LAB S - Simulator

Purpose

本次实验的目的在于用高级语言实现模拟器，在这个过程中可以加深对汇编语言的各种指令的理解，也可以进一步了解各种错误出现的机制。

Principles

模拟器的实验原理较为简单，就是根据各条指令的执行规则来编写程序，例如对于`ADD`指令，就是将两个寄存器中的数相加或是将一个寄存器中的数与一个立即数相加，最后将结果存在目标寄存器里。寄存器号及立即数都可以通过机器码获得。

在实现了模拟器的各项基本功能，只需要将它们组合在一起，即在一个main函数中依据模拟器的逻辑调用各项功能需要的函数即可：只需要依次从文件中读取机器码，再执行机器码，一直执行到汇编程序结束即可。在执行的过程中，需要实现单步调试，输出内存内容，修改内存值，输出寄存器内容，修改寄存器值等功能。

Procedure

下面只需要依次补全框架中残缺的部分即可完成实验（还需要对框架作出一定的修改）：

register.h

因为需要检查`Privilege mode violation`，所以我对框架中的`register.h`作了修改，增加了存储`PSR`的寄存器`reg[R_PSR]`，初始化为`0x8002`(`PSR[15] = 1`表示当前为`User mode`，`PSR[2:0] = 2`表示条件码初始化为010)。在`PSR[15] = 1`时使用`RTI`指令即会导致`Privilege mode violation`。

simulator.h

为了便于随时显示或修改内存值或寄存器值，我在`simulator.h`中增加了一些函数用来实现相应的功能：

```
    void ClearMem(); // 清空内存
    void ShowMem(int begin, int end); // 显示内存从begin到end范围的值
    void ChangeMem(int address, int16_t value); // 改变地址为address的内存单元中的值为value
    void ChangeReg(int x, int16_t value); // 改变寄存器x中的值为value
    void ShowReg(); // 显示各寄存器的值
    void ClearReg(); // 清空寄存器，其中reg[R_COND]清空为2（010）,reg[R_PSR]清空为0x8002
    void ShowMode(); // 显示当前处于的模式（User mode/Supervisor mode）
    void ChangeMode(); // 改变当前处于的模式
```

各个函数的功能已经注释在了函数的后面。

memory.cpp

对于这个`cpp`文件只需要补全代码中的`TO BE DONE`部分即可。

`GetContent`函数：直接返回内存中的值即可，即`return memory[address]`。

`operator`函数：这个函数也是一样，直接返回内存中的值(`return memory[address]`)即可。

simulator.cpp

SignExtend

这个函数用于将一个T类型的有效长度为B的函数进行符号位扩展，在实际应用中，即将一个`int16_t`的数值`x`从第B-1位开始进行符号位扩展，如果`x`的第B-1位是0，则需要将B-1位后面的位都变成0，否则需要将前面的位都变成1。可以用掩码实现上述要求，初始化一个为T类型的只有第B-1位为1的mask，如果`mask & x`不为0，则说明`x`的第B-1位为1，需要扩展1，只需要将mask不断左移，左移后与`x`进行或操作，一直移动至mask变成0即可；如果`mask & x`等于0，则需要扩展0，只需要将mask减1，就可以得到后B-1位均为0的mask，再与x进行与操作即可，这里因为x的第B-1位本就为0，所以不用担心mask会改变x的第B-1位的值。

由于传入的`x`是`const T`类型，所以不能直接对x进行上述操作，需要设一个临时变量`temp`，初始化为x，最后返回temp即可。

以下为代码实现：

```
template <typename T, unsigned B>
inline T SignExtend(const T x) {
    // Extend the number
    // TO BE DONE
    // 用掩码mask来将x的指定位变为1，放到temp中
    T mask = 1 << (B-1); // mask初始值设为最高位为1的B位宽的数值
    T temp = x;
    // 若x的首位为1，则需要在前面补1，如果是0，则不需要处理，因为在内存中会自动填充0
    if((mask & x) != 0)
    {
        while(mask) // 循环至mask为0结束，此时x前的所有位都填充了1
        {
        mask = mask << 1;// mask先左移一位
            temp |= mask;// 将temp的指定位变为1
        }
    }
    else
    {
        mask -= 1;
        temp &= mask;
    }
    return temp;
}
```

UpdateCondRegister

这个函数用于改变条件码，即修改`reg[R_COND]`，只需要`reg[rename]`的值来对`reg[R_COND]`进行赋值即可，如果为负则赋值为4（100），为0则赋值为2（010），为正则赋值为1（001）。以下为代码实现：

```
void virtual_machine_tp::UpdateCondRegister(int regname) {
    // Update the condition register
    // TO BE DONE
    // 根据reg[regname]的值来设置条件码即可，nzp分别对应4,2,1
    reg[R_COND] = (reg[regname]) ? (reg[regname] > 0 ? 1 : 4) : 2;
}
```

