CSC3050 Assignment 2

121020163 沈驰皓

1. Overview of How MIPS programs are executed

In this assignment, we built an MIPS simulator to simulate how a computer would execute the MIPS programs. Basically, the parts of the computer, such as CPU, main memory... are the places that execute the program. Here we just do a simulation of it.

Before executing the MIPS program, the computer allocates a new address space for saving the data. The memory has the components below: text segment, data segment and stack segment. The text segment is at the bottom of the memory, containing the binary machine code of the program. Above text segment, the data segment includes two parts: static and dynamic data segment. The static segment stores the data in `.data` segment of the MIPS file, whose lifetime is the whole execution time and can be accessed by the program. The dynamic part allocates dynamic data during the execution time, without assignment in advance. Finally, the stack is like the dynamic data segment. The difference is that all three parts above increase the address while storing the data, however, the stack segment starts at the top and expands down toward the data segment.

For preparation, the assembler first assembles the MIPS code to machine code (which we've done in assignment 1), and then the static data together with the machine code are stored in the relative parts of main memory. Registers in CPU also need to be initialized. The PC register will first be initialized at the first line of the machine code in the text segment. Other registers are set to be zero or some specific values and the program is ready for execution.

Now for the execution part, the CPU will start a loop. It will first fetch the machine code line where PC points at, and then the PC will be incremented by 4. After that, the CPU will analyze and execute the instruction. During execution, registers in CPU are used to hold variables used in the program. Besides 32 general purpose registers, here we also implement the PC register which is mentioned above and is used to store the current execution address, the HI register which contains the higher word of the results of division or multiplication, and the LO register which contains the lower word.

2. High Level Implementation Ideas

Since we also need to check what registers and the memory are like for each checkpoint, I built both `Simulator` and `Checkpoints` classes to solve the problem.

For execution, I construct a `Simulator` class and break down the problem into two parts: init and simulate. Before init, the construction of the `Simulator` class first does the memory allocation, initializes all the registers to their specific values, and also builds a `Checkpoints` class inside it. It also maintains a variable to count the number of instructions for checkpoints. Then during the init process, the program load the machine code to the text segment and its `.data` part to the static data segment, as well as load all the checkpoints needed to the `Checkpoints` class. Finally, for the simulating process, the program opens the input and output file at the beginning for preparation and then starts the simulation loop. During the simulation loop, the program first decides whether the current registers and memory need to be dumped by checkpoints. After that, it reads one instruction at a time, adds the program counter, and then executes it. While executing, I divide the instructions into R, I and J types. The program initially checks which type the instruction is, locates it to the corresponding function, loads the data needed from the registers, and ultimately executes the function as well as outputs. For the R type, the system call instruction is also supported. The program will decide which system call to be roused by the registers, and then read the input file, output to the output file, or do something according to the system call type. The loop will not stop until the program counter is out of space or the exit system call is roused.

For checkpoints, the registers and memory are dumped during execution part above. The `Checkpoints` class will packed all the data into two binary files correspondingly, then output the binary files for further checking.

3. Implementation Details

A. Simulator class

a. Construction

During construction, several structures need to be maintained. For memory, I simulate the main memory called `_block` for 6 MB, and I use the char array to implement it. For registers, I also maintain an unsigned integer array called `_regs` with the size of 32 (general registers) + 3 (3 specific registers). For some specific memory addresses, I use some constant integers to specialize them. For instance, `START_ADDR` is initialized to be `0x400000`, `STATIC_ADDR` is `0x500000`, and `STACK_ADDR` is `0xA00000`. Therefore, different segments of data in the simulated main memory can be specified by index. For register accessing, the following enumerator structure is constructed for convenience:

```
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
};
```

Most of the `_regs` are initialized to 0. Some are set to other values. The PC is set to `_regs[$pc] = START_ADDR`. FP and SP are `_regs[$fp] = this->_regs[$sp] = STACK_ADDR`. GP is `_regs[$gp] = STATIC_ADDR + 0x8000`.

A `staticDataPos` integer property is set to 0 for recording the upper address of the static data for further use, and an `instCount` integer property is also set to 0 for checkpoints. Also, the `Checkpoints` class is initialized here.

b. Initialization

It contains three parts. First, it reads all the points that need to be checked by the `push()` method from the `Checkpoints` class, which will be mentioned later. Then the program uses `fstream` library to read the `.data` part and machine code and allocates them into `_block` array.

For the data segment, we implement the `trim()` and `removeComments()` methods to remove comments and useless characters before and after each data line. Then we use `substr()` method from the string library to fetch the type and the exact data part of each line. For word and byte type, we store the data in `_block` by little endian. For ascii and asciiz, we store it by big endian. At the same time the `staticDataPos` need to be added by a multiple of 4 closest to the top of the static data segment, in order to protect the integrity of the block property.

