Report for Project 2

1. Overview

In project 2, we are required to develop a MIPS simulator to run MIPS code. The simulator should support 55 MIPS instructions and simulate the $32 + 3$ registers (including `$pc`, `$hi` and `$lo`) and 6MB memory containing text segment, static data segment, dynamic data segment and stack segment. What's more, system calls such as `print_int`, `sbrk`, `exit`, `open` should be supported.

In a real MIPS computer, the static data and binary machine code will be loaded to the main memory first, and then the registers will be initialized. `$pc` will point to the address in memory where the program starts. After that, the CPU will load the instruction where `$pc` points to, let `$pc = $pc + 4`, and execute the instruction. While instruction execution in real CPU is a complex procedure, we no longer give unnecessary detail here.

2. Implementation ideas and methods

a. Memory and register simulation

To do the procedure in our simulator, we should maintain the virtual memory and virtual registers corresponding to the real memory and registers of the MIPS program.

i. Memory mapping and maintaining

Hereby I maintain a 6 MB space in memory, mapping to the space from the address `0x400000` to `0xA00000`, where the text segment is in range `0x400000` ~ `0x500000`, the data segment is in range `0x500000` ~ `0xA00000` and the stack is below `0xA00000`. For all instructions involving memory, we will transfer the address and manipulate the space we maintain in the simulator.

ii. Memory initializing

Since the `.data` section defines the data in the static data segment and the binary code should also load to the memory, we should initialize the memory before we start the simulation.

To load the binary code to the memory, we just need to transfer the string containing '0' and '1' to binary number byte by byte and copy them to the memory using little endian, as the code shown below:

```
for (int i = 0; i < insts.size(); i++) {
    auto &inst = insts[i];
    /* Little endian */
    for (int j = 0; j < 4; j++) {
        mem[i * 4 + 3 - j] = binStrToNum(inst.substr(j * 8, 8));
    }
}
```

To load `.data` section, we should parse the data first. We should identify the type the data and apply different method to deal with the data. For `.ascii` and `.asciiz`, we store the string to the memory using big endian, and extra `\0` should be added for `.asciiz`. For `.byte`, `.half`, `.word`, we should store them using little endian, while each number occupies 1 byte for `.byte`, 2 byte for `.half`, 4 byte for `.word`, as the code shown below:

```
for (int i = 0; i < numList.size(); i++) {
    /* Mapping the address */
    byte *ptr = mem + TEXT_SEG_SIZE + memUsed;
    if (dataType == BYTE) {
        *ptr = numList[i];
        memUsed++;
    } else if (dataType == HALF) {
        *ptr = numList[i] & 0xff;
        *(ptr + 1) = (numList[i] >> 8) & 0xff;
        memUsed += 2;
    } else if (dataType == WORD) {
        *ptr = numList[i] & 0xff;
        *(ptr + 1) = (numList[i] >> 8) & 0xff;
        *(ptr + 2) = (numList[i] >> 16) & 0xff;
        *(ptr + 3) = (numList[i] >> 24) & 0xff;
        memUsed += 4;
    }
}
```

iii. Register simulation

To simulate the registers, we maintain an array of 32-bit integer, corresponding to the 35 registers (including `$pc`, `$hi` and `$lo`), as the code shown below.

```
uint32 reg[REG_CNT + 3];
```

When a register should be read or manipulated, we turn to the array above. and the identifiers of registers is represented using enumeration as below:

```
enum REGS {
    _zero, _at, _v0, _v1, _a0, _a1, _a2, _a3,
    _t0, _t1, _t2, _t3, _t4, _t5, _t6, _t7,
    _s0, _s1, _s2, _s3, _s4, _s5, _s6, _s7,
    _t8, _t9, _k0, _k1, _gp, _sp, _fp, _ra,
    _pc, _hi, _lo
};
```

b. Instruction execution

To execute the instruction, we should read the instruction from memory for each cycle. To accelerate the procedure, I buffer the instructions to simplified the addressing process.

