The Chinese University of Hong Kong, Shenzhen

CSC 3050

Computer Architecture

Report for Project 1: MIPS Assembler

Author:
Qiu Weilun

Student Number:
120090771

March 1, 2022

Contents

1 Introduction

The Instruction Set Architecture (ISA) is part of the abstract model of a computer which determines how the CPU is controlled by computer software. It functions as an interface between the software and hardware. There are two types of ISA, which are Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). The ISA used in this project is a kind of RISC instruction set which is called the MIPS architecture.

For most of the high level programming language like C or C++, the program code will be translated into the assembly language first. This process is called compilation. After that, the assembler may assemble the assembly language code into binary machine code for the machine to execute the program. This project focuses on the implementation of a MIPS assembler using C++. The input of the project would be a program file which contains some MIPS instructions. The output would be a txt file of the correponding binary machine code of these MIPS Instructions.

2 Big Picture: How the MIPS Assembler work?

There are three types of intructions in the MIPS architecture. They are the R-type instruction, the I-type instruction, and the J-type instruction. Different types of instructions have different formats.

2.1 R-type Instruction

opcode(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)

Table 1: R-type Instruction's General Format

The R-type instruction consists of 6 components. The opcode field shows the operation code of the instruction. All the R-type instructions have the same operation code which is 000000. The rs and rt fields shows the two source registers used in the instruction. The rd field shows the destination register. The value of the shamt field represents the shift amount in the instruction. For instructions that do not perform shift operation, the value of shamt is 00000. The funct field is unique for all the R-type instructions. It decides what kind of operation is performed in each instruction. **Therefore, a one-to-one mapping relation can be established between the operation name and the corresponding function code.**

The MIPS codes of different R-type instructions have different formats. Therefore, it is important to classify them before assembling. Most of the codes present each

field in the order of operation name, rd, rs, and rt. For example, the representation of add format follows this kind of order:

add $t0, $t1, $t2

In addition, some codes of R-type do not present all the field such as the jr, the mthi and the mtlo operation which only show the operation name and the rs field:

jr $t5

In this case, **the instructions will be classified into different classes according to their different ways of code representation.** All the missing field will be set to 0. Some code may display all the fields of a R-type instruction, but they do not follow the order as mentioned above. For instance, the code of sllv operation follows the order of operation name, rd field, rt field and the rs field. They will also be classified into another type.

2.2 I-type Instruction

opcode(6 bits)	rs(5 bits)	rt(5 bits)	immediate(16 bits)

Table 2: I-type Instruction's General Format

The I-type instructions are similar to the R-type instruction. The difference is that they have the immediate field instead of the shift amount and the function code. The opcode corresponds to the operation of the instruction. **Therefore, a one-to-one mapping relation can be established between the operation code and the operation name.** Most of the I-type instructions' MIPS code follow the pattern of operation name, rt, rs, and then the immediate field.

One of the problems relates to the implementation of I-type instruction is addressing. The addressing mode used in I-type instruction is **Relative Addressing**. It calculates the offset between addresses in this way.

$$\text{offset} = \frac{\text{label\_address} - (\text{current\_address} + 4)}{4} \tag{1}$$

Instructions which refer to addressing problem include beq, bgez, bgtz, blez, bltz, and bne instructions. To follow the rule of relative addressing mode, the address of the labels would be stored in a data structure called map in C++(or dictionary in python). In addition, a variable(pc) will function as a program counter when scanning the file to record current address of the code. In this way, the offset between the target address and the following address can be calculated.

2.3 J-type Instruction

opcode(6 bits)	target address(26 bits)

Table 3: J-type Instruction's General Format

The J-type instructions implemented in this project include the J(jump) and the Jal(jump and link) instructions. The addressing mode used in J-type instruction is **Pseudo Direct Addressing**. The 26-bits target address is generated by discarding the highest 4 bits and the lowest 2 bits of the 32-bits address of the target label. The opcode is determined by the corresponding operation name, which is the same as the I-type instruction.

