Project Report

CSC 3050 MIPS Assembler

Wei Wu

118010335

February 29, 2020

The School of Science and Engineering

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

1. Understandings of the Project

This project is about writing an assembler for the MIPS assembly language. Briefly speaking, the assembler takes as input an MIPS assembly language file. After assembling, it generates an output file which is binary and "executable". To be more specific, the assembler "translates" each line of instruction into a 32-bit machine code.

2. The Implementations of the Project

2.1.The Big Picture Idea

In MIPS assembly language, each line of instruction corresponds to a 32-bit binary code. Therefore, it is feasible to "translate" the instructions line by line.

First of all, the register information and the instruction information (*) are predefined. Due to the existence of labels, the assembler firstly scans through the whole file for all the labels. Whenever a label is found, the assembler stores both the name and the memory address of the label. If this step is skipped, when the target line of a label is after the current line, the assembler can never know the address of the label.

After that, the assembler scans through the file for the second time. The file is processed line by line. For each line read, the assembler identifies both instruction and arguments. With the register information and instruction information predefined as well as the label information generated, the assembler generates the

corresponding machine code.

(*) Some information is fixed. For example, the name and number of each register, as well as the opcode, name, and type (shamt, funct, rd, rs, rt, if applicable and fixed) of each instruction.

2.2. Problems and Solutions

2.2.1. How to compile a C++ program using command lines?

Before this project, I always use IDEs to compile C++ programs. After searching on the internet, I found out that compilation can be done in the terminal. For example, using a command line of the form “g++ -std=c++11 xxx.cpp -o assembler”.

2.2.2. How to get the input filename and the output filename?

In C++, it is feasible and convenient to get the input and output filenames from the command line. To be more specific, two parameters, int argc and char** argv, are added to the main() function. This way, after compiling, the user can use a command line of the form “xxx input.txt output.txt” to specify the input and output filenames. Here, “input.txt” corresponds to argv[1], while “output.txt” corresponds to argv[2]. Moreover, since there are always one input filename and one output filename, argc should always be 3. Otherwise, an error message will be generated.

2.2.3. How to translate register names to register numbers?

In assembly language, the registers are denoted by names such as “$v0” and “$s0”. To generate the corresponding machine code, the corresponding register numbers must be found. Therefore, in the program, a vector<string> is used to store the register names in the same order as the register numbers. Then, a map<string, int> is constructed. By referring to this map with register names, the program can get the corresponding register numbers.

2.2.4. How to store the name, type, opcode (, funct, shamt, rt, rs, rd, if applicable and fixed) of each instruction?

To generate the machine code, much information about the instructions is needed. A structure called Instruction is called. It stores the name, type, and opcode of each instruction. Moreover, for R-type instructions, the function is stored, while for some I-type instructions, the fixed rt is stored. After that, all instructions are stored in a vector<Instruction>. In some instructions, there is fixed funct, shamt, rt, rs, and (or) rd. To make the structure clearer, these information are stored in the corresponding functions.

2.2.5. How to output the machine code?

To be more specific, the program needs to have the ability to print a signed binary code whose value and number of digits are fixed. Therefore, a function called printBinary() is designed. Firstly, it remembers the sign of the value and gets the absolute value. Then, if the value is negative, it calculates the 2’s complement with the absolute value. Lastly, it checks every bit of the number

in binary from the most significant bit (determined by the number of digits to print). When a bit is 1, it outputs 1, otherwise it outputs 0.

2.2.6. How to determine which lines have no instruction?

In the input file, some lines contain no instruction but whitespaces, labels, and (or) comments. Thus, for these lines, no machine code should be generated. The program splits every line and checks if there is any valid instruction. In particular, if the line contains some instruction that can be found in the collection of instructions (vector<Instruction> Instructions), then it contains a valid instruction.

2.2.7. How to identify and store the labels?

The program first scan through the whole input file line by line. A variable line_number is used to track the address of each instruction. Whenever there is a colon (:) at the beginning of a line, there is a label. Then the program stores the name and address of the label in a map<string, int>. Since some lines contain no instruction, the variable line_number increases by 1 only when the current line contains a valid instruction.

