# ICS LABS 实验报告

## 1. 内容简介

本实验要求我们根据现有框架搭建LC-3模拟器，根据输入的机器码模拟出最终的运行结果。

## 2. 内容概述

### 2.2.1 寄存器单元

本实验使用如下的方式进行寄存器模拟。一共有 8 个通用寄存器、 1 个程序寄存器以及 1 个条件码寄存器。

```
namespace virtual_machine_nsp {
    const int kRegisterNumber = 10;
    enum RegisterName {
        R_R0 = 0,
        R_R1,
        R_R2,
        R_R3,
        R_R4,
        R_R5,
        R_R6,
        R_R7,
        R_PC, // 8
        R_COND // 9
    };

    typedef std::array<int16_t, kRegisterNumber> register_tp;
    std::ostream& operator<<(std::ostream& os, const register_tp& reg);
} // virtual machine namespace
```

### 2.2.2 内存单元

本实验使用如下的方式进行内存模拟。

```
const int kVirtualMachineMemorySize = 0xFFFF;

class memory_tp {
    private:
    int16_t memory[kVirtualMachineMemorySize];

    public:
    memory_tp() {
        memset(memory, 0, sizeof(int16_t) * kVirtualMachineMemorySize);
    }
    // Managements
    void ReadMemoryFromFile(std::string filename, int beginning_address=0x3000);
```

```
    int16_t GetContent(int address) const;
    int16_t& operator[](int address);
};
```

程序会在一开始将所有的指令码（预先存放在input.txt中）读入模拟内存，再进行后续操作，符合LC-3的实现方式。内存初始读入过程如下：

```
namespace virtual_machine_nsp {
    void memory_tp::ReadMemoryFromFile(std::string filename, int
beginning_address) {
        // Read from the file
        std::ifstream input_file;
        input_file.open(filename);
        if (input_file.is_open()) {
            const int line_length = 20;
            char read_in_line[line_length];
            while(input_file.getline(read_in_line, line_length)){
                read_in_line[16] = '\0';
                memory[beginning_address++] = strtol(read_in_line, NULL, 2);
            }
        }
    }

    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];
    }
}; // virtual machine namespace
```

## 2.2.3 模拟单元

模拟单元为本实验的功能核心，包括：从内存中读取当前指令、指令解码与执行、程序运行判断等。其类实现如下：

```
class virtual_machine_tp {
    public:
    register_tp reg;
    memory_tp mem;

    // Instructions
    void VM_ADD(int16_t inst);
    void VM_AND(int16_t inst);
    void VM_BR(int16_t inst);
    void VM_JMP(int16_t inst);
    void VM_JSR(int16_t inst);
    void VM_LD(int16_t inst);
    void VM_LDI(int16_t inst);
    void VM_LDR(int16_t inst);
```

```
    void VM_LEA(int16_t inst);
    void VM_NOT(int16_t inst);
    void VM_RTI(int16_t inst);
    void VM_ST(int16_t inst);
    void VM_STI(int16_t inst);
    void VM_STR(int16_t inst);
    void VM_TRAP(int16_t inst);

    // Managements
    virtual_machine_tp() {}
    virtual_machine_tp(const int16_t address, const std::string &memfile, const
std::string &regfile);
    void UpdateCondRegister(int reg);
    void SetReg(const register_tp &new_reg);
    int16_t NextStep();
};
```

其中包括了各指令的模拟执行、程序更新、时序控制等操作。

#### 2.2.3.1 LC-3指令实现

以 AND（1001) 为例。其实现代码如下：

```
void virtual_machine_tp::VM_AND(int16_t inst) {
    int flag = inst & 0b100000;
    int dr = (inst >> 9) & 0x7;
    int sr1 = (inst >> 6) & 0x7;
    if (flag) {
        // add inst number
        int16_t imm = SignExtend<int16_t, 5>(inst & 0b11111);
        reg[dr] = reg[sr1] & imm;
    } else {
        // add register
        int sr2 = inst & 0x7;
        reg[dr] = reg[sr1] & reg[sr2];
    }
    // Update condition register
    UpdateCondRegister(dr);
}
```