执行指令

运算指令

对于与或非三种指令，首先需要获得目标寄存器的编号和第一个源寄存器的编号，它们都表示在机器码中。设传入的机器码为`inst`，如果要获得目标寄存器的编号，就只需要`int dr = (inst >> 9) & 0x7`即可，其他各项需要获得的值都可以依据这样的模式获得：将机器码右移n位后与相应的数值（这个数值的二进制表示的后n位均为1）进行与操作。对于`NOT`指令，只需要将源寄存器1的数值取反后放到目标寄存器中即可；而对于`ADD`指令和`AND`指令，还需要先判断它的第5位是1还是0，如果是1，则后5位表示一个立即数，需要对5位进行符号位扩展后与源寄存器1中的数值相加或相与再存到目标寄存器中；如果是0，则后3为表示源寄存器2的编号，需要将两个源寄存器的数值相加或相与再存到目标寄存器中。最后还需要更新条件码。以下为代码实现：

```
void virtual_machine_tp::VM_ADD(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);
}

void virtual_machine_tp::VM_AND(int16_t inst) {
    // TO BE DONE
    // AND指令和ADD指令基本相同，仿照上面的ADD指令的操作即可
    int flag = inst & 0b100000; // 取指令的第5位
    int dr = (inst >> 9) & 0x7; // 取目标寄存器的编号
    int sr1 = (inst >> 6) & 0x7; // 取源寄存器1的编号
    if (flag) {
        // and inst number
        int16_t imm = SignExtend<int16_t, 5>(inst & 0b11111);
```

```
        reg[dr] = reg[sr1] & imm;
    } else {
        // and register
        int sr2 = inst & 0x7;
        reg[dr] = reg[sr1] & reg[sr2];
    }
    // Update condition register
    UpdateCondRegister(dr);
}

void virtual_machine_tp::VM_NOT(int16_t inst) {
    // TO BE DONE
    int16_t dr = (inst >> 9) & 0x7; // 取目标寄存器的编号
    int16_t sr = (inst >> 6) & 0x7; // 取源寄存器的编号
    reg[dr] = ~reg[sr];
    UpdateCondRegister(dr);
}
```

而对于`LEA`指令，则需要先获得目标寄存器的编号和`pc offset`，机器码的后9位表示`pc offset`，对其进行符号位扩展后和PC（即`reg[R_PC]`的值）相加，再将结果存放到目标寄存器中即可。以下为代码实现：

```
void virtual_machine_tp::VM_LEA(int16_t inst) {
    // TO BE DONE
    int16_t dr = (inst >> 9) & 0x7; // 取目标寄存器的编号
    reg[dr] = reg[R_PC] + SignExtend<int16_t,9>(inst & 0x1FF);
}
```

移数指令

移数指令的操作流程为：首先获得目标寄存器或源寄存器的编号，然后获得内存单元的地址，然后将内存中的数值存到目标寄存器中或是将源寄存器中的值存到内存中。下面以取数指令为例（设机器码为inst）：

首先获得inst[11 : 9]，它表示目标寄存器，设为`dr`

- `LD`指令：获得`pcoffset`，即为inst[8 : 0]，然后对其进行符号位扩展后与`pc`相加，得到内存单元的地址。
- `LDR`指令：获得基址寄存器的编号，即为inst[8 : 6]，最后5位表示偏移量，进行符号位扩展后与基址寄存器中的值相加，得到内存单元的地址。
- `LDI`指令：首先根据`LD`指令的逻辑获得第一个内存单元的地址，然后需要判断此时是否发生了`Access violation`，当`PSR[15] = 1 && (address < 0x3000 || address >= 0xFE00)`时会导致`Access violation`，若发生，则报错并结束程序，否则以访问第一个内存单元，以其中的值为第二个内存单元的地址。

在获得了内存单元的地址后，也是需要判断是否发生了`Access violation`，若发生了则报错并结束程序，否则取出内存单元中的数值存放到目标寄存器中，最后再更新条件码即可。以下为代码实现：

```
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);
    int16_t addr = reg[R_PC] + pc_offset;
```

```
    if (reg[R_PSR] >> 15 && (addr < 0x3000 || addr >= 0xFE00)) {
        std::cout << "--- Access violation ---" << std::endl;
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
        exit(0);
    }
    reg[dr] = mem[addr];
    UpdateCondRegister(dr);
}

void virtual_machine_tp::VM_LDI(int16_t inst) {
    // TO BE DONE
    int16_t dr = (inst >> 9) & 0x7; // 取目标寄存器的编号
    int16_t pc_offset = SignExtend<int16_t, 9>(inst & 0x1FF); // 取偏移量
    int16_t addr_1 = reg[R_PC] + pc_offset;
    if (reg[R_PSR] >> 15 && (addr_1 < 0x3000 || addr_1 >= 0xFE00)) {
        std::cout << "--- Access violation ---" << std::endl;
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
        exit(0);
    }
    int16_t addr_2 = mem[addr_1];
    if (reg[R_PSR] >> 15 && (addr_2 < 0x3000 || addr_2 >= 0xFE00)) {
        std::cout << "--- Access violation ---" << std::endl;
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
        exit(0);
    }
    reg[dr] = mem[addr_2];
    UpdateCondRegister(dr);
}

void virtual_machine_tp::VM_LDR(int16_t inst) {
    // TO BE DONE
    int16_t dr = (inst >> 9) & 0x7; // 取目标寄存器的编号
    int16_t br = (inst >> 6) & 0x7; // 取基址寄存器的编号
    int16_t offset = SignExtend<int16_t,6>(inst & 0x3F); // 取基址偏移量并扩展
    int16_t addr = reg[br] + offset;
    if (reg[R_PSR] >> 15 && (addr < 0x3000 || addr >= 0xFE00)) {
        std::cout << "--- Access violation ---" << std::endl;
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
        exit(0);
    }
    reg[dr] = mem[addr];
    UpdateCondRegister(dr);
}
```