2.2.8. How to reset the input stream after the first scanning?

Say “input” is the ifstream. We can use input.clear() and input.seekg(0) to reset the input stream.

2.2.9. How to get the 32-bit machine code for each instruction?

Generally, there are 3 types of instruction: R, I and J type. To make the code clearer, I implemented one function for each type of instruction. However,

within a single type, different instructions could have different parameters. Both the number and the order of parameters can be different. Therefore, I sort the instructions of the same type firstly by the number of parameters (1, 2, and 3), and then by the order of parameters (rt, rs, rd, offset, etc.). Using this kind of grouping, the length of the code is reduced.

2.2.10. How to deal with branch (or jump) instructions that have no label but numerical offset (absolute address)?

Whenever a branch (or jump) instruction is detected, the program firstly gets the argument, and then refers to the map of label for it. If the argument is an existed label, then find out its address and calculate the offset (target). If the argument is not an existed label but a number, then consider it as the offset (target). Otherwise, generate an error.

2.3. Data Structures

2.3.1. const vector<string> REG:

The collection of register names.

2.3.2. static map<string,int> regMap:

The map from register names to register numbers.

2.3.3. struct Instruction {

```
string name;
string type;
```

```
        int opcode;
        int function;}:
/* Notes for int function:
     * For R-type instructions, this variable stores "function"
     * For some I-type instructions, this variable stores "rt"
     * For other instructions, this variable is set to 0 and is
meaningless.
     */
```

The data structure that stores the name, type, opcode and function (rt) of a MIPS instruction.

2.3.4. static vector<Instruction> Instructions

The collection of MIPS instructions.

2.3.5. map<string, int> labels

The map from label names to label addresses (line numbers).

2.4. Functions

2.4.1. int main(int argc, char** argv)

This function takes the input filename and the output filename from command lines and calls assembler() to start the work.

2.4.2. void constructRegMap(map<string,int> & regMap)

This function constructs the Register Map (Name -> Number), which is a global

variable.

2.4.3. int regNameToNum(string name)

This function takes as input a string, which contains a register name. The input can be of the form either "$s0" or "100($s0)". After referring to the Register Map, it returns the corresponding register number.

2.4.4. void printBinary(int num, int digit, ostream & output)

This function is used to print a signed binary number. The parameter num is the number to print, digit is the number of digits to print, and output is the output stream.

2.4.5. map<string, int> scanForLabels(istream & input)

This function scans through the whole input file for all the labels, remembering each label and its address (line number).

2.4.6. string getInstruction(string & line)

This function takes as input a line and returns the instruction. Meanwhile, it cuts off the instruction from the original line for further oprations.

2.4.7. vector<string> split(const string &str, const string &sign)

The function splits a string with respect to the given sign, and then returns a vector containing the substrings.

2.4.8. void trim(string & str)

This function takes as input a reference to a string, and erases all whitespaces, '\f', '\v', '\r', '\t', and '\n' at the beginning of and the end of the string.

2.4.9. void assembler(istream & input, ostream & output)

This function takes as input an input stream and an output stream. It firstly scans through the whole input file for the labels, and then reads and translates the input file line by line.

2.4.10. void rType(Instruction ins, vector<string> args, ostream & output)

This function takes as input an R-type Instruction, a vector of arguments and an output stream. It determines the opcode, rd, rs, rt, shamt, and funct for an R-type instrucion. Then, it calls printBinary() to output the machine code.

2.4.11. void iType(Instruction ins, vector<string> args, ostream & output, int line_number, map<string, int> labels)

This function takes as input an I-type Instruction, a vector of arguments, an output stream, a line number and a label map. It determines the opcode, rt, rs, and imm (offset) for an I-type instrucion. Then, it calls printBinary() to output the machine code

2.4.12. void jType(Instruction ins, vector<string> args, ostream & output, map<string, int> labels)

This function takes as input a J-type Instruction, a vector of arguments, an output stream, and a label map. It determines the opcode and target for an J-type instrucion. Then, it calls printBinary() to output the machine code.

