ICS Lab Report - labS

SCGY Cao Gaoxiang PB20000061

January 4, 2022

Lab Name

Simulation Lab

Lab Purpose

Implementing a basic LC-3 simulator.

Lab Content

You will get the following C++ project, containing the basic framework of the program. You need to learn how to use CMake and finish the code.
Here are tasks.

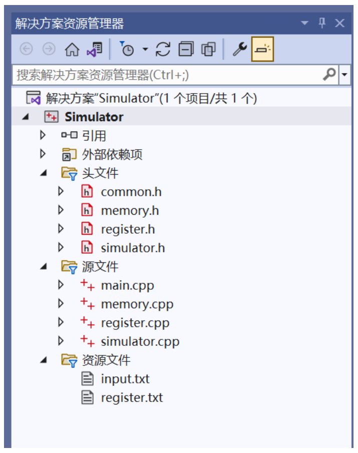
1. Learn how to use CMake.
2. Read the code, understand the framework of the program, and train the ability to read the program.
3. Replace all **TO BE DONE** in the code with the correct code.

Lab Environment

Operating system: Windows 11 Home Edition build 21H2 (with WSL 2), macOS Monterey.
Software: Visual Studio Code, LC3Tools, Visual Studio 2022.

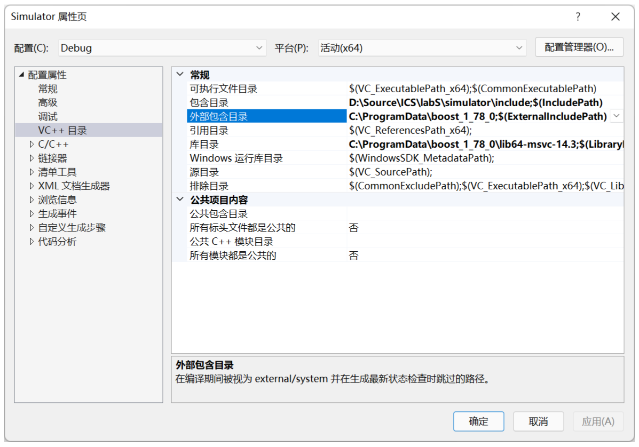
Lab Procedure

1. (a) For Windows, if we don't use WSL, it may be difficult to build the project with CMake, so we use Visual Studio Command Line Application Project to build and debug the program. The hierarchy of the project is as Figure 1(a). Besides, in order to use Boost Library, we need to modify the project setting as Figure 1(b).



```
解决方案"Simulator"(1 个项目/共 1 个)
  Simulator
    引用
    外部依赖项
    头文件
      common.h
      memory.h
      register.h
      simulator.h
    源文件
      main.cpp
      memory.cpp
      register.cpp
      simulator.cpp
    资源文件
      input.txt
      register.txt
```

(a) Project Hierarchy



配置(C): Debug 平台(P): 活动(x64)

常规	
可执行文件目录	$(VC_ExecutablePath_x64);$(CommonExecutablePath)
包含目录	**D:\Source\ICS\labS\simulator\include;$(IncludePath)**
外部包含目录	**C:\ProgramData\boost_1_78_0;$(ExternalIncludePath)**
引用目录	$(VC_ReferencesPath_x64);
库目录	**C:\ProgramData\boost_1_78_0\lib64-msvc-14.3;$(Library**
Windows 运行库目录	$(WindowsSDK_MetadataPath);
源目录	$(VC_SourcePath);
排除目录	$(CommonExcludePath);$(VC_ExecutablePath_x64);$(VC_Lib
公共项目内容	
公共包含目录	
所有标头文件都是公共的	否
公共 C++ 模块目录	
所有模块都是公共的	否