存数指令的操作类似，不同是在获得内存单元的地址后，是要将源寄存器中的值存到内存中，并且不需要更新条件码，以下为代码实现：

```
void virtual_machine_tp::VM_ST(int16_t inst) {
    // TO BE DONE
    int16_t sr = (inst >> 9) & 0x7; // 取源寄存器的编号
    int16_t pc_offset = SignExtend<int16_t,9>(inst & 0x1FF);
    int16_t addr = reg[R_PC] + pc_offset;
    if (reg[R_PSR] >> 15 && (addr < 0x3000 || addr >= 0xFE00)) {
        std::cout << "--- Access violation ---" << std::endl;
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
```

```
        exit(0);
    }
    mem[addr] = reg[sr];
}

void virtual_machine_tp::VM_STI(int16_t inst) {
    // TO BE DONE
    int16_t sr = (inst >> 9) & 0x7;
    int16_t pc_offset = SignExtend<int16_t,9>(inst & 0x1FF);
    int16_t addr_1 = reg[R_PC] + pc_offset;
    if (reg[R_PSR] >> 15 && (addr_1 < 0x3000 || addr_1 >= 0xFE00)) {
        std::cout << "--- Access violation ---" << std::endl;
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
        exit(0);
    }
    int16_t addr_2 = mem[addr_1];
    if (reg[R_PSR] >> 15 && (addr_2 < 0x3000 || addr_2 >= 0xFE00)) {
        std::cout << "--- Access violation ---" << std::endl;
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
        exit(0);
    }
    mem[addr_2] = reg[sr];
}

void virtual_machine_tp::VM_STR(int16_t inst) {
    // TO BE DONE
    int16_t sr = (inst >> 9) & 0x7;
    int16_t br = (inst >> 6) & 0x7;
    int16_t offset = SignExtend<int16_t,6>(inst & 0x3F);
    int16_t addr = reg[br] + offset;
    if (reg[R_PSR] >> 15 && (addr < 0x3000 || addr >= 0xFE00)) {
        std::cout << "--- Access violation ---" << std::endl;
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
        exit(0);
    }
    mem[addr] = reg[sr];
}
```

控制指令

- `BR` 指令：首先需要获得跳转条件，即 `nzp` 中的哪几项为真时跳转，也即inst[11 : 9]，然后将其与条件码进行与操作，若结果为真则更新 `pc`，准备跳转，否则不做操作。并且如果 `gIsDetailedMode` 为真的话，还需要打印 `pc` 和 `pcoffset`。以下为代码实现：

```
void virtual_machine_tp::VM_BR(int16_t inst) {
    int16_t pc_offset = SignExtend<int16_t, 9>(inst & 0x1FF);
    //std:: cout << pc_offset << std::endl;
    int16_t cond_flag = (inst >> 9) & 0x7;
    if (gIsDetailedMode) {
        std::cout << reg[R_PC] << std::endl;
        std::cout << pc_offset << std::endl;
    }
    if (cond_flag & reg[R_COND]) {
        reg[R_PC] += pc_offset;
    }
}
```

- `JMP` 指令：首先获得基址寄存器的编号(inst[8 : 6])，然后将基址寄存器的值赋值给 `pc` 即可（`RET` 指令即为 `JMP R7`），以下为代码实现：

```
void virtual_machine_tp::VM_JMP(int16_t inst) {
    // TO BE DONE
    int16_t baseR = (inst >> 6) & 0x7; // 取基址寄存器的编号
    reg[R_PC] = reg[baseR];
}
```

- `JSR` 指令：因为 `JSR` 指令和 `JSRR` 指令的操作码一样，所以首先需要判断是哪一种指令：若inst[11]为1则为 `JSR` 指令，若为0则为 `JSRR` 指令。若为 `JSR` 指令则后11位表示 `pcoffset`，对其进行符号位扩展后与 `pc` 相加，然后赋值给 `pc` 即可；若为 `JSRR` 指令则将基址寄存器（inst[8 : 6）中的值赋值给 `pc`。以下为代码实现：

```
void virtual_machine_tp::VM_JSR(int16_t inst) {
    // TO BE DONE
    // 这里JSR和JSRR一起处理，所以需要先判断是哪一种指令
    int flag = (inst >> 11) & 1; // 取指令的第11位
    if(flag)
    {
        // JSR
        reg[R_PC] += SignExtend<int16_t, 11>(inst & 0x7FF); // PC的值加上偏移量
    }
    else
    {
        // JSRR
        int16_t baseR = (inst >> 6) & 0x7; // 取基址寄存器的编号
        reg[R_PC] = reg[baseR]; // PC的值变为基址寄存器的值
    }
}
```

