LabA: Writing a LC-3 Assembler

September 25, 2024

1 Introduction

In this lab, you are required to implement a simple LC-3 assembler. Your assembler should take LC-3 assembly code as input and output the translated binary code in text form. The purpose of this lab is to:

- Understand the basic syntax and instruction set of LC-3 assembly language.
- Understand how an assembler works.
- Learn the basic usage of CMake and improve the ability to program in C or C++.

We provide a basic framework. We have completed many complex and unimportant tasks in the framework and provided many test cases. You only need to complete the core part of the assembler. **You can also write the assembler from scratch using any programming language you like.**

2 Environment Setup

If you choose to use the framework we provide, you can set up your environment as follows.

2.1 Configure the C and C++ Compilation Environment

The framework is written in C++ and provides C bindings, so you can use C or C++ to complete the lab. You need a C++ compiler to compile the framework code. If you choose to use C to complete the lab, you also need a C compiler.

On Windows, you can use the `MSVC` compiler toolset to compile C and C++ code. We recommend you to install Visual Studio, which will serve as your IDE and install `MSVC` and other tools you may need and configure the environment for you. You can read this tutorial to learn how to install and configure Visual Studio.

On Linux, you can use the `gcc`/`g++` or `clang`/`clang++` compiler to compile C and C++ code. You can install the `build-essential` package, which will install the `gcc`/`g++` compiler and other tools you may need:

```
sudo apt-get install build-essential
```

Although our framework supports you to complete the lab on Windows, we still recommend you to finish it on Linux. The error diagnosis of `MSVC` may not be as friendly as `gcc` and `clang`, and it is easier to configure the development environment on Linux.

2.2 Install CMake

We use CMake to manage the compilation process. On Windows, you can download the installer from the official website. You can read this tutorial to learn how to install and configure CMake on Windows. On Linux, you can install CMake using the package manager:

```
sudo apt-get install cmake
```

2.3 Setup your IDE (Optional)

We recommend you to use an IDE to complete the lab, as an IDE can provide you with many convenient features, such as code completion and debugging. Many mature IDEs also support CMake projects, so you can easily import the framework code we provide.

On Windows, Visual Studio is a good choice. You can install and configure the compiler toolchain when installing Visual Studio. CLion is a powerful cross-platform IDE that provides detailed and rich code suggestions and error diagnosis. You can also use CLion on Linux. We also recommend using Visual Studio Code, a lightweight cross-platform editor that supports rich plugins and can meet most development needs. We recommend you to use the clangd plugin to provide code completion and error diagnosis functions.

2.4 Configure and Compile the Framework

You need to download the framework code according to the instructions of the TA and unzip it to your working directory. Assume that your code is stored in the `laba` directory.

If you want to use **C++** to complete the lab, you need to execute the following commands to configure the framework code:

```
cd laba
cmake -S . -B build -DUSE_CPP=ON
```

On Windows, you may need to replace `cmake` with `cmake.exe`.

On the other hand, if you want to use **C** to complete the lab, you need to execute the following commands to configure the framework code:

```
cd laba
cmake -S . -B build -DUSE_CPP=OFF
```

For the above two commands, `build` is the build directory generated by CMake. You can specify the name of the build directory you like. CMake will put the files generated during the build process in this directory.

If CMake reports an error during this process, you may need to read the error message carefully and check your compilation environment. We hope you can search for error messages and try to solve the problem, so that you can better understand the compilation process. If you cannot solve it independently, we also welcome you to seek help from the TA.

You can enable optimization compilation by adding the `-DCMAKE_BUILD_TYPE=Release` option, which can improve the speed of the program. However, compiler optimization may make debugging difficult, so we recommend you to use the `Debug` mode when debugging. In addition, if you find that your program behaves differently before and after optimization, it is very likely that there is undefined behavior in your code.

If you use `clangd`, you may need to add the `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` option to generate the `compile_commands.json` file.

After configuring the framework code, you can compile it by executing the following command:

```
cmake --build build
```

You can add the `-j` option to enable multi-threaded compilation. **You can find the executable file in the `build` directory, named `lc3-assembler`.**

During the build process, you may see some warnings, such as unused variables. This is because you have not completed the framework code yet, so you don't need to worry about these warnings. After you complete the lab, your code should not generate any warnings.

For each modification of the code, you need to execute the `cmake --build build` command to recompile the code.

