Lab report - Simulator

PB20111686 Ruixuan Huang

Lab Goal

Read the given code, understand the framework of the program, and replace all `TO BE DONE` in the code with the correct code. And, learn to use cmake.

Overview

The goal of this lab is to realize a LC-3 binary code simulator. For an assembly code, I can use the assembler I realized in labA to transform it to binary code. And using this simulator, we can get the result after the binary-represented instructions are carried.

The process of simulation is easy to comprehend. The simulator simulates all registers in LC-3 in the form of variables, including memory and temporary registers. We just need to read the instructions to be executed from the file to the virtual machine memory and then execute them in the logical order of LC-3 instructions.

Completion

common.h

When running, the IDE prompts that the file is missing the BitSet class, so this file needs to add a line.

```
#include <bitset>
```

memory.cpp

The functions here are to obtain the instruction represented by a string from a file, obtain the specified instruction, and overload the subscript operator.

```
void memory_tp::ReadMemoryFromFile(std::string filename, int beginning_address) {
    // Read from the file
    std::fstream file(filename);
    int address_ptr = beginning_address;
    std::string line;
    while (getline(file, line)) {
        for (int i = 0; i < 16; i++) {
            memory[address_ptr] |= (line[i] == '1') ? (1 << (15 - i)) : 0;
        }
        address_ptr++;
    }
}

int16_t memory_tp::GetContent(int address) const {
    // get the content
    return memory[address];
}

int16_t& memory_tp::operator[](int address) {
    // get the content
    return memory[address];
}
```

simulator.cpp

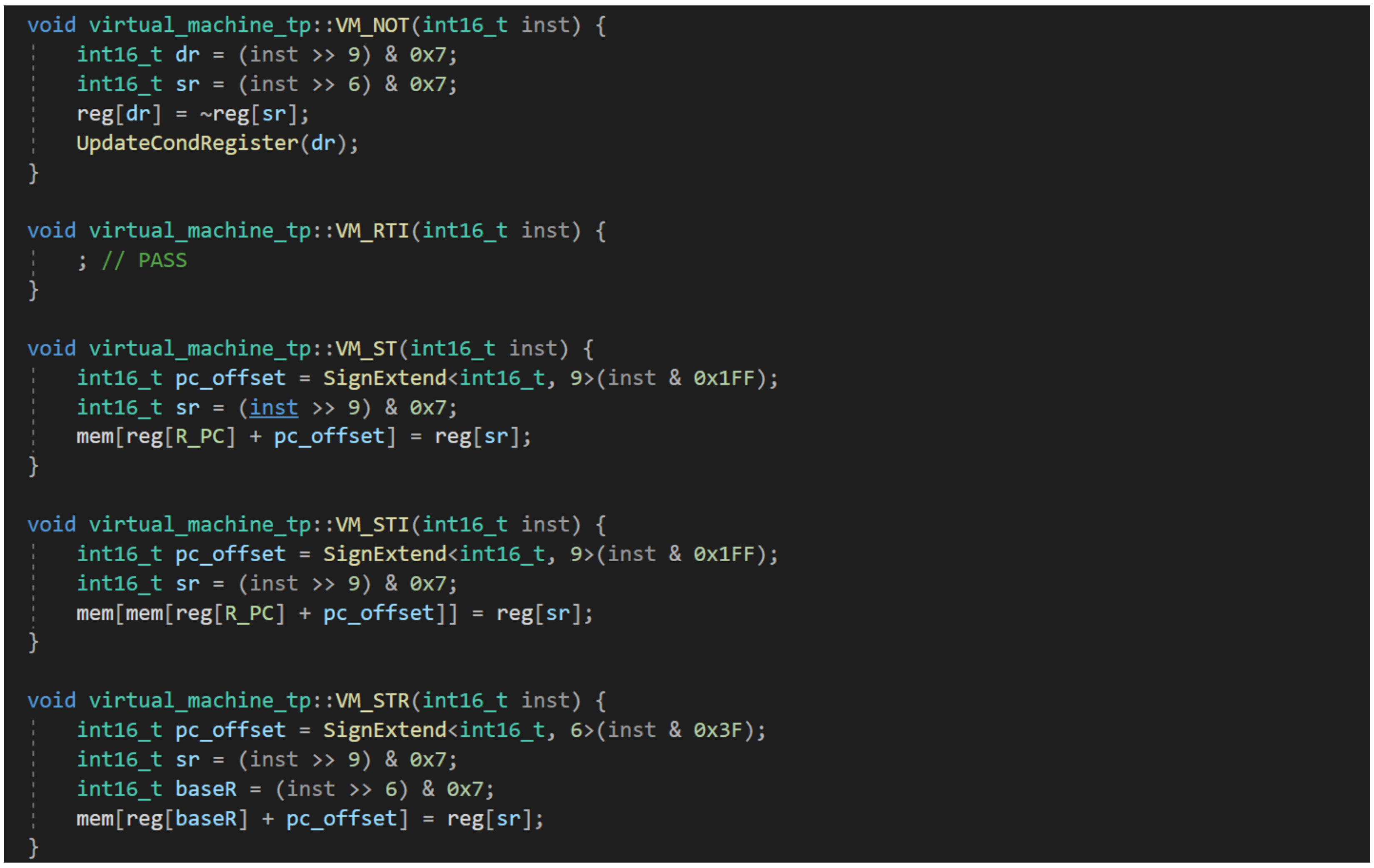
The first thing to complete is the symbolic expansion function.

```
template <typename T, unsigned B>
inline T SignExtend(const T x) {
    // Extend the number
    T highest_bit = x & (1 << (B - 1));
    if (highest_bit) {
        return x | ~((T)pow(2, B) - 1);
    }
    else {
        return x;
    }
}
```