以上几条跳转指令都没有判断是否会导致 `Access violation`，这是因为在调用它们的函数中会统一判断。

- `TRAP` 指令：因为 `TRAP` 指令非常多，无法一一实现，所以这里只实现框架中列出的五个指令。首先读取指令的后八位，然后根据这八位数值来分别实现不同的功能：
  - `0x20`：表示 `GETC`，即读取一个字符，将其ASCII码存到 `R0` 中即可。
  - `0x21`：表示 `OUT`，将 `R0` 的后八位当做ASCII码，输出对应的字符。

- `0x22`：表示`PUTS`，以`R0`中的数值为基址，输出内存中的数值的后八位作为ASCII码表示的字符，一直输出至内存中的数值的后八位为0结束，最后换行。
- `0x23`：表示`IN`，也是读取一个字符，与`GETC`不同的是它读取完之后还会将字符回显在屏幕上。
- `0x24`：表示`PUTSP`，与`PUTS`基本相同，不同的是它会将一个内存中的低八位和高八位都作为ASCII码输出，先输出低八位，后输出高八位，在低八位为0时结束。
- `0x25`：表示`HALT`，结束程序。

以下为代码实现：

```
void virtual_machine_tp::VM_TRAP(int16_t inst) {
    int trap_num = inst & 0xFF;
    // if (trap_num == 0x25)
    //     exit(0);
    // TODO: build trap program
    char temp;
    if(trap_num == 0x20)
    {
        // GETC
        std::cin >> temp;
        reg[R_R0] = temp;
    }
    else if(trap_num == 0x21)
    {
        // OUT
        temp = reg[R_R0] & 0xFF;// 取R0的第八位作为ASCII码输出
        std::cout << temp << std::endl;
    }
    else if(trap_num == 0x22)
    {
        // PUTS
        int16_t address = reg[R_R0];
        while((temp = mem[address++] & 0xFF) != 0)
        {
            std::cout << temp;
        }
        std::cout << std::endl;
    }
    else if(trap_num == 0x23)
    {
        // IN
        std::cout << "Input a character> ";
        std::cin >> temp;
        std::cout << temp << std::endl;
        reg[R_R0] = temp & 0xFF;
    }
    else if(trap_num == 0x24)
    {
        // PUTSP
        // 与PUTS基本相同，不同的是一个存储单元中存储了两个字符
        int16_t address = reg[R_R0];
        while((temp = mem[address++] & 0xFF) != 0)
        {
            std::cout << temp;
            if((temp = (mem[address] >> 8) & 0xFF))
```

```
            {
                std::cout<< temp ;
            }
        }
        std::cout << std::endl;
    }
    else if(trap_num == 0x25)
    {
        // HALT，此时直接打印结束信息并退出程序即可
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
        exit(0);
    }
}
```