3 Framework Introduction

The structure of the framework code is as follows:

```
.
|-- ./CMakeLists.txt
|-- ./.clang-format
|-- ./cmake
|-- ./include
|   |-- ./include/assembler
|   `-- ./include/assembler-c
|-- ./solution
|   |-- ./solution/solution.c
|   `-- ./solution/solution.cpp
|-- ./src
|-- ./test
|   |-- ./test/input
|   `-- ./test/output
`-- ./unittest
    |-- ./unittest/instruction_test.cpp
    `-- ./unittest/parser_test.cpp
```

- `CMakeLists.txt`: The root CMake configuration file.
- `.clang-format`: The `clang-format` configuration file. You can use `clang-format` to format your code, but it is not required.
- `cmake` folder: Contains the CMake scripts for adding third-party dependencies, running tests, etc. Note that all the third-party dependencies have been added to the framework code, so that you do not need to install them manually.
- `include` folder: Header files. `assembler` folder contains the header files of the assembler, while `assembler-c` folder contains the header files of the C bindings. For students who use C to complete the lab, you should only use the header files in the `assembler-c` folder.
- `solution` folder: **This is where you should implement your assembler.** We have provided the declarations of the functions you need to implement, as well as detailed documentation to guide your implementation. **For students who use C to complete the lab, you should implement the functions in `solution.c`. Otherwise, you should implement the functions in `solution.cpp`.** If you modify the file that does not match the language you choose, your changes will not be compiled.

- `src` folder: The implementation of the framework. You do not need to modify the code, but you can read it to understand the framework.
- `test` and `unittest` folder: Contain the test cases.

3.1 Language Standard and Third-party Dependencies

The framework of this lab only uses C++11 features, so mainstream compilers are sufficient to compile our code. However, you can use a higher version of the C/C++ standard to enjoy more features as long as your compiler supports the features you are using. If you want to use a higher version of the C or C++ standard, you should modify the `CMakeLists.txt` file in the root directory of the project and set the `CMAKE_CXX_STANDARD` and `CMAKE_C_STANDARD` variables. For example, you can use the `std::ranges` library feature by setting `CMAKE_CXX_STANDARD` to 20:

```
set(CMAKE_CXX_STANDARD 20)
```

Please note that you should ensure that your compiler supports the C++ standard you are using.

Our framework uses GoogleTest to provide unit test support, and CLI11 to provide command line argument parsing support. The source code of these third-party dependencies has been included in the project, so you do not need to install and configure them. These dependencies are only used to implement the framework. When you complete the lab, **you should not use any other third-party libraries**. **You only need to use the C/C++ standard library to complete the lab.**

3.2 Test Your Code

During your lab, you may need to test the correctness of your code frequently. We provide you with some test cases that you can use to verify the correctness of your code. Although our test cases cover most cases, you can still write more test cases to verify your code.

Our code contains two folders, `unittest` and `test`. The `unittest` folder contains unit test cases supported by GoogleTest. If you are familiar with GoogleTest, you can write more unit test cases to verify your code. The `test` folder contains two subfolders, `input` and `output`. The `input` folder contains some LC-3 assembly code files, which will be used as input to your assembler, and the test will pass only if the content of the corresponding file in the `output` folder is consistent with the output of the assembler.

3.2.1 Run the Test

You can run the test using the following command:

```
cd build        # Enter the build directory
cmake --build . # Ensure that you have built the project
                # before running the test.
ctest
```

On Windows, you may need to replace `ctest` with `ctest.exe`.

You may get the following output:

```
Test project laba/build
      Start  1: ParserTest.TokenEOLAndEOF
 1/49 Test  #1: ParserTest.TokenEOLAndEOF ................   Passed    0.00 sec
...
```

```
      Start 49: parse-operand-list5
49/49 Test #49: parse-operand-list5 .......................***Failed    0.01 sec

27% tests passed, 36 tests failed out of 49

Total Test time (real) =   0.47 sec

The following tests FAILED:
          7 - ParserTest.TokenStringLiteral (Failed)
...
         49 - parse-operand-list5 (Failed)
Errors while running CTest
Output from these tests are in: ...
Use "--rerun-failed --output-on-failure" to re-run the failed cases verbosely.
```

You may notice that `ctest` reports that 36 out of 49 tests failed. This is because you have not completed the lab yet. You can run

```
ctest --output-on-failure
```

to view detailed output of the failed tests. You can run

```
ctest -R <test_name>
```

to run a specific test case.

3.2.2 Write Your Own Test

If you are familiar with GoogleTest, you can write more unit test cases in the `unittest` folder. A simpler way is to write more LC-3 assembly code files in the `test/input` folder, and write the expected output in the `test/output` folder. For example, you can create a file named `example.in` in the `test /input` folder, with the following content:

```
.ORIG x3000
ADD R1, R2, R3
.END
```

Then create a file named `example.out` in the `test/output` folder, with the following content:

```
(3000) 0001001010000011
```

i.e., the machine code of this `ADD` instruction.