For the text segment, we directly read the machine code line by line, allocating them in the text segment of `_block` by little endian as well. The little endian allocation of the text segment is shown below, while that of the data segment is similar. And for big endian, since it stores index by index, so we omit it here.

```
int count = 0;
while (inB >> inst) {
    for (int offset = 0; 4 > offset; offset++) {
        unsigned int bin = strToNum(inst.substr(offset * 8, 8));
        this->_block[4 * count + 3 - offset] = bin;
    }
    count++;
}
```

c. Simulation

As explained above, in the beginning, it opens both input and output files and stores them as file streams `inF` and `outF`. Then the execution loop begins. For checkpoints dump, we'll mention it later. For the loop body part, It first uses a `_fetchCode` method to fetch the machine code:

```
string Simulator::_fetchCode(unsigned int pc) {
    string res = bitset<8>(this->_block[pc - START_ADDR + 3]).to_string() +
                 bitset<8>(this->_block[pc - START_ADDR + 2]).to_string() +
                 bitset<8>(this->_block[pc - START_ADDR + 1]).to_string() +
                 bitset<8>(this->_block[pc - START_ADDR]).to_string();
    return res;
}
```

Then it adds the PC register by 4 and executes the machine code. For the execution part, we define a private method named `_execute` as an instruction identifier. It uses the opcode to determine which type of instruction it is, fetch the fields, and send them to the corresponding type execution method.

```
void Simulator::_execute(const string &inst) {
    unsigned int op = strToNum(inst.substr(0, 6));
    if (op == 0b000000) {
        // rs rt rd shamt function
        _rType(strToNum(inst.substr(6, 5)), strToNum(inst.substr(11, 5)),
               strToNum(inst.substr(16, 5)), strToNum(inst.substr(21, 5)),
               strToNum(inst.substr(26, 6)));
    } else if (op == 0b000010 || op == 0b000011) {
        // opcode target
        _jType(op, strToNum(inst.substr(6, 26)));
    } else {
        // opcode rs rt immediate
        _iType(op, strToNum(inst.substr(6, 5)),
               strToNum(inst.substr(11, 5)), strToNum(inst.substr(16, 16)));
    }
}
```

Then for three types of execution methods, `_rType()`, `_jType()` and `iType()`, we all use the switch case statements to implement the detailed function. While R type uses the function code to distinguish which instruction is used, the other two type use opcode to specify. Take `_rType()` as an example:

```
void Simulator::_rType(unsigned int rs, unsigned int rt, unsigned int rd,
unsigned int sa, unsigned int func) {
    switch (func) {
        case 0b100000:  // add
            this->_regs[rd] = (int) this->_regs[rs] + (int) this->_regs[rt];
            break;
        ...
        case 0b001100:  // syscall
            _syscall();
            break;
        case 0b100110:  // xor
            this->_regs[rd] = this->_regs[rs] ^ this->_regs[rt];
        default:
            break;
    }
}
```

Each case represents a detailed function. Since the number of cases is large, we no longer explain it here. We only explain the `syscall` instruction in detail.

For the `syscall` instruction we are asked to implement, it reads `_regs[$v0]` to decide which system call should the program arouse. It can be divided into four types.

The first deals with the I/O of the program. it uses the file streams `inF` and `outF` to input and output data. For output, `print_int` and `print_char` output the corresponding data stored in `_regs[$a0]`, while `print_string` outputs the string in `_block`, the address of which is in `_regs[$a0]`. For input, `read_int` and `read_char` read the data from file stream `inF` and store it in `_regs[$v0]`, while `read_string` stores the string to `_block` from `inF`, the address of which is also in `_regs[$a0]`, while the length of the string should be smaller than `_regs[$a1]`.

The second deals with file manipulation. As is required, `open`, `read`, `write` and `close` are implemented. We call Linux API for those four calls in the implementation directly.

The third is used to exit the execution. For `exit`, we directly exit the program with status code 0, while for `exit2`, we exit the program with status code stored in `_regs[$a0]`.

The last type is for dynamic memory allocation. The only call for it is `sbrk`. It'll allocate a memory size of `_regs[$a0]` after `STATIC_ADDR + staticDataPos` and then return the pointer at the beginning of the location to `_regs[$v0]`. The `staticDataPos` will be increased by `_regs[$a0]` as well for new memory allocation.

B. Checkpoints class

Basically, it uses a `vector<int>` structure property called `_points` to contain all the instruction numbers that need to be checked.

a. Push

As mentioned above, during initialization, it should push all the instruction numbers that read from the checkpoints text into the vector. So the `push()` method of this class uses the `push_back()` function from the vector standard library to implement that.

b. Dump

The `dump()` function firstly checks whether the current instruction number is in the `_points` vector. If true, then it will construct an `ofstream` class called `d`, and then write the registers and memory data at the current point to the corresponding output binary files. It is implemented as shown below:

```cpp
void Checkpoints::dump(int instCount, unsigned int *_regs, unsigned char
*_block, int size) {
    if (count(_points.begin(), _points.end(), instCount)) {
        ofstream d;

        // dump register
        d.open("register_" + to_string(instCount) + ".bin", ios::binary);
        d.write((char *) _regs, 35 * 4);
        d.flush();
        d.close();
        // dump memory
        d.open("memory_" + to_string(instCount) + ".bin", ios::binary);
        d.write((char *) _block, size);
        d.flush();
        d.close();
    }
}
```