NextStep

首先需要获得当前的`pc`和指令，然后需要判断是否存在`Access violation`，若存在，则报错并结束程序，若不存在则继续执行程序：

首先需要根据指令的高4位确定是哪一条指令，然后调用相应的函数执行指令即可。并且如果`gIsDetailedMode`为真，还需要打印出指令的类型。此外，还需要考虑指令非法的情况，包括未定义的`1101`作为操作码的指令，还有其他对特定要求必须为1或0的指令，如果非法，则报错并结束程序。最后如果指令为0则返回0，否则返回`pc`。以下为代码实现：

```
int16_t virtual_machine_tp::NextStep() {
    int16_t current_pc = reg[R_PC];
    reg[R_PC]++;
    if ((reg[R_PSR] >> 15) & 1 && (current_pc < 0x3000 || current_pc >= 0xFE00))
{
        std::cout << "--- Access Violation ---" << std::endl;
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
        exit(0);
    }
    int16_t current_instruct = mem[current_pc];
    int opcode = (current_instruct >> 12) & 15;

    switch (opcode) {
        case 1101:
        // 考虑opcode非法的情况
        std::cout << "--- Illegal opcode exception ---" << std::endl;
        std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
        exit(0);
        break;
        case O_ADD:
        if (gIsDetailedMode) {
            std::cout << "ADD" << std::endl;
        }
        if ((current_instruct >> 5) & 1 == 0 && (current_instruct >> 3) & 0x3 !=
0) {
            std::cout << "--- Illegal opcode exception ---" << std::endl;
            std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
            exit(0);
        }
```

```
        VM_ADD(current_instruct);
        break;
        case O_AND:
        // TO BE DONE
        if (gIsDetailedMode) {
            std::cout << "AND" << std::endl;
        }
        if ((current_instruct >> 5) & 1 == 0 && (current_instruct >> 3) & 0x3 !=
0) {
            std::cout << "--- Illegal opcode exception ---" << std::endl;
            std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
            exit(0);
        }
        VM_AND(current_instruct);
        break;
        case O_BR:
        // TO BE DONE
        if (gIsDetailedMode) {
            std::cout << "BR" << std::endl;
        }
        VM_BR(current_instruct);
        break;
        case O_JMP:
        // TO BE DONE
        if (gIsDetailedMode) {
            std::cout << "JMP" << std::endl;
        }
        if ((current_instruct >> 6) & 0x3F || (current_instruct >> 9) &0x7) {
            std::cout << "--- Illegal opcode exception ---" << std::endl;
            std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
            exit(0);
        }
        VM_JMP(current_instruct);
        break;
        case O_JSR:
        // TO BE DONE
        if (gIsDetailedMode) {
            if((current_instruct >> 11) & 1)
                std::cout << "JSR" << std::endl;
            else
                std::cout << "JSRR" << std::endl;
        }
        if ((current_instruct >> 11) & 1 == 0 && ((current_instruct >> 6) & 0x3F
|| (current_instruct >> 9) &0x3) ) {
            std::cout << "--- Illegal opcode exception ---" << std::endl;
            std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
            exit(0);
        }
        VM_JSR(current_instruct);
        break;
        case O_LD:
        // TO BE DONE
        if (gIsDetailedMode) {
            std::cout << "LD" << std::endl;
        }
```

```
        VM_LD(current_instruct);
        break;
        case O_LDI:
        // TO BE DONE
        if (gIsDetailedMode) {
            std::cout << "LDI" << std::endl;
        }
        VM_LDI(current_instruct);
        break;
        case O_LDR:
        // TO BE DONE
        if (gIsDetailedMode) {
            std::cout << "LDR" << std::endl;
        }
        VM_LDR(current_instruct);
        break;
        case O_LEA:
        // TO BE DONE
        if (gIsDetailedMode) {
            std::cout << "LEA" << std::endl;
        }
        VM_LEA(current_instruct);
        break;
        case O_NOT:
        // TO BE DONE
        if (gIsDetailedMode) {
            std::cout << "NOT" << std::endl;
        }
        if (current_instruct & 0x3F != 0x3F) {
            std::cout << "--- Illegal opcode exception ---" << std::endl;
            std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
            exit(0);
        }
        VM_NOT(current_instruct);
        break;
        case O_RTI:
        // TO BE DONE
        if (gIsDetailedMode) {
            std::cout << "RTI" << std::endl;
        }
        if (reg[R_PSR] >> 15) {
            // 不能在User mode 下使用RTI指令
            std::cout << "--- Privilege mode violation ---" << std::endl;
            std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
            exit(0);
        }
        if (current_instruct & 0xFFF) {
            std::cout << "--- Illegal opcode exception ---" << std::endl;
            std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
            exit(0);
        }
        VM_RTI(current_instruct);
        break;
        case O_ST:
        // TO BE DONE
```

```
        if (gIsDetailedMode) {
            std::cout << "ST" << std::endl;
        }
        VM_ST(current_instruct);
        break;
        case O_STI:
        // TO BE DONE
        if (gIsDetailedMode) {
            std::cout << "STI" << std::endl;
        }
        VM_STI(current_instruct);
        break;
        case O_STR:
        // TO BE DONE
        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 >> 8) & 0xF) {
            std::cout << "--- Illegal opcode ---" << std::endl;
            std::cout << std::endl << "--- Halting the LC-3 ---" << std::endl;
            exit(0);
        }
        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
        //std::cout << "--- Halting the LC-3 ---" << std::endl;
        return 0;
    }
    return reg[R_PC];
}
```