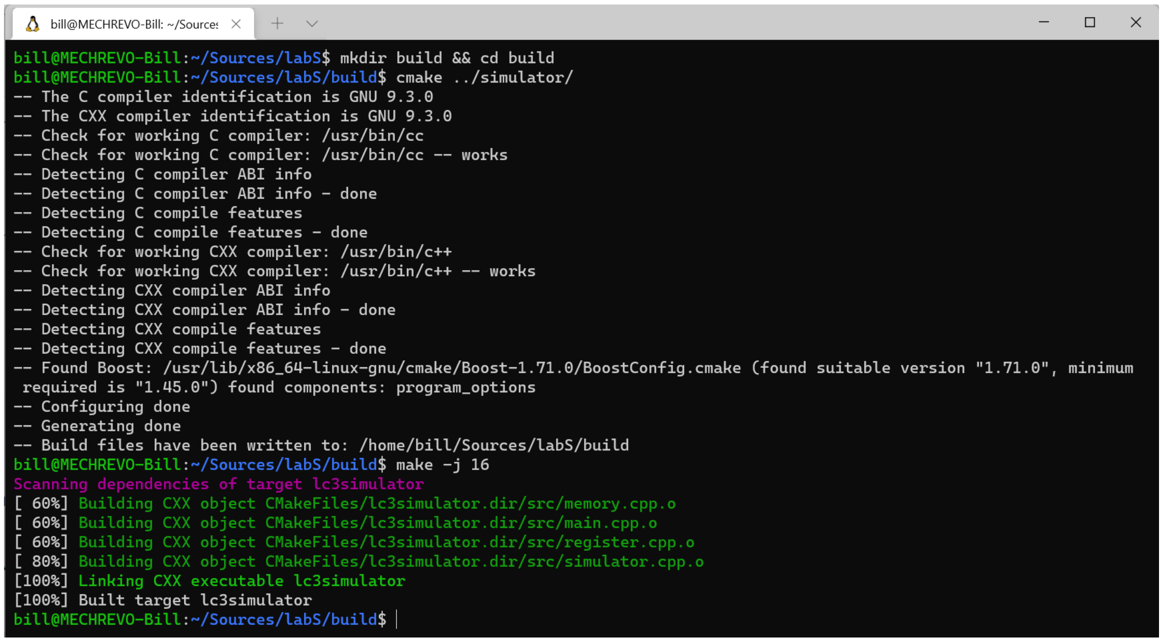
外部包含目录
在编译期间被视为 external/system 并在生成最新状态检查时跳过的路径。

(b) Project Settings

Figure 1: Project Settings

   (b) For UNIX/Linux, it is much easier to build the project with CMake.

```
mkdir build && cd build
cmake ../simulator
make -j 16
```

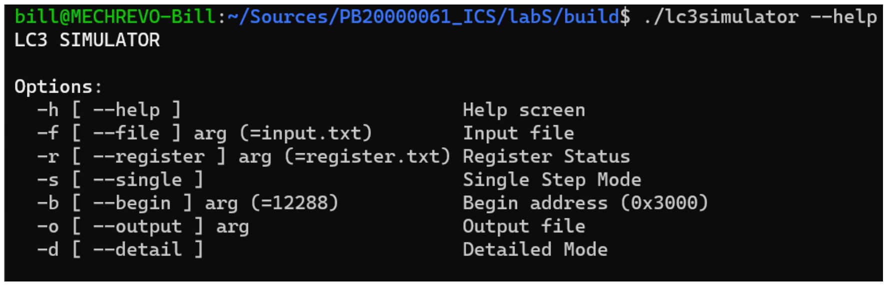


```
bill@MECHREVO-Bill:~/Sources/labS$ mkdir build && cd build
bill@MECHREVO-Bill:~/Sources/labS/build$ cmake ../simulator/
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found Boost: /usr/lib/x86_64-linux-gnu/cmake/Boost-1.71.0/BoostConfig.cmake (found suitable version "1.71.0", minimum
 required is "1.45.0") found components: program_options
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bill/Sources/labS/build
bill@MECHREVO-Bill:~/Sources/labS/build$ make -j 16
Scanning dependencies of target lc3simulator
[ 60%] Building CXX object CMakeFiles/lc3simulator.dir/src/memory.cpp.o
[ 60%] Building CXX object CMakeFiles/lc3simulator.dir/src/main.cpp.o
[ 60%] Building CXX object CMakeFiles/lc3simulator.dir/src/register.cpp.o
[ 80%] Building CXX object CMakeFiles/lc3simulator.dir/src/simulator.cpp.o
[100%] Linking CXX executable lc3simulator
[100%] Built target lc3simulator
bill@MECHREVO-Bill:~/Sources/labS/build$
```

Figure 2: Build the Project with CMake

As Figure 2.

2. First, after building the project, type "./lc3simulator –help" to get a general information about the program, as Figure 3.



```
bill@MECHREVO-Bill:~/Sources/PB20000061_ICS/labS/build$ ./lc3simulator --help
LC3 SIMULATOR

Options:
  -h [ --help ]                      Help screen
  -f [ --file ] arg (=input.txt)     Input file
  -r [ --register ] arg (=register.txt) Register Status
  -s [ --single ]                    Single Step Mode
  -b [ --begin ] arg (=12288)        Begin address (0x3000)
  -o [ --output ] arg                Output file
  -d [ --detail ]                    Detailed Mode
```