Each time you add a new test case, you need to reconfigure CMake and run the test:

```
cd build            # Enter the build directory
cmake ..            # Reconfigure CMake
ctest
ctest -R example    # You can only run the test case named example.
```

and you will see whether your new test case has passed in the test results.

3.2.3 Run the Assembler

You can find the generated executable file `lc3-assembler` in the `build` directory. You can run the assembler directly and observe its output. For example, given an LC-3 assembly code file named `example.asm`, you can run the following command:

```
./lc3-assembler example.asm
```

The assembler will translate the LC-3 assembly code in `example.asm` into binary code and output it to the standard output. Note that on Windows, you may need to replace `./lc3-assembler` with `lc3-assembler.exe`.

4 Step by Step Completion of the Lab

Our assembler needs to process LC-3 assembly code in text form and translate it into binary form. Processing strings is a complex and error-prone process, so we divide it into several steps to complete it step by step. In each step, we leave some unimplemented parts for you to complete. We hope you can complete this lab step by step like reading a continuous story.

4.1 Step 1: Tokenize LC-3 Assembly Code

So how do we convert a string into a series of "instructions"? The LC-3 assembly code text entered by the user may contain comments, any number of spaces and tabs, any number of blank lines, and even strange syntax errors. So our first step is to break down the source code into individual "**token**s", which allows us to eliminate the interference of these irrelevant factors and extract the instructions we need.

4.1.1 What is a Token?

Token is a concept in compilers, representing the smallest unit in the source code. You can think of a token as a **word** in the source code. Our assembler will not directly process a character in the source code, but a whole token. In this step, the assembler needs to extract meaningful words from the source code and identify the specific types of these words, such as labels, instructions, operands, etc. The assembler will ignore useless spaces, tabs, comments, etc., so that we will not be disturbed by these contents in the subsequent steps.

For example, for the following LC-3 assembly code:

```
LOOP AND R3, R3, #0,  ; Clear R3
```

Our assembler will break it down into the following tokens. You may notice that we ignore comments

Word	`LOOP`	`AND`	`R3`	`,`	`R3`	`,`	`#0`	`,`	`EOF`
Type	`Label`	`Opcode`	`Register`	`Comma`	`Register`	`Comma`	`Immediate`	`Comma`	`End`

and spaces, and divide the remaining content into meaningful words. For example, we can easily find that this is an instruction with a Label `LOOP`, and the opcode is `Opcode`. We can parse all the operands of this instruction with simple logic, without having to process comments and spaces while parsing operands. We can also find that there is an extra comma at the end of the instruction, so our assembler can notice that this is an instruction with a syntax error. Finally, you will notice that we generate an additional `End` token, indicating the end of the source code.

We have predefined all the token types that may be used in the assembler. You can find them in `include/assembler/token.hpp` (for C++) or `include/assembler-c/token.h` (for C). You only need to understand what kind of word each token type represents by reading the comments.

4.1.2 Your Task: Parse Tokens of Specific Types

In this step, you do not need to implement a complete parser, since it is not the core part of the lab. On the contrary, we have implemented most of the parser's functions for you, including the parser's driver and the parsing of some complex tokens. Furthermore, in this step, you do not need to care about the syntax rules of LC-3 assembly code. **You are only required to implement the functions that parse two simple token types:**

- `parse_decimal_number()`: Parse a decimal signed integer, such as `+13`, `-42`, and `100`.
- `parse_string_literal()`: Parse a string literal, such as `"Hello World"`.

We have defined the prototypes of these two functions in `solution/solution.cpp` and `solution/solution.c`. Please implement one of them according to the programming language you choose. We have also attached detailed documentation comments in the code to describe the functions, and provided some example inputs and outputs to guide your implementation.

It is worth noting that for `parse_decimal_number()`, **you do not need to convert the textual representation of a decimal integer to the corresponding integer value**. This function is only used to inform the parser of the end position of the decimal integer token, without the need to parse the value of the integer.

Since you are directly processing strings in this step, you may fail the test due to processing too few or too many characters, or ignoring boundary conditions. In addition to the test methods mentioned in Section 3.2, you can also run `lc3-assembler` directly to observe which tokens the assembler has generated:

```
./lc3-assembler example.asm --tokens
```

Here, the `--tokens` option is used to instruct `lc3-assembler` to output all the tokens it sees and stop. If you find that the list of tokens output is different from what you expected, you need to check whether you have correctly implemented these two functions.