添加的函数

因为添加的几个函数都很容易实现，所以这里一并介绍：

`ClearMem`：使用一个for循环将所有内存中的值都赋值为0即可。

`ChangeMem`和`ChangeReg`：将指定内存单元或寄存器的值赋值为传入的值即可。

`ShowMem`：依次打印指定范围的内存单元中的值即可，这里我分别打印了16进制形式和2进制形式的值。

`ShowReg`：依次打印所有寄存器中的值即可。

`ShowMode`：若`PSR[15] = 1`则为`User mode`，否则为`Supervisor mode`。

`ChangeMode`：将`PSR[15]`取反即可。

以下为代码实现：

```
void virtual_machine_tp::ClearMem() {
    for(int i = 0;i < 0xFFFF;i++)
        mem[i] = 0;
}

void virtual_machine_tp::ShowMem(int begin, int end) {
    for(;begin <= end;begin++)
    {
        std::cout << "Address: x" << std::setw(4) << std::setfill('0') <<
std::hex << begin;
        std::cout << " Value: x" << std::setw(4) << std::setfill('0') <<
std::hex << mem[begin] << "/b" << std::bitset<16>(mem[begin]) <<std::endl;
    }
}

void virtual_machine_tp::ChangeMem(int address, int16_t value) {
    mem[address] = value;
}

void virtual_machine_tp::ChangeReg(int x,int16_t value)
{
    reg[x] = value;
}

void virtual_machine_tp::ShowReg()
{
    std::cout << "\e[1mR0\e[0m = " << std::setw(4) << std::setfill('0') <<
std::hex << reg[R_R0] << ", ";
    std::cout << "\e[1mR1\e[0m = " << std::setw(4) << std::setfill('0') <<
std::hex << reg[R_R1] << ", ";
    std::cout << "\e[1mR2\e[0m = " << std::setw(4) << std::setfill('0') <<
std::hex << reg[R_R2] << ", ";
    std::cout << "\e[1mR3\e[0m = " << std::setw(4) << std::setfill('0') <<
std::hex << reg[R_R3] << std::endl;
    std::cout << "\e[1mR4\e[0m = " << std::setw(4) << std::setfill('0') <<
std::hex << reg[R_R4] << ", ";
    std::cout << "\e[1mR5\e[0m = " << std::setw(4) << std::setfill('0') <<
std::hex << reg[R_R5] << ", ";
    std::cout << "\e[1mR6\e[0m = " << std::setw(4) << std::setfill('0') <<
std::hex << reg[R_R6] << ", ";
    std::cout << "\e[1mR7\e[0m = " << std::setw(4) << std::setfill('0') <<
std::hex << reg[R_R7] << std::endl;
    std::cout << "\e[1mCOND[NZP]\e[0m = " << std::bitset<3>(reg[R_COND]) <<
std::endl;
```

```
    std::cout << "\e[1mPC\e[0m = " << std::setw(4) << std::setfill('0') <<
std::hex << reg[R_PC] << std::endl;
}

void virtual_machine_tp::ClearReg()
{
    reg[R_R0] = 0;
    reg[R_R1] = 0;
    reg[R_R2] = 0;
    reg[R_R3] = 0;
    reg[R_R4] = 0;
    reg[R_R5] = 0;
    reg[R_R6] = 0;
    reg[R_R7] = 0;
    reg[R_PC] = 0x3000;
    reg[R_COND] = 2;
    reg[R_PSR] = 0x8002;
}

void virtual_machine_tp::ShowMode()
{
    std::cout << "Current mode: ";
    if (reg[R_PSR] >> 15) {
        std::cout << "User mode" << std::endl;
    } else {
        std::cout << "Supervisor mode" << std::endl;
    }
}

void virtual_machine_tp::ChangeMode()
{
    std::cout << "You have successfully changed from ";
    if (reg[R_PSR] >> 15) {
        std::cout << "User mode to Supervisor mode" << std::endl;
        reg[R_PSR] &= 0x7FFF; // 最高位清零，后面的保持不变
    } else {
        std::cout << "Supervisor mode to User mode" << std::endl;
        reg[R_PSR] |= 0x8000; // 最高位置1，后面的保持不变
    }
}
```

其中在打印内存值或寄存器时，为了打印的整齐，我使用了`setw(4)`和`setfill(0)`，保证打印出的16进制数有4位，不足4位时高位补0，这两个函数需要引用`<iomanip>`。

main.cpp

这个函数主要使用了一个循环来依次执行每一条指令，在`NextStep()`函数返回值为0时结束。如果`gIsDetailedMode`为真，每次执行完指令后都会打印所有寄存器中的值；如果`gIsSingleStepMode`为真，每执行完一条指令都会暂停，等待输入。

为了实现查看，修改内存值，寄存器值或当前的模式的功能，我增加了一个函数`void menu(virtual_machine_tp virtual_machine)`，它首先会打印功能菜单，菜单中除了上面提到的六个功能外，还有退出单步调试，退出菜单这两个功能。然后根据用户输入调用相应的函数，执行相应的功能。