3 Implementation Ideas

According to the analysis mentioned above, this project will be mainly divided into 3 parts. The first two parts are the phase1.cpp and the labeltable.cpp. The general goal of this two parts is to generate an intermediate file called intermediate.txt which is a simplified version of the original input file. In this intermediate file, there will be no more blankspace in front of each line of codes. The comments and labels won't appear in this file. In addition, a map will be returned in the labeltable.cpp which uses label names in the original file as keys, and the addresses of the first line of codes under the corresponding labels as values. Therefore, some useful functions will be implemented in the phase1.cpp. In the labeltable.cpp, the first scanning of the original input file is conducted. A map and an intermediate file are generated.

In phase2.cpp, the intermediate file generated in the first two stages will be scaned. For R-type instructions, a map is created using operation names as the keys and the corresponding function codes as the values of the map. For I-type instructions and J-type instructions, two maps are created using operation names as keys and operation codes as values. Another map is used to store the information of the registers and their binary code representations. Finally, the corresponding machine code will be generated according to the map generated in the labelTable.cpp.

4 Implementation details

4.1 Implementation of phase1.cpp

In the phase1.cpp, there are 4 functions be implemented. The first function is the **findTextSegment** function which is used to find the text segment in the input

file. All the segments which are in front of the text segment will be omitted. The second function is the **findCommentIndex** function which check whether there are comments in each line. The criteria to judge whether there are comments is to check whether a "#" symbol appears in each line. This function returns the index of this symbol in each line. The third function is the **isLabel** function, which check whether there is a new label appears by looking for ":" symbol in each line. The last function is the **findLabel** function which store the label name and it corresponding address in a structure called "label". The implementation of this structure is in the phase1.h file. The structure is shown as following.

```
struct label{
        std::string labelName;
        int address;
};
```

The the name of the label is stored in the labelName field. The address of the first line of code under the label will be recoreded in the address field.

4.2 Implementation of labelTable.cpp

The input file is scaned in the lebalTable.cpp. The variable(cp) acts as the program counter in the whole procedure. Therefore, the value of the program counter is originally set to be 0x400000 since the instructions are assumed to start from 0x400000 in this project.

The first task when scanning the input file is to find out the ".text" section. Therefore, the first loop is designed to find the text segment. Once it is found, the program will enter another loop. In this loop, each line of code will be checked whether it is an empty line. If yes, then it continue the next loop. If no, then the program started to check whether there are comments in this line and locate where the comment start. After that, the program will also check whether there is a new label appears in the line of code. If yes, then it will write the corresponding information to the label table. The program counter will be deducted by 4 since labels will not occupied any space. Finally, the program write the codes in each line to the intermediate file according to different situations. In addition, at the end of each loop the program counter will automatically be added by 4 in order to keep track of the address of each line of code.

The corresponding logic flow chart of the labelTable.cpp can be found in the Figure 1 of the Appendix part.

4.3 Implementation of phase2.cpp

As mentioned in the Implementation Ideas part, at the beginning of the phase2.cpp, 4 maps are used to stored all the informations. Then the program started to read each line in the file. Firstly, the operation name of each instruction will be recognized and check which types of instruction it belongs to. Then, the program got the information of operation code or function code by looking up the maps implemented before. After that, the program will read and manipulate in different ways according to its format in MIPS language.

Instruction	Format in MIPS Language
add, addu, and, nor, or, slt, sltu, sub, subu, xor	rd, rs, rt
div, divu, mult, multu	rs, rt (rd = 00000)
jalr	rd, rs (rt = 00000)
jr, mthi, mtlo	rs (rt = 00000, rd = 00000)
mfhi, mflo	rd (rt = 00000, rs = 00000)
sll, sra, srl	rd, rt, sa (rs = 00000)
sllv, srav, srlv	rd, rt, rs
syscall	NAN (rd = 00000, rs = 00000, rt = 00000)

Table 4: R-type Instruction's Classification

Instruction	Format in MIPS Language
addi, addiu, andi, ori, slti, sltiu	rt, rs, immediate
beq, bne	rs, rt, label
bgtz, blez, bltz	rs, label (rt = 00000)
bgez	rs, label (rt = 00001)
lb, lbu, lh, lhu, lw, sb, sh, sw, xori, lwl, lwr, swl, swr	rt, immediate(rs)
lui	rt, immediate (rs = 00000)

Table 5: I-type Instruction's Classification

Table 4 and Table 5 show the classification of I-type and R-type instructions according to their format in MIPS language. The program will perform different operations according to different types in the tables.