If you have correctly implemented these two functions, you should be able to pass 4 **additional** tests:

```
 7 - ParserTest.TokenStringLiteral     8 - ParserTest.TokenImmediateAndNumber
 9 - ParserTest.TokenComment          11 - ParserTest.TokenMixed
```

4.2 Step 2: Package Tokens into Instructions

As we mentioned earlier, our assembler processes the source code in units of tokens. In the first step, we have broken down the source code into a series of tokens. When completing this step, you can forget about irrelevant content such as spaces and comments, and you do not need to process characters in the source code.

In this step, we need to convert a sequence of tokens into an instruction. In `solution/solution.cpp` and `solution/solution.c`, we have defined the `parse_instruction()` function and provided part of its implementation. This function is used to parse from the current token until a complete instruction is parsed and returned. Specifically, we need to parse whether the instruction has a label, the content of the label, the opcode of the instruction, and all the operands of the instruction. **You need to complete this function according to the documentation comments we provide.**

Although you do not need to write the code to parse the operand list (we have provided the `Parser::parse_operand_list()` function for students using C++, and the `parser_parse_operand_list()` function for students using C), we still need to consider how to convert a token into an actual operand. For example, we need to convert a token of type `Register` with content `R0` into an actual register number (i.e., the integer `0`). We have implemented the conversion of most operands from tokens, but we have left you with a small challenge: **You are required to implement the `string_to_integer()` function**, which converts the textual representation of an `Immediate` or `Number` token into the corresponding integer value. You need to handle the prefixes that an immediate may have (`#`, `x`, and `b`), handle the possible positive and negative signs, and return an error when the conversion result overflows. We have defined the prototype of this function in `solution/solution.cpp` and `solution/solution.c`, and you need to implement this function according to the documentation comments we provide.

When you complete the two functions required in this step, you can test whether your assembler can correctly generate the instruction sequence. In addition to the test methods mentioned in Section 3.2, you can also run `lc3-assembler` directly to observe which instructions the assembler has generated:

```
./lc3-assembler example.asm --instructions
```

Here, the `--instructions` option is used to instruct `lc3-assembler` to output all the instructions it sees and stop. If you find that the list of instructions output is different from what you expected, you need to check whether you have correctly implemented these two functions. In addition, **the assembler always outputs immediate numbers in decimal form**, so you need to correctly convert the immediates you use in the input to decimal to observe whether they are consistent.

If you have correctly implemented these two functions, you should be able to pass 18 **additional** tests:

```
13 - InstructionTest.AddIntegerOperand  20 - as-check-orig1              21 - as-check-orig2
28 - instr-check-label1                 29 - instr-check-label2          30 - instr-check-operand-count1
31 - instr-check-operand-count2         32 - instr-check-operand-count3  35 - instr-check-operand-range3
39 - instr-check-operand-type1          40 - instr-check-operand-type2   41 - instr-check-operand-type3
44 - parse-label2                       45 - parse-operand-list1         46 - parse-operand-list2
47 - parse-operand-list3                48 - parse-operand-list4         49 - parse-operand-list5
```

4.3 Step 3: Check the Legality of Instructions

Now we have the instruction sequence. Before translating these instructions into binary code, we need to check whether these instructions are legal. Since checking the legality of instructions is not the core task of this lab, we have implemented most of the checking logic for you. You can find our checking logic for operands in `Instruction::add_operand()`, and the `string_to_integer()` you implemented in the previous step will be called here. You can also find our checking logic for the entire instruction in `Instruction::validate_and_emit_diagnostics()`, which includes checking whether the number of operands of the instruction is correct, whether the types of operands are correct, etc.

In this step, **you need to implement `Instruction::immediate_range()` (for students using C++) or `get_instruction_immediate_range()` (for students using C)**, which is a small part of the logic of `Instruction::validate_and_emit_diagnostics()` function, to deepen your understanding of the legal range that a binary integer of a specific number of bits can represent. They have been defined in `solution/solution.cpp` and `solution/solution.c` respectively.

This function returns the legal range of the immediate operand in an instruction. For example, the `ADD` and `AND` instructions allow their third operand to be an immediate number, and reserve 5 bits for it. You need to calculate the range that a 5-bit signed integer can represent and return it through this function. You may need to read page 656 of the textbook to understand which instructions allow immediate operands and the number of bits of the corresponding operands. You do not need to handle instructions that do not have immediate operands, and instructions that accept a label as an operand and need to calculate the PC-relative offset.

If you have correctly implemented this function, you should be able to pass 4 **additional** tests:

```
33 - instr-check-operand-range1     34 - instr-check-operand-range2
37 - instr-check-operand-range5     38 - instr-check-operand-range6
```

4.4 Step 4: Assign Addresses to Instructions and Scan Labels

We finally arrive at the core part of the assembler. We have done enough preparation to make it easy to implement this step. You should read sections 7.3.2 and 7.3.3 of the textbook before completing this step to understand what we need to do.