After reading the instruction, we identify the type of instruction and parse the instruction, to find out what kind of instruction it is and its argument.

```
    opCode = binStrToNum(inst.substr(0, 6));
    if (opCode == 0b000000) {
        instType = R_TYPE;
        rs = binStrToNum(inst.substr(6, 5));
        rt = binStrToNum(inst.substr(11, 5));
        rd = binStrToNum(inst.substr(16, 5));
        sa = binStrToNum(inst.substr(21, 5));
        funct = binStrToNum(inst.substr(26, 6));
    } else if (opCode == 0b000010 or opCode == 0b000011) {
        instType = J_TYPE;
```

```
        target = binStrToNum(inst.substr(6, 26));
    } else {
        instType = I_TYPE;
        rs = binStrToNum(inst.substr(6, 5));
        rt = binStrToNum(inst.substr(11, 5));
        imm = binStrToNum(inst.substr(16, 16));
    }
```

As you can see, opcode is used to identify the type, and we parse each part of the instruction. Then we let `$pc = $pc + 4`

Then we run different procedure according to the opcode and function code:

```
    switch (instType) {
        case R_TYPE:
            switch (funct) {
                case 0b100000:
                    _add();
                    break;
                    ......
        case J_TYPE:
            switch (opCode) {
                case 0b000010:
                    _j();
                    break;
                case 0b000011:
                    _jal();
                    break;
            }
     }
```

Taking `add` as an example:

```
void Simulator::_add() {
    reg[rd] = (int)reg[rs] + (int)reg[rt];
}
```

Each function of instruction is implemented in detail, we no longer give unnecessary details here.

c. System calls

System calls involves IO or memory allocation. First we identify the kind of system call using `$v0` and the apply different method

i. Standard IO

For input and output, we use std::fstream. Taking `print_int` and `read_int` as examples:

```
void Simulator::_print_int() {
    fout << (int)reg[_a0];
    fout.flush();
}

void Simulator::_read_int() {
    int rst = 0;
    fin >> rst;
    std::string str;
    getline(fin, str);
    reg[_v0] = rst;
}
```

To eliminate the extra characters, we call `getline` for safety.

ii. File manipulation

For file manipulation, we call Linux API `open`, `read`, `write` and `close`:

```
void Simulator::_open() {
    reg[_a0] = open((const char *)mem + reg[_a0] - MEM_BIAS, reg[_a1],
reg[_a2]);
}

void Simulator::_read() {
    reg[_a0] = read(reg[_a0], mem + reg[_a1] - MEM_BIAS, reg[_a2]);
}

void Simulator::_write() {
    reg[_a0] = write(reg[_a0], mem + reg[_a1] - MEM_BIAS, reg[_a2]);
}

void Simulator::_close() {
    close(reg[_a0]);
}
```

iii. Memory allocation

We use a variable to store the space we already used, when `sbrk` is called, we return the address and increase that variable:

```
void Simulator::_sbrk() {
    reg[_v0] = MEM_BIAS + memUsed + TEXT_SEG_SIZE;
    memUsed += reg[_a0];
}
```

3. Usage

a. Build

To build the simulator, run commands below please:

```
make
# or
make simulator
```

b. Test

To run the simulator using `many` test case:

```
./simulator many/many.asm many/many.txt many/many_checkpts.txt many/many.in
many/many.out
```

To compare the register dump file and memory dump file:

```
cmp many/correct_dump/register_97.bin register_97.bin
cmp many/correct_dump/memory_97.bin memory_97.bin
```

To run the simulator using `lwlr_swlr_2` test case:

```
./simulator lwlr_swlr_2/lwlr_swlr_2.asm lwlr_swlr_2/lwlr_swlr_2.txt
lwlr_swlr_2/lwlr_swlr_checkpts.txt lwlr_swlr_2/lwlr_swlr.in
lwlr_swlr_2/lwlr_swlr.out
```

You can also run other test cases in a similar way.