THE CNINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC3050

Computer Architecture

Report for Project 1

Author:
Li Jingyu 李璟瑜

Student Number:
118010141

March 2, 2020

Contents

1 Understanding

This is a project to design an MIPS assembly language assembler using C/C++. Since we are learning C++ at the same time in CSC3002, I use C++ to finish the project.

MIPS is a kind of low-level language that connects software and hardware of the computer. It contains many instructions as the input to the hardware. These instructions control the registers, memory access, I/O, and so on. Computer works according to the instructions.

However, the computer can only recognize binary number (0,1) when it is running. The readable instructions should be translated into binary form at first. So out task is to write an assembler that transfers the MIPS assembly language into machine code, which is executable by the computer.

There are three kinds of formats for the instructions – R, I and J. They are all 32-bit, with similar structures and combination rule. For example:

add $s0, $s1, $s2 – 00000010001100101000000000100000

This is R-format: opcode(6) + rs(5) + rt(5) + rd(5) + shamt(5) + funct(6)

addi $s0, $s1, -100 – 00100010001100001111111110011100

This is I-format: opcode(6) + rs(5) + rt(5) + imm(16)

jal label - 00001100000000000000000000000000

J-format: opcode(6) + target(26)

For our project, we need to decide which format the instruction belongs to. Then according to the structure and combination rule, we translate the information into binary form and fit it in the certain location.

More specifically, we receive an input file of MIPS code, do the translation, and output the file of machine code at last.

2 Logic and Details

The general logic of my program is nearly the same with that specified in the project instruction.

1) **First scan**: scan through the whole file, store the labels (texts between the :) and their corresponding address (usually the sequential number of line), filter the comments (texts after #) and blank (space, \t), generate an immediate file (whale.txt) which contains only the effective instructions.

   (e.g. Original file: R: add $s0, $s1, $s2 #r instructions)

   (e.g. whale.txt: add $s0, $s1, $s2)(No blank at the front or at the back)

2) **Second scan**: scan through the output file of the first scan (whale.txt), read line by line, first split the instruction into different parts (token=, ()\t)(comma, space, parentheses, \t), take the first element (the operation name e.g. add), judge its format, get its corresponding sequential number (defined by me) (e.g. add-0,addi-32), according to the format, transform the remaining elements ($s0 $v1 $t2 R -100 1234567)(register name, label, immediate value, address, offset, and so on) into binary form, and store them in a vector<string>:

   R – The remaining elements are either register name or shift immediate. If it is register name, then find its corresponding binary number string (defined in a constant map); else if it is shift immediate, change the integer into 5-bit binary number string. Put these binary strings into the vector.

I – The remaining elements can be register name, label name, or integer (immediate of offset). If it is register name, deal it with the same way as above, else if it is a label name, first get the current line number, then get the label address, do the subtraction (Here the line number should add 1 because of the PC counter: PC = PC + 4;), and get the offset. Change the offset into 16-bit binary number string, else if it is an integer (immediate of offset), just directly change it into 16-bit binary number string (there is a function to realize this). Put these binary strings into the vector.

J – The remaining element (only one, named target) can be label name or integer (absolute address). If it is label name, find the corresponding address(line number) in the map, just change it to 26-bit binary. Else, change the integer to 26-bit binary string. Put it into the vector.

An example: Original instruction code: add $s0, $s1, $s2

The argument: 0 <"10001"," 10010"," 10000"> (<- This is the binary number of the three registers.

3) **Return code**: we get an integer representing the operation name and a string vector containing the relative binary string from the second scan. Put the two arguments to the functions: ReturnRcode / ReturnIcode / ReturnJcode. In these

functions, they will switch to the relative operation according to the integer instruction representation, and give values to the certain locations (opcode, rs, rt,imm……) using some values of the string vector. The mapping rule is regulated by the textbook. For example, the opcode for R format is "00000". The function will return a complete 32-bit machine code (binary string) for each instruction.

4) **Output**: Line by line, put the 32-bit machine code into the output file. Generate a new output file with arbitrary name.

The reasons why the logic looks like this are: Two scans – one to store the label-address map and filtration, one to translate, the division is clear. If there is only one scan, there will be some bugs (latter label, instruction split…) ; Immediate file: easier to read and translate ; Three formats: Convenient to handle, avoid mistakes…

All in all, the general logic should go like this because it is simple and efficient and I can only think of this way.

