MIPS Program Execution Simulation Project Report

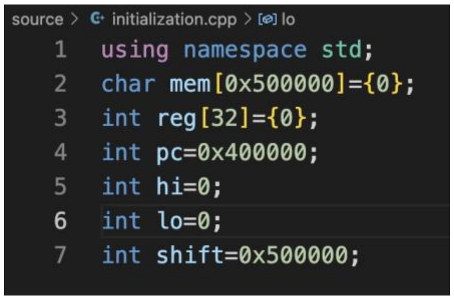
This report describes how MIPS program execution is being simulated in our programming language C++. There are two main steps:

1. Store the static data given in *.data* section in the MIPS program into our simulated memory;
2. Fetch the MIPS instructions' machine codes line by line, each line indicating an instruction that needs to be executed. Iterations stop until the program is commanded to exit.

In our case, we take five arguments as input for our C++ program. They are:
./simulator: our program name;
file.asm: contains MIPS .data section that needs to be loaded into our simulated memory;
file.txt: contains machine codes of MIPS instructions that need to be executed by our program;
file_checkpts.txt: used to verify the correctness of our program;
file.in: used by reading data as an outside source to the program;
file.out: contains our program's output until termination.

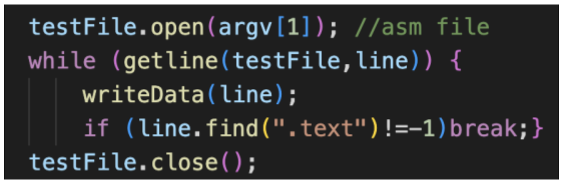
To simulate the MIPS execution, we define four C++ files. Their names and usage are:
initialization.cpp: initialize the simulated memory and the registers;
execution.cpp: identify the current machine code as a certain type of MIPS instruction;
instructions.cpp: execute the instruction identified in *execution.cpp*
simulator.cpp(where the main function is located): store the static data in *file.asm*, store all given machine codes in *file.txt*, and then simulate the machine cycle, i.e. fetch the stored machine code one by one and throw it to *execution.cpp* to execute it until the MIPS program is terminated.

To describe our program execution in detail, we simulate a time that the program is being executed:
After getting the input: *./simulator file.asm file.txt file_checkpts.txt file.in file.out*
Our program first initializes all registers and memory as empty arrays in *initialization.cpp*:



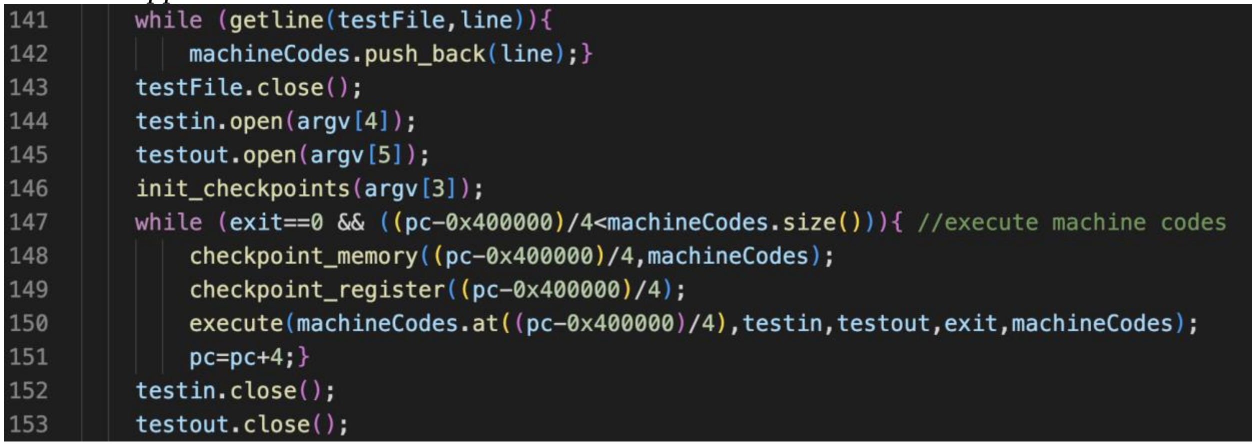
```
using namespace std;
char mem[0x500000]={0};
int reg[32]={0};
int pc=0x400000;
int hi=0;
int lo=0;
int shift=0x500000;
```

Then we write all static data in *file.asm* into the memory realized in *simulator.cpp*. The data types we support are asciiz, ascii, word, byte, and half, with the former two being in Big-Endian format, and the latter three being in Little-Endian format.