The second is to complete the function used to update the condition code.

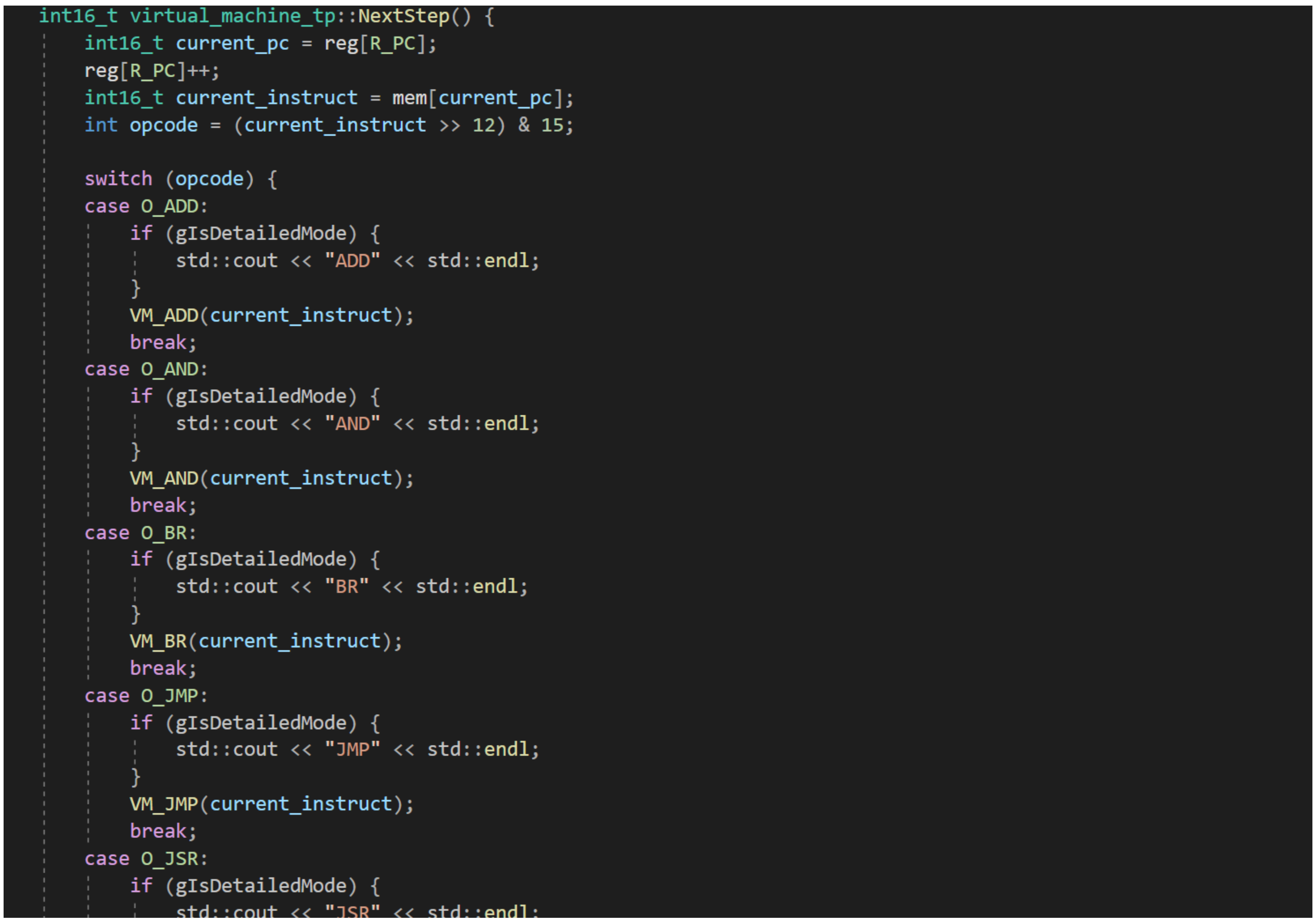
```
void virtual_machine_tp::UpdateCondRegister(int regname) {
    // Update the condition register
    if (reg[regname] == 0) {
        reg[R_COND] = 0x2;
    }
    else if (reg[regname] > 0) {
        reg[R_COND] = 0x1;
    }
    else {
        reg[R_COND] = 0x4;
    }
}
```

