Report for Project 2

1. Brief Introduction

This project allow us to build a MIPS simulator,which can run MIPS code. Below are the details on how to build this simulator.

2. Initialize

Since we may jump some instructions and to another one, it is essenstial to store all the instructions. I chose a vector to store all the instructions.

3. Virtual Registers and Virtual Memory

(1). The Reg Array

Since the registers are ordered, it is easy to find that we can use two arrays to simulate them. If we want to find a register, we will go to the register with same index, for example:

```
reg[Reg_list["v0"]]
```

Here `Reg_list` is an map which contains the relationship between the name of a register and its order number.

(2). The Mem Array

This array is used to simulate memory. We store all the binary code to the memory with little endian first:

```
for(int i=0;i<insts.size();i++)
{
    std::string &inst_temp=insts[i];
    for(int j=0;j<4;j++)
    {
        std::string temp=inst_temp.substr(j*8,8);
        mem[i*4+3-j]=bin_string(temp);
    }
}
```

For the ".data" part, we need to find out the type of the data first:

```
stringstream ss;
data_str=del(str);//str is the original string
ss.str(data_str);
ss>>data_type;
if(data_type==".ascii"||data_type==".asciiz")
{
    ......
}
else if(data_type==".word"||data_type==".byte"||data_type==".half")
{
    ......
}
```

Here the `del()` function is to delete the explanatory note of this line. Then after we do some operations to make it fit its data type, we can put these datas into `mem[]`; The `used_mem` variable is used to see how many memories are used, and guide new datas into the unused zone.

3. Executing the Instrutions

For a single instruction, we need to read its opcode to decided its type:

```
opcode=bin_string(inst.substr(0,6));
```

The `bin_string()` function is to convert a string(which is in binary form) to a integer. And then we can separate the other parts of the machine code to certain arguments(like rd, rt, rs and others). After we give value to all the arguments we need, we can use `switch` to excute certain instructions. We take J type instructions as example here:

```
if(opcode==0b000010||opcode==0b000011)
{
    inst_type='j';
    tar=bin_string(inst.substr(6,26));
    switch(opcode)
    {
        case 0b000010:
            J(reg,tar);
            break;
        case 0b000011:
            Jal(reg,tar);
            break;
        default:
            break;
    }
}
```

I use a variable `index` to find out which instruction I am excuting now. To find the next instruction should be excuted, I use the following sentence:

```
index=(reg[Reg_list["$pc"]]-START_MEM)>>2
```

If `index` is larger than `insts.size()` after the above line, it means that we have run all the instructions we need to excute, thus this is the terminate condition of the whole while-loop.

4. Instructions

I stored most of the instructions in *Instructions.h*, but the `Syscall()` function was writen in *main.cpp*, since too much auguments are needed for this function.

5 How to Run the program

(1). Make

This is a c++ project so a `make` command is used to run the makefile;

(2). How to Run

We need to run the program first:

```
main
```

After the following line come out:

```
please enter the asm file, binary codes file, checkpoints file, input file and output
file in order.
```

we input the file names (take "1" as an example)

```
1.asm 1.txt 1_checkpts.txt 1.in 1.out
```

to compare the dump files:

```
cmp correct_dump/memory_x.bin memory_x.bin
cmp correct_dump/register_x.bin register_x.bin
```