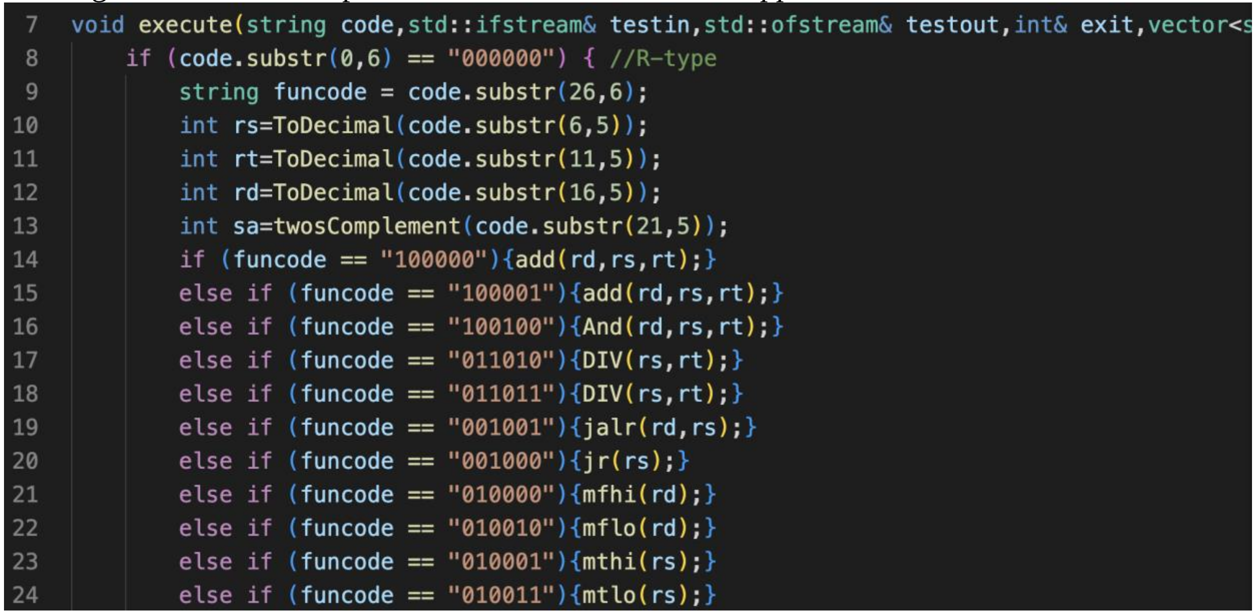
```
testFile.open(argv[1]); //asm file
while (getline(testFile,line)) {
    writeData(line);
    if (line.find(".text")!=-1)break;}
testFile.close();
```

After storing the data, we store all machine codes into a vector structure and then fetch out them one by one and execute them, i.e. throw them to *execution.cpp*. This process is also realized in *simulator.cpp*.



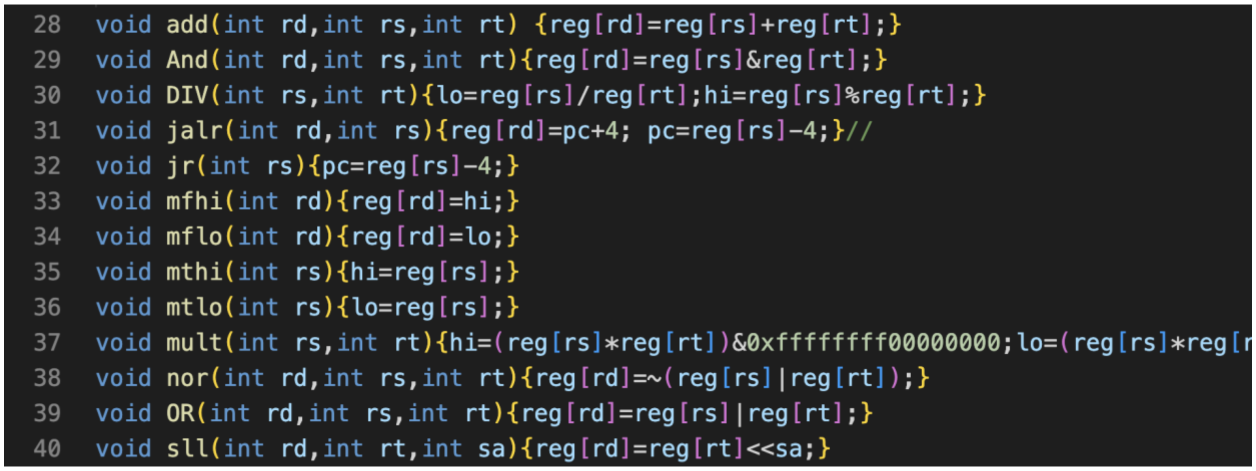
```
    while (getline(testFile,line)){
        machineCodes.push_back(line);}
    testFile.close();
    testin.open(argv[4]);
    testout.open(argv[5]);
    init_checkpoints(argv[3]);
    while (exit==0 && ((pc-0x400000)/4<machineCodes.size())){ //execute machine codes
        checkpoint_memory((pc-0x400000)/4,machineCodes);
        checkpoint_register((pc-0x400000)/4);
        execute(machineCodes.at((pc-0x400000)/4),testin,testout,exit,machineCodes);
        pc=pc+4;}
    testin.close();
    testout.close();
```

When one machine code instruction is thrown to *execution.cpp*, we first decode it according to MIPS instruction list. More specifically, we classify them into R-type, I-type, and J-type instruction according to their operation codes, and decode their corresponding registers, offsets, and target addresses. This process is realized in *execution.cpp*.