The third part is based on the given VM_ADD and VM_BR section to simulate the instructions to be executed by other opcodes. Because there are too many contents, they are not displayed in the report. Most of them are translated according to the instruction table at the back of the book. Just put one picture to show part of my result.



```
void virtual_machine_tp::VM_NOT(int16_t inst) {
    int16_t dr = (inst >> 9) & 0x7;
    int16_t sr = (inst >> 6) & 0x7;
    reg[dr] = ~reg[sr];
    UpdateCondRegister(dr);
}

void virtual_machine_tp::VM_RTI(int16_t inst) {
    ; // PASS
}

void virtual_machine_tp::VM_ST(int16_t inst) {
    int16_t pc_offset = SignExtend<int16_t, 9>(inst & 0x1FF);
    int16_t sr = (inst >> 9) & 0x7;
    mem[reg[R_PC] + pc_offset] = reg[sr];
}

void virtual_machine_tp::VM_STI(int16_t inst) {
    int16_t pc_offset = SignExtend<int16_t, 9>(inst & 0x1FF);
    int16_t sr = (inst >> 9) & 0x7;
    mem[mem[reg[R_PC] + pc_offset]] = reg[sr];
}

void virtual_machine_tp::VM_STR(int16_t inst) {
    int16_t pc_offset = SignExtend<int16_t, 6>(inst & 0x3F);
    int16_t sr = (inst >> 9) & 0x7;
    int16_t baseR = (inst >> 6) & 0x7;
    mem[reg[baseR] + pc_offset] = reg[sr];
}
```

The last step is the most boring part of this file. It is to complete the single-step execution function. Write the name of the instruction function to be executed according to the operation code, all of which are copy and paste.



```cpp
int16_t virtual_machine_tp::NextStep() {
    int16_t current_pc = reg[R_PC];
    reg[R_PC]++;
    int16_t current_instruct = mem[current_pc];
    int opcode = (current_instruct >> 12) & 15;

    switch (opcode) {
    case O_ADD:
        if (gIsDetailedMode) {
            std::cout << "ADD" << std::endl;
        }
        VM_ADD(current_instruct);
        break;
    case O_AND:
        if (gIsDetailedMode) {
            std::cout << "AND" << std::endl;
        }
        VM_AND(current_instruct);
        break;
    case O_BR:
        if (gIsDetailedMode) {
            std::cout << "BR" << std::endl;
        }
        VM_BR(current_instruct);
        break;
    case O_JMP:
        if (gIsDetailedMode) {
            std::cout << "JMP" << std::endl;
        }
        VM_JMP(current_instruct);
        break;
    case O_JSR:
        if (gIsDetailedMode) {
            std::cout << "JSR" << std::endl;
```

Notice the use of this function in `main.cpp`. Finally, we need to return the value of PC.

main.cpp

Just to complete the use of single step function.

```cpp
while(halt_flag) {
    // Single step
    halt_flag = virtual_machine.NextStep();//my edit
    if (gIsDetailedMode)
        std::cout << virtual_machine.reg << std::endl;
    ++time_flag;
}
```

CMake usage

After installing cmake, enter the following instructions at the terminal to create a VS solution.

```
mkdir build
cd build
cmake ..
```

At this point, the project solution is in the `build` folder. Then we can use VS to compile the project.