Report For Project 1

121090697 Jiayan Yang

1. Overview

This project required us to build a MIPS assembler, which can translate assembly language to machine code. I divided the problem into two parts under the suggestion from the assignment. I use phase1 to go through the whole code of the assembly language first to find out all the labels, and then transfer the code into machine code using phase2. In the following pages I would introduce my code in different parts: tester.cpp, LabelTable.h, LabelTabel.cpp, phase1.cpp, phase2.cpp and, my makefile.

2. Tester.cpp

Since the original tester is in c, I modified it to cpp and make some changes. But the core idea are same.

3. LabelTable.h and LabelTable.cpp

In this file I designed some basic structures and some useful functions.

There are 32 registers in total and they have the fixed order. So in order to find one register's order number in a convenient way, I decided to use a map to store all the registers：

```
 //LabelTable.cpp
 std::map<std::string,int> Reg_list
{
        {"$zero",0},{"$at",1},
        {"$v0",2},{"$v1",3},
        {"$a0",4},{"$a1",5},{"$a2",6},{"$a3",7},
        {"$t0",8},{"$t1",9},{"$t2",10},{"$t3",11},{"$t4",12},{"$t5",13},{"$t6",14},
{"$t7",15},
        {"$s0",16},{"$s1",17},{"$s2",18},{"$s3",19},{"$s4",20},{"$s5",21},{"$s6",22},
{"$s7",23},
        {"$t8",24},{"$t9",25},
        {"$k0",26},{"$k1",27},
        {"$gp",28},{"$sp",29},{"$fp",30},{"$ra",31}
};
```

For a basic command in assembly language, it had its own registers and its own operation code(or function code).So it is easy to think that we can build a structure to store the elements of its own registers and operation codes：

```
//LabelTable.h
struct Inst
{
    char inst_type;//"i", "j", or "r"
    uint opcode;
    uint funct;
    std::string arg[3];
};
//arguments types: rd, rs, rt, sa, imm, imm_rs, label;"" means empty;
```

and then use a map to save all the commands:

```
std::map<std::string,Inst*> Inst_list
{
        {"add",new Inst{'r',0,0x20,{"rd","rs","rt"}}},
        {"addu",new Inst{'r',0,0x21,{"rd","rs","rt"}}},
        {"and",new Inst{'r',0,0x24,{"rd","rs","rt"}}},
        {"div",new Inst{'r',0,0x1A,{"rs","rt",""}}},
        {"divu",new Inst{'r',0,0x1B,{"rs","rt",""}}},
        ......
}
```

There are four functions had been declared in *LabelTable.h* and written in *LabelTable.cpp*. The *del_space()* function is to filter all the spaces or returns in the file, while the *del()* function is to filter annotations and the labels.

The *convert()* function converts any decimal number into binary number, the variable *num* is the number that to be converted, and *digit* is the digit of the target binary number.

The *convert_num()* function converts type 'string' numbers to type 'int'.

In order to contain all the labels, I use a structure which only contains a map, to store all the relationships between a label and its position:

```
struct LabelTable
{
    std::map<std::string, int> label_list;
};
```

4. Phase1.cpp

In phase1, I use fstream to read the files. Since we only need to consider the commands after ".text", I use the following code to pass the commands that comes before ".text":

```
std::string str;
while(getline(file,str))
    if(del_space(str).substr(0,5)==".text")
        break;
```

Then I use the following codes to find every label and it postion:

```
int cnt=0;//to count how many commands are passed until reach this label

while(getline(file,str))
{
    if(str.back()==':')
        table.label_list[str.substr(0,str.length()-1)]=cnt;
    if(del(str).length())
        cnt++;
}
```

5. Phase2.cpp

Phase2 handles the most difficult part. At first I used the same way as in phase1 to reach the parts after ".text". I then read the files and created and vector name Arg_list to store all the arugments of this command, I use sstream to separate a command to different arguments since it will seperate the string when a space is met:

```
std::stringstream ss;
std::vector<std::string> arg_list;
ss.clear();
ss.str("");
std::string inst,arg;
ss.str(str);

ss>>inst;
while(ss>>arg)
    if(arg!=",")
    {
        if(arg.back()==',')
            arg=arg.substr(0,arg.length()-1);
            arg_list.push_back(arg);
    }
```

I then do corresponding translation to the instruction depend on its type:

```
if(Inst_list[inst]->inst_type=='i')
    out_file<<I_type(inst,arg_list,table)<<std::endl;
else if(Inst_list[inst]->inst_type=='j')
    out_file<<J_type(inst,arg_list,table)<<std::endl;
else if(Inst_list[inst]->inst_type=='r')
    out_file<<R_type(inst,arg_list,table)<<std::endl;
else
    out_file<<""<<std::endl;
cnt++;//Do not forget to calculate the amount of instructions
```

Then There are three Types of translation, but the are almost the same, we use the *convert()* function to convert each part to its binary form (in string), combined these string together, and then output the final string. Here I took *R_type* as an example:

```
std::string I_type(std::string inst_name,std::vector<std::string> list,LabelTable
table)
{
    std::string opcode="000000",
                rs="00000",
                rt="00000",
                imm="0000000000000000";
    if(inst_name=="bgez")
        rt="00001";
    opcode=convert(Inst_list[inst_name]->opcode,6);
    for(int i=0;i<list.size();i++)
    {
        std::string temp=Inst_list[inst_name]->arg[i];
        if(temp=="rs")
            rs=convert(Reg_list[list[i]],5);
        else if(temp=="rt")
            rt=convert(Reg_list[list[i]],5);
        else if(temp=="imm")
            imm=convert(convert_num(list[i]),16);
        else if(temp=="imm_rs")
        {
            int l=0,r=0;    //the position of left and right brackets
            l=list[i].find('(');
            r=list[i].find(')');
            int num=convert_num(list[i].substr(0,l));
            imm=convert(num,16);
            std::string reg=list[i].substr(l+1,r-l-1);
            rs=convert(Reg_list[reg],5);
        }
        else if(temp=="label")
        {
            int num=table.label_list[list[i]];
            int dis=num-cnt-1;    //if dis<0 the convert period behaves bad
            if(dis<0)
                dis+=65536;
            imm=convert(dis,16);
        }
    }
    std::string com=opcode+rs+rt+imm;
    return com;
}
```

6. Makefile

Finally I wrote the makefile to tell the compiler how to compile this project. The following codes are how to run the project in certain enviroment given by the virtual machine.

This is to compile to whole projject:

```
make
```

This is to run test cases:

```
./tester input.asm output.txt expectedoutput.txt
```

Note that there should not have spaces in the name of these files;