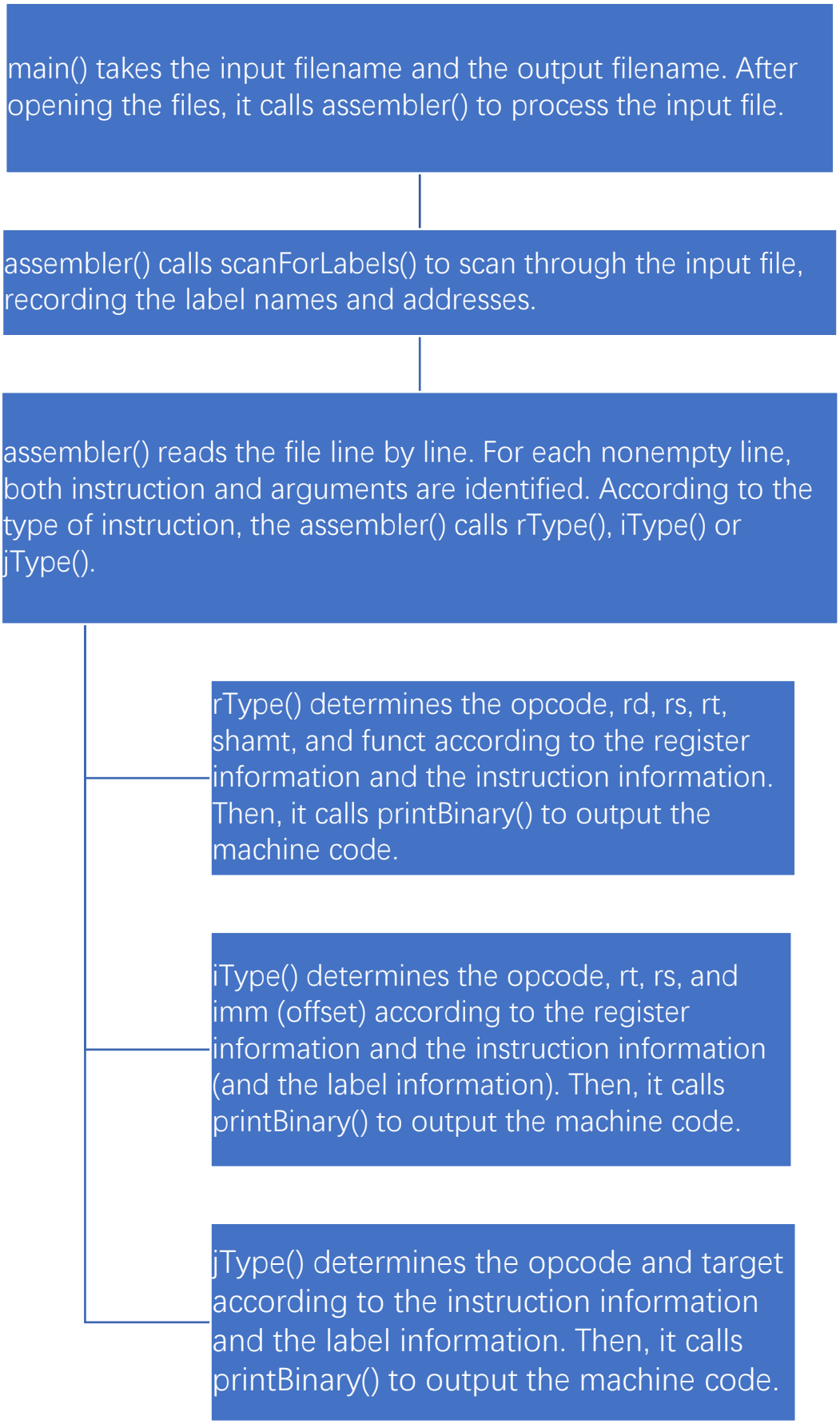
2.4.13. bool isInt(string arg)

This function takes as input a string, and returns whether the string is an integer.