Figure 3: Usage of the Simulator

We needn't do anything to the files under directory include/. In src/memory.cpp, we need to read the input file and store them into std::array.

```
void memory_tp::ReadMemoryFromFile(std::string filename,
     int beginning_address) {
     // Read from the file
     if (filename.find(".obj") != filename.npos) {
          std::ifstream input(filename,
               std::ios::binary | std::ios::in);
          while (!input.eof()) {
               input.read((char*)(memory + (beginning_address++)),
                    sizeof(int16_t));
               if (beginning_address > kVirtualMachineMemorySize)
                    return;
          }
     }
     else {
        std::ifstream input(filename);
        int cnt = 0;
        char c;
        int16_t current = 0, current_cnt = 0;
        while (!input.eof()) {
            c = input.get();
            if (c == '0' || c == '1') {
                ++current_cnt;
                current = (current << 1) | (c - '0');
            }
            if (current_cnt == 16) {
                current_cnt = 0;
                memory[beginning_address++] = current;
                if (beginning_address > kVirtualMachineMemorySize)
                    return;
            }
        }
    }
}

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];
}
```

The function will ignore irrelevant characters.

In src/main.cpp, we need to add support for single-step running.

```
virtual_machine_tp virtual_machine(gBeginningAddress,
    gInputFileName, gRegisterStatusFileName);
int halt_flag = true;
int time_flag = 0;
while(halt_flag) {
    // Single step
    halt_flag = virtual_machine.NextStep();
    if (gIsDetailedMode)
        std::cout << virtual_machine.reg << std::endl;
    ++time_flag;
    if (gIsSingleStepMode) std::cin.get();
}
```

Our main work is in src/simulator.cpp. The first one is a template funtion *SignExtend*, which is used many times. We uses bitwise operation to calculate it.

```
template <typename T, unsigned B>
inline T SignExtend(const T x) {
    // Extend the number
    return (x & (1 << (B - 1))) ?
        (((((1 << (sizeof(T) * 8)) - 1) >> B) << B)
        | (x % (1 << B))) : (x % (1 << B));
}
```

From reading the code of VM_BR we can understand how to handle conditional code.

```
void virtual_machine_tp::UpdateCondRegister(int regname) {
    // Update the condition register
    if (reg[regname] > 0) reg[R_COND] = 1;
    else if (reg[regname] == 0) reg[R_COND] = 2;
    else reg[R_COND] = 4;
}
```

The rest of the operation is to implement each of the basic instructions of LC-3. Fortunately, many of them are similar, such as ADD and AND, LD and ST, LDI and STI, LDR and STR. Here we only present the functions wriiten by us.

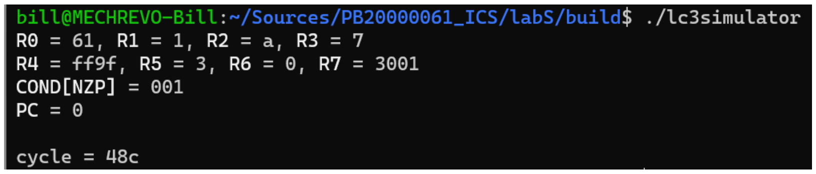
```
void virtual_machine_tp::VM_AND(int16_t inst) {
    int flag = inst & 0b100000;
    int dr = (inst >> 9) & 0x7;
    int sr1 = (inst >> 6) & 0x7;
    if (flag) {
        // and inst number
        int16_t imm = SignExtend<int16_t, 5>(inst & 0b11111);
        reg[dr] = reg[sr1] & imm;
    } else {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

void virtual_machine_tp::VM_JMP(int16_t inst) {
    int BaseR = (inst >> 6) & 7;
    reg[R_PC] = reg[BaseR];
}

void virtual_machine_tp::VM_JSR(int16_t inst) {
    reg[R_R7] = reg[R_PC];
    bool flag = (inst >> 11) & 1;
    if (flag) reg[R_PC] += SignExtend<int16_t, 11>(inst); // JSR
    else reg[R_PC] = mem[reg[(inst >> 6) & 7]]; // JSRR
}

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

void virtual_machine_tp::VM_LDR(int16_t inst) {
    int16_t dr = (inst >> 9) & 0x7;
    int16_t baser = (inst >> 6) & 0x7;
    reg[dr] = mem[reg[baser] + SignExtend<int16_t, 6>(inst)];
    UpdateCondRegister(dr);
}

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

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

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

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

For the rest part of the code, they can be finished the same as TA's code of case ADD.

Correctness Verification

Because the simulator can only read *.bin* files, we first use assembler in labA to convert assembly code written before into binary codes, then use LC3Tools to compare the its result with ours.
The following figure shows the result after running the simulator with the code of lab5, and the input R0 is 97 in decimal.



```
bill@MECHREVO-Bill:~/Sources/PB20000061_ICS/labS/build$ ./lc3simulator
R0 = 61, R1 = 1, R2 = a, R3 = 7
R4 = ff9f, R5 = 3, R6 = 0, R7 = 3001
COND[NZP] = 001
PC = 0

cycle = 48c
```

Figure 4: One Test Case