5 Appendix

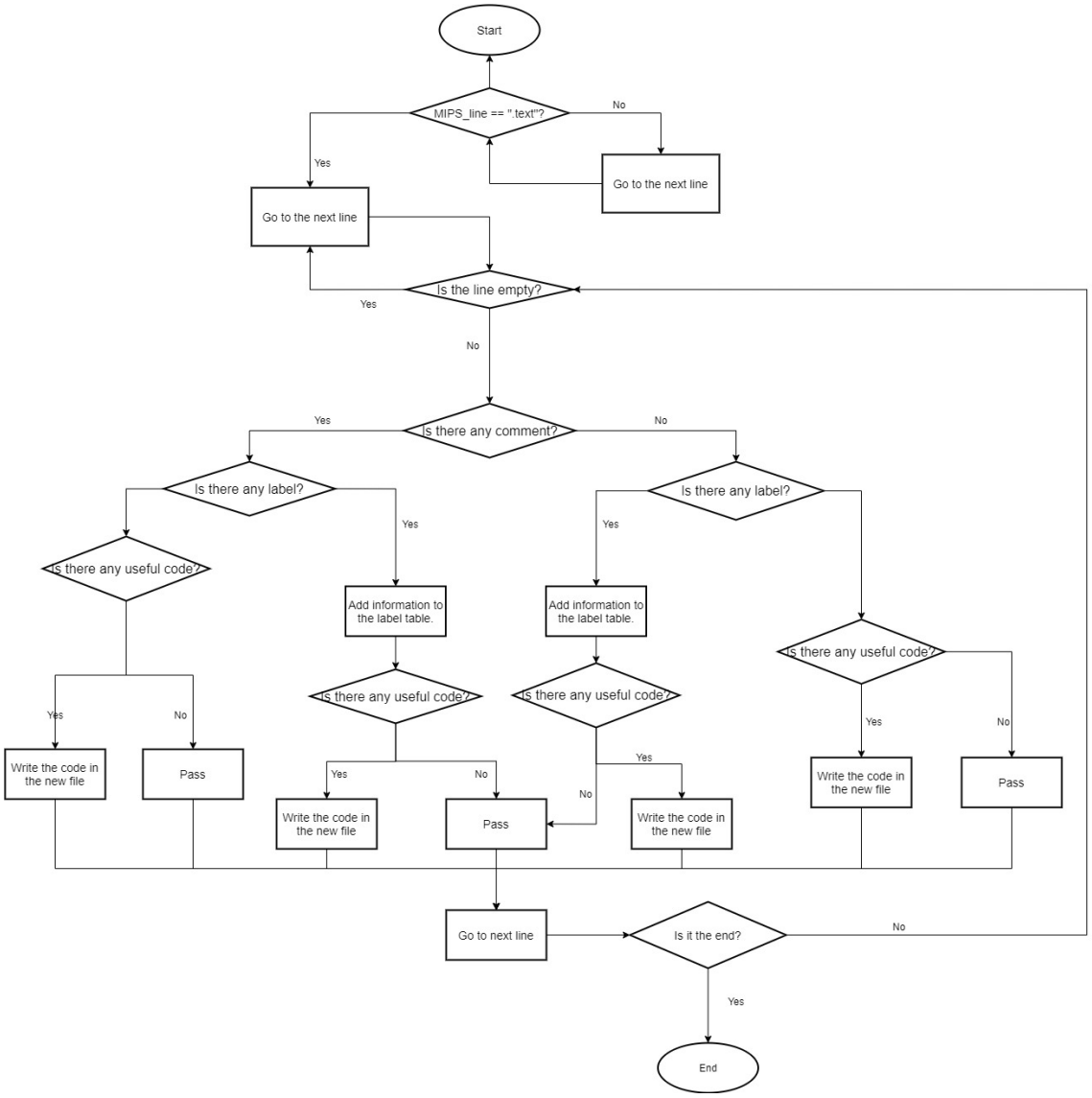

Figure 1: The Logic Flow Chart of the First Scan in labelTable.cpp