我的设计是，在单步调试的每一步先询问用户是否要打印菜单，如果需要打印（这需要读取用户的输入），则调用menu函数，如果不需要打印，则用户可按任意键继续，以下为代码实现:

```
int main(int argc, char **argv) {
    po::options_description desc{"\e[1mLC3
SIMULATOR\e[0m\n\n\e[1mOptions\e[0m"};
    desc.add_options()
                  //
        ("help,h", "Help screen")
                   //
        ("file,f", po::value<std::string>()->default_value("input.txt"), "Input
file")             //
        ("register,r", po::value<std::string>()->default_value("register.txt"),
"Register Status") //
        ("single,s", "Single Step Mode")
                  //
        ("begin,b", po::value<int>()->default_value(0x3000), "Begin address
(0x3000)")
        ("output,o", po::value<std::string>()->default_value(""), "Output file")
        ("detail,d", "Detailed Mode");

    po::variables_map vm;
    store(parse_command_line(argc, argv, desc), vm);
    notify(vm);

    if (vm.count("help")) {
        std::cout << desc << std::endl;
        return 0;
    }
    if (vm.count("file")) {
        gInputFileName = vm["file"].as<std::string>();
    }
    if (vm.count("register")) {
        gRegisterStatusFileName = vm["register"].as<std::string>();
    }
    if (vm.count("single")) {
        gIsSingleStepMode = true;
    }
    if (vm.count("begin")) {
        gBeginningAddress = vm["begin"].as<int>();
    }
    if (vm.count("output")) {
        gOutputFileName = vm["output"].as<std::string>();
    }
    if (vm.count("detail")) {
        gIsDetailedMode = true;
    }

    virtual_machine_tp virtual_machine(gBeginningAddress, gInputFileName,
gRegisterStatusFileName);
```

```
    int halt_flag = true;
    int time_flag = 0;
    char c;
    virtual_machine.ClearReg();
    while(halt_flag) {
        // Single step
        // TO BE DONE
        if (virtual_machine.NextStep() == 0)
            halt_flag = false; //如果后面没有要执行的程序，就可以结束了
        if (gIsDetailedMode)
            //std::cout << virtual_machine.reg << std::endl;
            virtual_machine.ShowReg();
        if (gIsSingleStepMode)
        {
            std::cout << "Do you want to print menu?[y/...]" << std::endl;
            //c = getchar();
            std::cin >> c;
            if( c == 'y')
                menu(virtual_machine);
            std::cout << "Enter any key to continue..." << std::endl;
        }
        ++time_flag;
    }

    std::cout << virtual_machine.reg << std::endl;
    std::cout << "cycle = " << time_flag << std::endl;
    return 0;
}

void menu(virtual_machine_tp virtual_machine)
{
    std::cout << "Please choose the function:" << std::endl;
    std::cout << "0. Clear the memory" << std::endl;
    std::cout << "1. Show the value in memory" << std::endl;
    std::cout << "2. Change the value in memory" << std::endl;
    std::cout << "3. Show the value in register" << std::endl;
    std::cout << "4. Change the value in register" << std::endl;
    std::cout << "5. Show your mode" << std::endl;
    std::cout << "6. Change your mode" << std::endl;
    std::cout << "7. Exit SingleStep mode" << std::endl;
    std::cout << "8. Exit" << std::endl;
    std::cout << "Your choice: ";
    int choice;
    int16_t value;
    char c;
    std::cin >> choice;
    switch (choice)
    {
        case 0:
            virtual_machine.ClearMem();
            break;
        case 1:
            unsigned int begin, end;
            std::cout << "Enter the beginning address in hexadecimal: ";
            std::cin >> std::hex >> begin;
```

```
            std::cout << "Enter the end address in hexadecimal: ";
            std::cin >> std::hex >> end;
            virtual_machine.ShowMem(begin, end);
            break;
        case 2:
            unsigned int address;
            //int value;
            std::cout << "Enter the address in hexadecimal: ";
            std::cin >> std::hex >> address;
            std::cout << "Press 'd' to enter the value in decimal or 'h' to
enter the value in hexadecimal: " << std::endl;
            std::cin >> c;
            if(c == 'd')
            {
            std::cin >> std::dec >> value;
            }
            else if(c == 'h')
            {
                std::cin >> std::hex >> value;
            }
            else
            {
                std::cout << "Invalid input!" << std::endl;
                menu(virtual_machine);
                return ;
            }
            virtual_machine.ChangeMem(address, value);
            break;
        case 3:
            virtual_machine.ShowReg();
            break;
        case 4:
            int reg;
            std::cout << "Enter the register number: ";
            std::cin >> reg;
            std::cout << "Press 'd' to enter the value in decimal or 'h' to
enter the value in hexadecimal: " << std::endl;
            std::cin >> c;
            if(c == 'd')
            {
                std::cin >> std::dec >> value;
            }
            else if(c == 'h')
            {
                std::cin >> std::hex >> value;
            }
            else
            {
                std::cout << "Invalid input!" << std::endl;
                menu(virtual_machine);
                return ;
            }
            virtual_machine.ChangeReg(reg, value);
            break;
        case 5:
```

```
            virtual_machine.ShowMode();
            break;
        case 6:
            virtual_machine.ChangeMode();
            break;
        case 7:
            gIsSingleStepMode = false;
            break;
        case 8:
            return ;
            break;
        default:
            std::cout << "Invalid input!" << std::endl;
            menu(virtual_machine);
        return ;
    }
    std::cout << "Do you want to show this menu again?[y/...]" << std::endl;
    std::cin >> c;
    if(c == 'y')
        menu(virtual_machine);
}
```

实验过程中遇到问题

1. 在上面的main函数中，我先以16进制的形式读取了内存单元的地址address，然后再以10进制形式读取要存放的数值时却发现会被以16进制的形式读取。查阅资料后发现这是因为我在以16进制形式读取完一个数据后，程序会持续以16进制形式读取数据直至下一次改变读取形式。而一般常用的 `std::cin >>` 虽然默认以10进制形式读取，但在以16进制形式读取完一个数据后，它也会以16进制形式读取，所以需要在中间加上 `std::dec`，使数据以10进制形式被读取，这是我以前所没有注意过的。

Results

实验结果如下：

首先是Lab A的三个测试样例执行后的结果：

test 1:

```
STI
R0 = 0000, R1 = 0000, R2 = 0001, R3 = 0000
R4 = 0000, R5 = 0000, R6 = 0000, R7 = 0000
COND[NZP] = 010
PC = 300d
TRAP

--- Halting the LC-3 ---
```

test 2:

```
AND
R0 = ffff, R1 = 0000, R2 = 0000, R3 = 0001
R4 = 0000, R5 = 03ff, R6 = 0000, R7 = 0001
COND[NZP] = 001
PC = 3010
TRAP

--- Halting the LC-3 ---
```