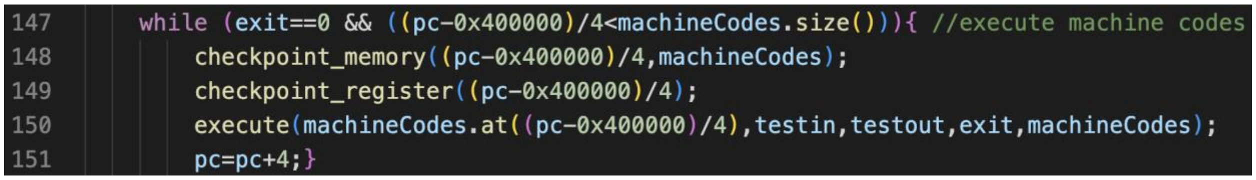
```
 7  void execute(string code,std::ifstream& testin,std::ofstream& testout,int& exit,vector<s
 8      if (code.substr(0,6) == "000000") { //R-type
 9          string funcode = code.substr(26,6);
10          int rs=ToDecimal(code.substr(6,5));
11          int rt=ToDecimal(code.substr(11,5));
12          int rd=ToDecimal(code.substr(16,5));
13          int sa=twosComplement(code.substr(21,5));
14          if (funcode == "100000"){add(rd,rs,rt);}
15          else if (funcode == "100001"){add(rd,rs,rt);}
16          else if (funcode == "100100"){And(rd,rs,rt);}
17          else if (funcode == "011010"){DIV(rs,rt);}
18          else if (funcode == "011011"){DIV(rs,rt);}
19          else if (funcode == "001001"){jalr(rd,rs);}
20          else if (funcode == "001000"){jr(rs);}
21          else if (funcode == "010000"){mfhi(rd);}
22          else if (funcode == "010010"){mflo(rd);}
23          else if (funcode == "010001"){mthi(rs);}
24          else if (funcode == "010011"){mtlo(rs);}
```

After the instruction and their target registers, offsets, and addresses are decoded, we send the instruction into its corresponding realization, which is realized in *instructions.cpp*. As an example, the “add(rd, rs, rt)” in Line 15 of the above codes is the R-type add instruction which adds two numbers rs and rt, and stores their result in rd. Therefore, the next step is to realize those institutions in *instructions.cpp*, as shown below.



```
void add(int rd,int rs,int rt) {reg[rd]=reg[rs]+reg[rt];}
void And(int rd,int rs,int rt){reg[rd]=reg[rs]&reg[rt];}
void DIV(int rs,int rt){lo=reg[rs]/reg[rt];hi=reg[rs]%reg[rt];}
void jalr(int rd,int rs){reg[rd]=pc+4; pc=reg[rs]-4;}//
void jr(int rs){pc=reg[rs]-4;}
void mfhi(int rd){reg[rd]=hi;}
void mflo(int rd){reg[rd]=lo;}
void mthi(int rs){hi=reg[rs];}
void mtlo(int rs){lo=reg[rs];}
void mult(int rs,int rt){hi=(reg[rs]*reg[rt])&0xffffffff00000000;lo=(reg[rs]*reg[r
void nor(int rd,int rs,int rt){reg[rd]=~(reg[rs]|reg[rt]);}
void OR(int rd,int rs,int rt){reg[rd]=reg[rs]|reg[rt];}
void sll(int rd,int rt,int sa){reg[rd]=reg[rt]<<sa;}
```

After this instruction is being executed, we add the program counter pc by 4 since each instruction is word-aligned. And then we execute the next instruction, or the corresponding instruction indicated by the program counter. This iterative procedure in also implemented in *simulator.cpp*.



```
    while (exit==0 && ((pc-0x400000)/4<machineCodes.size())){ //execute machine codes
        checkpoint_memory((pc-0x400000)/4,machineCodes);
        checkpoint_register((pc-0x400000)/4);
        execute(machineCodes.at((pc-0x400000)/4),testin,testout,exit,machineCodes);
        pc=pc+4;}
```

The whole program ends when the MIPS program is commanded to end by its particular instruction, or ends when all instructions have been executed and there is no more to execute.