Header File: (can jump)

#include "stringOperation.h":

The file includes some functions to deal with the string (texts).

```
void DeleteBlank(string& text); // Delete the space or \t of the string (texts) at the front and at the back. Used to filter the instruction.
string ChangeDecToBinStr(string& num,int& bit); // Change the decimal string to binary string
```

with certain bit

vector<string> **SplitString**(const string& str, const string& token); // Split the string by the token (can have multiple tokens)

#include "RegisterAndInstructions.h"

This file defines some constants and map: The number of instructions of each format (32,36,2). Three arrays which contain the name of the operations. map<string, string> **RegBinMap**()a map from register name to its binary string (e.g. “$s0”-“10000”)

A function:

int **JudgeInsNumber**(string& Operation) // Judge the sequential number of the instruction e.g. add -> 0; addi -> 32;

#include "scanAndOutput.h"

This file is the most important. It implements the two scan process. It defines some items the same as the "RegisterAndInstructions.h". Functions:

map<string, int> **FirstScan**(string& Ifile, string& Ofile) // As the name suggests and explained above, return a map from the label to its address

void **SecondScan**(string& Ifilex, string& Ofilex, map<string, int>& labelAndAddress,map<string, string>& regToBin) // As the name suggests and explained above, output a file of the machine code.

string **ReturnRcode**(int& insNumber, vector<string>& regis) // Return the R-format code according to the instruction number and other instruction elements of binary form

There are also ReturnIcode and ReturnJcode, omitted here.

Some details:

In the ReturnR/I/JCode function, the sentence of switch is used (the argument is the integer representing the operation name), and it can go to that operation directly, which I think it is faster than using many if/else sentences.

At first I want to write a class to represent the three formats. However, I think it will make the problems more complex, which is “OK, BUT NOT NECESSARY”.

Something strange: \t inside the instruction. slt \t …

There are also detailed comments in the program.

3 How to run the program

Just the same way as the instruction specifies, but adding: –std=c++11.

```
g++ -std=c++11 main.cpp RegisterAndInstructions.cpp scanAndOutput.cpp stringOperation.cpp -o assembler
./assembler inputfile.asm outputfile.txt
```

The input file should be in the same location as the “assembler” file. Name of the output file can be set arbitrarily.

By the way, the program will generate an immediate file called “whale.txt”, which stores the instruction being filtered. The file can be ignored. But if you have interest, you can have a check ☺.

4 Difficulty

Debugging:

1) location of the {} of a for statement. Wrong alignment of the } (lack one). The bug costs me 40 minutes to fix…
2) typo mistake. “01110” - “001110” for tnei instruction.
3) redefinition of the constant (adjust the header file)/ undefined variable (add extern)
4) Wrong output format: delete the \t and space at the front and at the back
5) Some decimal numbers in the output: did not consider the negative numbers, should use 2’s complement.

……

Header file:

It costs me a lot of time to create header files, split a long main function into different small functions and copy them to different header files.

Header files should be divided by logic (which is taught by TA Handsome)

Easier to write, check and maintain. It will also make other companions happier to work with you.

Unfamiliar with Linux:

It costs me some time to download Ubuntu 16.04 and VMware and test the code in the Linux. I don’t know how to transfer a file from Windows to Linux (I have used the qqmail and an USB). Still need time to get familiar with Linux.

The number of instructions:

70 instructions, kind of time-consuming and hard work

5 Feeling

In a way, I gain a lot and make some progress in this project.

C++ is different from Python in terms of simplicity. It is more difficult to implement some useful functions in C++ than in Python (e.g. split function). Python is really easy and friendly. We need to consider more in C++.

Since I am new to C++ and not familiar with it, I need to search many basic knowledge of the programming language, such as the usage of vector and map, how to define a class, global variables, references…

Though the project itself is not so difficult in terms of logic or algorithm, it still costs me a lot of time in learning new knowledge and dealing with some details.

I have asked TA Handsome for some help. He was warm-hearted and willing to help, which moved me, ahahaha.

Actually TA Handsome was the tutor of CSC3001 last term, so I knew him. He taught well.

I also learned a lot from the discussion board on bb and wechat group (sometimes very interesting).

That’s all.