以上两个样例运行的结果均与用LC-3运行的结果一致，说明结果正确。

test 3:

```
TRAP
--- Illegal opcode ---

--- Halting the LC-3 ---
```

因为第一行代码就是`.FILL xffff`，在内存中存储了`xffff`，会被当做指令执行，其操作码是`TRAP`的操作码，但`TRAP`指令要求机器码中间四位均为0，所以这里会报告`Illegal opcode`的错误。

Access violation:

```
BR
12289
-2
R0 = 0000, R1 = 0000, R2 = 0000, R3 = 0000
R4 = 0000, R5 = 0000, R6 = 0000, R7 = 0000
COND[NZP] = 010
PC = 2fff
Do you want to print menu?[y/...]
n
Enter any key to continue...
--- Access Violation ---

--- Halting the LC-3 ---
```

这里因为我在`x3000`处的指令为`BR -2`，并且初始化为`User mode`，所以这里会造成`Access violation`。

Privilege violation:

```
RTI
--- Privilege mode violation ---

--- Halting the LC-3 ---
```

这里因为我在`User mode`下使用了`RTI`指令，所以会有`Privilege mode violation`。

查看指定范围的内存中的值：

```
Your choice: 1
Enter the beginning address in hexadecimal: 3000
Enter the end address in hexadecimal: 3010
Address: x3000 Value: xa00d/b1010000000001101
Address: x3001 Value: xa20d/b1010001000001101
Address: x3002 Value: x14a1/b0001010010100001
Address: x3003 Value: x927f/b1001001001111111
Address: x3004 Value: x1261/b0001001001100001
Address: x3005 Value: x0406/b0000010000000110
Address: x3006 Value: x5602/b0101011000000010
Address: x3007 Value: x0401/b0000010000000001
Address: x3008 Value: x1921/b0001100100100001
Address: x3009 Value: x1482/b0001010010000010
Address: x300a Value: x1261/b0001001001100001
Address: x300b Value: x0ff9/b0000111111111001
Address: x300c Value: xb803/b1011100000000011
Address: x300d Value: xf025/b1111000000100101
Address: x300e Value: x3100/b0011000100000000
Address: x300f Value: x3101/b0011000100000001
Address: x3010 Value: x3102/b0011000100000010
```

修改内存中的值：

```
Your choice: 2
Enter the address in hexadecimal: 3000
Press 'd' to enter the value in decimal or 'h' to enter the value in hexadecimal
:
d
10
Do you want to show this menu again?[y/...]
y
Please choose the function:
0. Clear the memory
1. Show the value in memory
2. Change the value in memory
3. Show the value in register
4. Change the value in register
5. Show your mode
6. Change your mode
7. Exit SingleStep mode
8. Exit
Your choice: 1
Enter the beginning address in hexadecimal: 3000
Enter the end address in hexadecimal: 3000
Address: x3000 Value: x000a/b0000000000001010
```

查看各寄存器中的值：

```
Your choice: 3
R0 = 0000, R1 = 0000, R2 = 0000, R3 = 0000
R4 = 0000, R5 = 0000, R6 = 0000, R7 = 0000
COND[NZP] = 010
PC = 3001
```

修改寄存器中的值：

```
Your choice: 4
Enter the register number: 1
Press 'd' to enter the value in decimal or 'h' to enter the value in hexadecimal
:
d
10
Do you want to show this menu again?[y/...]
y
Please choose the function:
0. Clear the memory
1. Show the value in memory
2. Change the value in memory
3. Show the value in register
4. Change the value in register
5. Show your mode
6. Change your mode
7. Exit SingleStep mode
8. Exit
Your choice: 3
R0 = 0000, R1 = 000a, R2 = 0000, R3 = 0000
R4 = 0000, R5 = 0000, R6 = 0000, R7 = 0000
COND[NZP] = 010
PC = 3001
```

查看当前所处的模式：

```
Your choice: 5
Current mode: User mode
```

修改当前所处的模式：

```
Your choice: 6
You have successfully changed from User mode to Supervisor mode
```

退出单步调试：

```
Your choice: 7
Do you want to show this menu again?[y/...]
n
Enter any key to continue...
LDI
R0 = 0000, R1 = 0000, R2 = 0000, R3 = 0000
R4 = 0000, R5 = 0000, R6 = 0000, R7 = 0000
COND[NZP] = 010
PC = 3002
ADD
R0 = 0000, R1 = 0000, R2 = 0001, R3 = 0000
R4 = 0000, R5 = 0000, R6 = 0000, R7 = 0000
COND[NZP] = 001
PC = 3003
NOT
R0 = 0000, R1 = ffff, R2 = 0001, R3 = 0000
R4 = 0000, R5 = 0000, R6 = 0000, R7 = 0000
COND[NZP] = 100
PC = 3004
```

可以看到，在选择7退出单步调试后，后面的每一条指令都会直接执行，不会再暂停并询问是否要显示菜单。

以上功能均正确执行！