successfully opened.
declares the LabelTable, instruction queue
translate the MIPS instructions into machine codes by using the function in phase1.cpp and phase2.cpp
write the binary codes into the output.txt
compare the output.txt and expectedoutput.txt and tell the result to users.

phase1.cpp:

1. deque<string> ReFine(string filename)

Input: name of the input file.
Output: a queue containing the instruction and labels after refinement
This function will read the .text part line by line. For each line, the program will examine whether it is a comment or not and discard redundant spaces and indentation at the same time.

2. deque<string> pass1_step1(deque<string> filename)

Input: a queue containing the instruction and labels after refinement.

Output: a queue containing only the instructions without labels.
The function will move out the labels by recognizing the ':'. It will return a queue without labels.

3. map<string, int> pass1_step2(deque<string> filename)

Input: a queue containing the instruction and labels after refinement.
Output: a map<string, int> containing all the labels in the input file and its corresponding address.
The function will collect all the labels by recognizing the ':'. The function is essentially a loop with a PC recording the number of iterations. Whenever a label is read, it will store a key-value pair with label name as the key and PC as the value.

phase2.cpp:

1. string decimalToBinary(enum Registers rg)

Input: The register (rs, rt, rd, decimal) in R type or I type MIPS instruction
Output: The corresponding number in the binary system.
The function will turn the register written in each line of instruction into its binary form. It will be added to make up the 32 bits machine code with other parts.

2. string decimalToBinary(int rg)

Input: The immediate number/ Sa number (decimal) in R type or I type MIPS instruction
Output: The corresponding number in the binary system.
The function will turn the immediate number or sa number in sll, sra, srl into its binary form. It will be added to make up the 32 bits machine code with other parts.

3. string toLabeladdress(int rg)

Input: the address of the label in J type MIPS instruction (decimal)
Output: The corresponding number in the binary system.
The label has already turned into its address stored in LabelTable before being input into the function. This function only focuses on turning the address from the decimal form into its binary form.

4. void pass2(deque<string> instructions, string filename, map<string, Registers> regmap, map<string, int>ltp)

Input: deque<string> instructions made in pass1_step1 function; output.txt; map<string, Registers> regmap made in setRegmap function when initializing the whole program; map<string, int> LabelTable made in pass1_step2 function
No output
This function will read all the instructions stored in the instruction queue one by one until there is no element in the queue. Each instruction will be split into several parts, by recognizing ',' or ' '. The program will decide the way it combines all the parts based on the mnemonic, the first part of the MIPS instruction. After turning

all the parts into their corresponding binary code and combining all of them, this line will be stored in the output.txt.

To be more specific, here is an example.
“add $t1, $t2, $t3” is read from the queue.
It will be split into “add”, “$t1”, “$t2”, “$t3”. The “add” is recognized first, so opcode will be fixed to 000000, sa will be fixed to 000000 and function will be fixed to 100000. $t1 will be recognized as rd and will be turned to 01001 by decimalToBinary, and $t2, $t3 in the same way. Then all of those separate parts will be combined in the form of opcode+rs+rt+rd+sa+function to make a string. At last, the string will be added to the output.txt

● Data Structure:

1. PC (pointer counter)

An integer. Every time the program finishes translating an instruction, the PC will add 4, which means that it moves to the next instruction.

2. LabelTable

The LabelTable is based on a map<string, int>. The keys in this map are the names of the label, and the value is the address of the label.

3. Registers

A enum type containing 32 different registers. The value of the register is the same as its register number. For example, $gp has the value of 28.

4. Instructions

A string queue. Every element in the queue is a string of a complete MIPS instruction. For example: “add $t1, $t2, $t3”