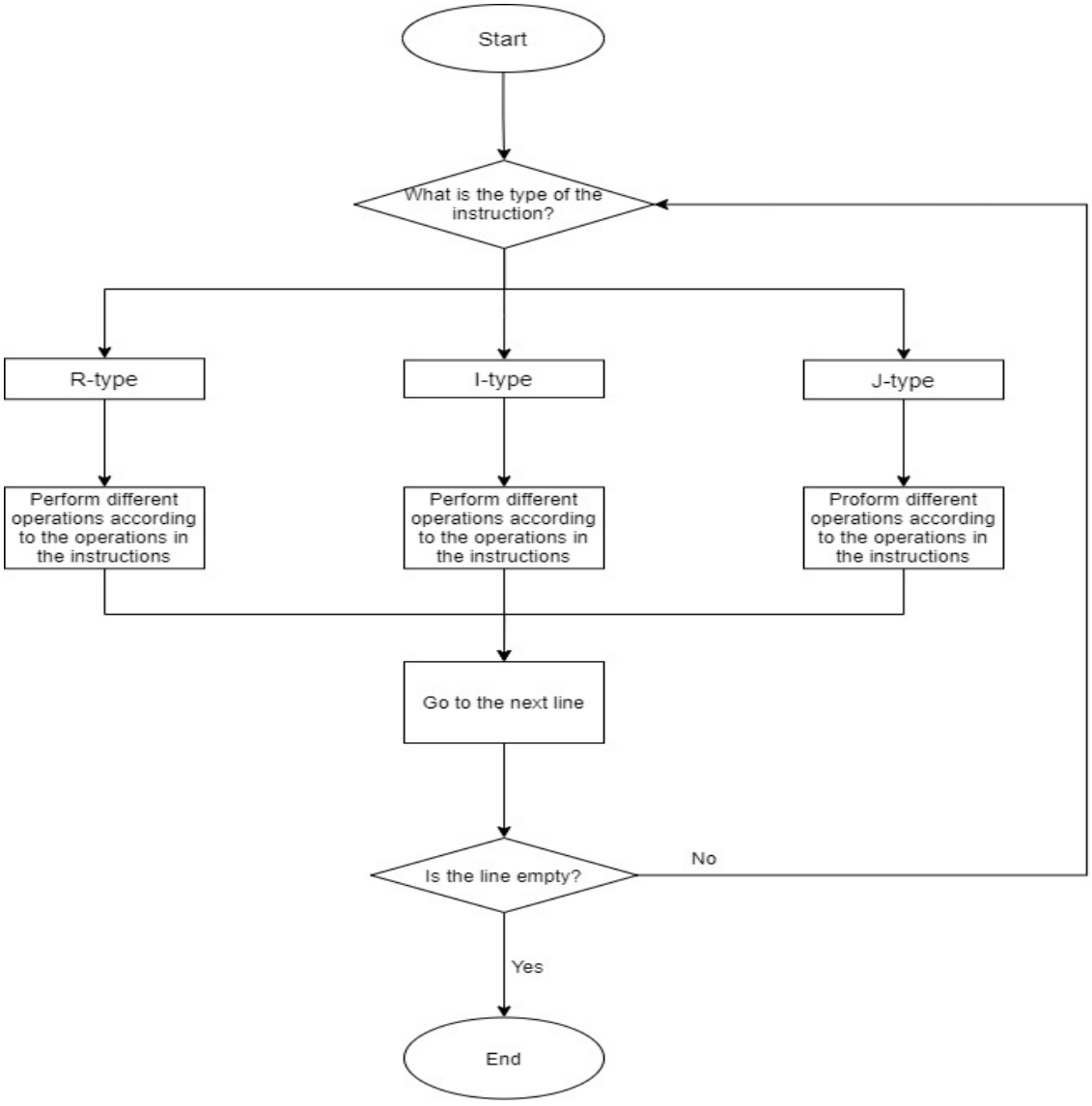

Figure 2: The Logic Flow Chart of the Second Scan in phase2.cpp