2.5. Discussion & Further Improvement

In this project, I mainly focus on how to translate the MIPS assembly language into machine code. If the input assembly language code is totally correct, the program runs nicely. However, in real life, there might be all kinds of errors in the input file. To make things easier, the assembler should be able to identify the types and locations of errors. In this project, I did write some code to identify some of the possible errors, but not all of them. Moreover, I did not give the locations of them. Thus, to make the assembler more robust, I think I should implement more of error detection.

2.6. Simplified Procedures

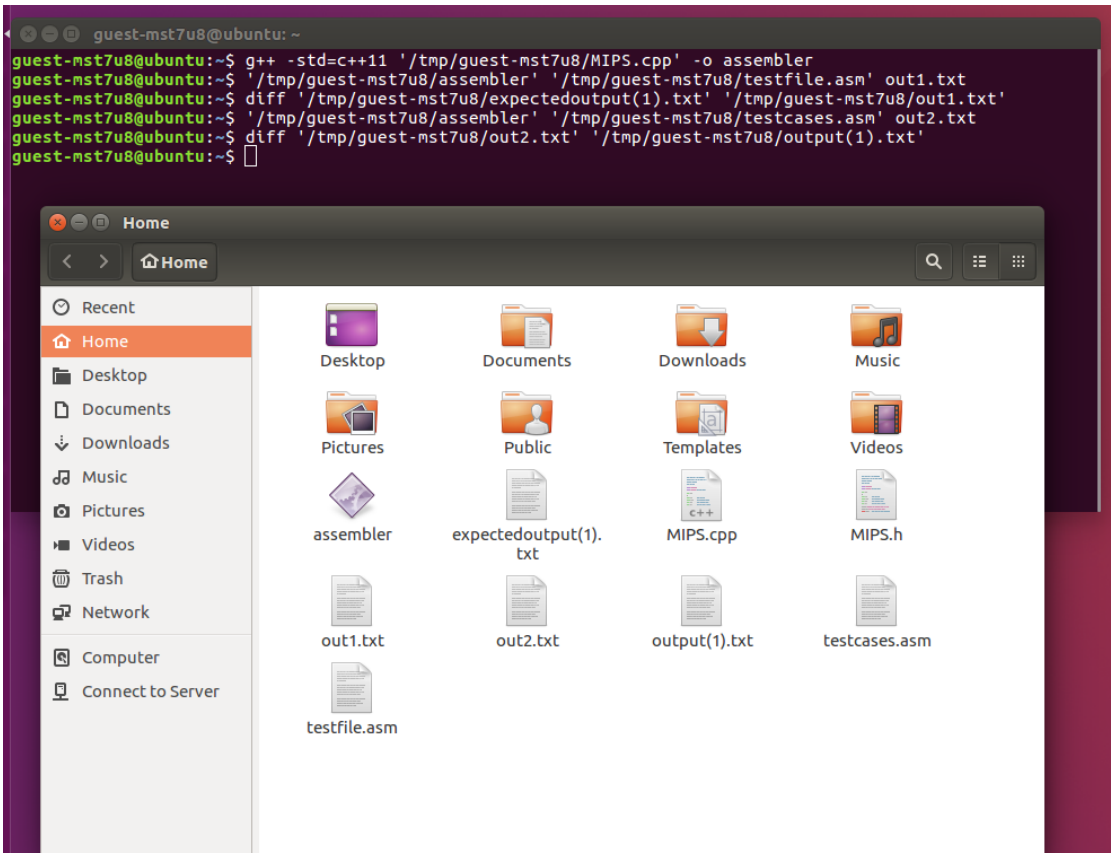


3. Testing

3.1. Environment:

Ubuntu 16.04
g++
C++11

3.2. Result:



Note:
Test input 1: testfile.asm
Test output 1: out1.txt
Sample output 1: expectedoutput(1).txt

Test input 2: testcases.asm
Test output 2: out2.txt
Sample output 2: output(1).txt