可以看到，基本的实现逻辑是：首先对输入的 inst 进行拆分，得到对应的DR、SR1、SR2。随后按照LC-3的运行逻辑对其进行按位与操作。最后，程序对状态码进行更新，完成这条指令周期。

对于涉及内存访问的指令，我们以 LD (0010) 为例：

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

可以看到，程序首先根据 PC 偏移计算出目标地址，再从内存中相应位置获取数值，存入目标寄存器。其他指令的执行方式与此类似。

### 2.2.3.2 时序控制函数

下面的函数实现了指令执行的时序逻辑。具体为：取指令、PC自增、译码、执行。如果当前指令无效或已经执行完毕，则会发出停机信号HALT。

```
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;
        }
        VM_JSR(current_instruct);
        break;
        case O_LD:
        if (gIsDetailedMode) {
            std::cout << "LD" << std::endl;
        }
        VM_LD(current_instruct);
        break;
        case O_LDI:
        if (gIsDetailedMode) {
            std::cout << "LDI" << std::endl;
        }
        VM_LDI(current_instruct);
        break;
        case O_LDR:
```

```
        if (gIsDetailedMode) {
            std::cout << "LDR" << std::endl;
        }
        VM_LDR(current_instruct);
        break;
        case O_LEA:
        if (gIsDetailedMode) {
            std::cout << "LEA" << std::endl;
        }
        VM_LEA(current_instruct);
        break;
        case O_NOT:
        if (gIsDetailedMode) {
            std::cout << "NOT" << std::endl;
        }
        VM_NOT(current_instruct);
        break;
        case O_RTI:
        if (gIsDetailedMode) {
            std::cout << "RTI" << std::endl;
        }
        VM_RTI(current_instruct);
        break;
        case O_ST:
        if (gIsDetailedMode) {
            std::cout << "ST" << std::endl;
        }
        VM_ST(current_instruct);
        break;
        case O_STI:
        if (gIsDetailedMode) {
            std::cout << "STI" << std::endl;
        }
        VM_STI(current_instruct);
        break;
        case O_STR:
        if (gIsDetailedMode) {
            std::cout << "STR" << std::endl;
        }
        VM_STR(current_instruct);
        break;
        case O_TRAP:
        if (gIsDetailedMode) {
            std::cout << "TRAP" << std::endl;
        }
        if ((current_instruct & 0xFF) == 0x25) {
            reg[R_PC] = 0;
        }
        VM_TRAP(current_instruct);
        break;
        default:
        VM_RTI(current_instruct);
        break;
    }

    if (current_instruct == 0) {
        // END
        // TODO: add more detailed judge information
```

```
        return 0;
    }
    return reg[R_PC];
}
```

### 2.2.3.3 相关辅助函数

#### I. 补码扩展函数

该函数的功能是将输入的补码扩展到16位。

```
template <typename T, unsigned B>
inline T SignExtend(const T x) {
    // Extend the number
    if (x & (1 << (B - 1))) {
        return x | (0xFFFF << B);
    }
    else {
        return x & (~(0xFFFF << B));
    }
}
```

基本思路是，先判断最高位是 0 还是 1，再将其剩下的位上补 0 或 1 。

#### II. 状态码更新函数

该函数的功能是根据目标寄存器的纸进行状态码更新。

```
void virtual_machine_tp::UpdateCondRegister(int regname) {
    // Update the condition register
    if (reg[regname] > 0)
        reg[R_COND] = 0b001;
    else if (reg[regname] == 0)
        reg[R_COND] = 0b010;
    else
        reg[R_COND] = 0b100;
}
```

# 3.小结

本实验成功实现了 LC-3 的模拟器，让我对LC-3程序运行的内在逻辑有了更为深入的了解。