4.4.1 Compute the Address of Instructions

Our goal is to scan all the labels of the instructions and build a mapping table (**symbol table**) from the label to the address where the label is located. To do this, we first calculate the address of each instruction. In general, an instruction immediately follows the previous instruction, i.e., the addresses of the two instructions differ by one word. Here we list some situations that will affect the address of the instruction:

- The `.ORIG` pseudo-instruction specifies the starting address of the program. In our implementation, we set the address of the `.ORIG` pseudo-instruction itself and the address of the next instruction to the address specified by the `.ORIG` pseudo-instruction. For example, for the following code:

  ```
  .ORIG x3000     ; Instruction 1
  ADD R0, R0, #0  ; Instruction 2
  ADD R1, R2, R3  ; Instruction 3
  ```

  the addresses of instructions 1 and 2 are both `x3000`, and the address of instruction 3 is `x3001`, which follows the normal rule.

  When implementing this function, you can assume that the first element of the instruction sequence is definitely the `.ORIG` pseudo-instruction, and there will be no other `.ORIG` pseudo-instructions in the instruction sequence.

- The `.BLKW` pseudo-instruction allocates a certain amount of memory based on the current address. This will affect the address of the instruction following the `.BLKW` pseudo-instruction.

You need to carefully understand the meaning of instructions and pseudo-instructions to find all factors that will affect the address of the next instruction.

You need to implement the `Assembler::assign_addresses()` function (for students using C++) or the `assign_addresses()` function (for students using C), which have been defined in `solution/solution.cpp` and `solution/solution.c`, respectively.

We have provided corresponding test cases to check whether your address assignment is correct. In addition to the test methods mentioned in Section 3.2, you can also run `lc3-assembler` directly to observe the addresses of the instructions:

```
./lc3-assembler example.asm
```

We will output the address of each instruction in hexadecimal before each binary instruction. However, since you have not implemented the specific translation function yet, you will see incorrect binary instructions. Don't worry, we will implement this feature soon.

4.4.2 Scan Labels

When you have calculated the address of each instruction, you will find that scanning labels and building the symbol table becomes very simple. You only need to traverse each instruction, and if you find that an instruction has a label, you should take out the address of this instruction that has been calculated, and store the label and address in the symbol table. However, you also need to check whether a label is redefined and report an error on the interface provided by the framework. We have defined the `Assembler::scan_label()` function (for students using C++) and the `scan_label()` function (for students using C), in `solution/solution.cpp` and `solution/solution.c`, respectively.

When you have completed this step, you should be able to pass 1 **additional** test:

```
23 - as-scan-label2
```

4.5 Step 5: Translate Instructions into Binary Code

Now you have come to the most difficult part of the entire lab. In this step, you need to implement several small functions to convert each part of the instruction. Finally, you need to implement two larger functions that call these small functions to complete the translation of the instruction. It is worth noting that **all functions in this step work under the `uint16_t` type, i.e., 16-bit unsigned integers**. Although our assembler ultimately outputs binary code in text form, you do not need to process text output but only need to store the binary code in a variable of type `uint16_t`. In this process, you will use many bitwise operations, so make sure you are familiar with the meaning and usage of each bitwise operation.

We handle regular instructions and pseudo-instructions separately, because each regular instruction is strictly translated into a `uint16_t`, while pseudo-instructions are not. For example, `.BLKW` needs to generate multiple `uint16_t`, while `.FILL` only needs to generate one `uint16_t`.

4.5.1 Translate Regular Instructions

We divide a regular instruction such as `ADD` and `NOT` into two parts: the **opcode** and the **operands**. The operands may have different types, such as a register, an immediate number, or a label.

In LC-3 assembly code, the opcode of an instruction consists of 4 bits. For example, the opcode of the `ADD` instruction is `0001`, which is 1 in decimal. We have defined the `translate_opcode()` function for this conversion. It is quite simple: you only need to return the corresponding 4-bit integer value according to the given opcode enumeration. We have implemented half of the opcode conversion work for you to reduce your workload. You can refer to our implementation to complete the remaining opcode conversions. You can find all the opcodes in `include/assembler/opcode.def`.

Next, you need to implement the translation of operands. If you have read the code in `include/assembler/opcode.hpp`, you will notice that we store different types of operands in the same structure and need to check whether an operand is "really this type" before accessing a specific type of operand.

However, you do not need to perform such checks here, because we have checked the legality of the instruction. This means that you can assume that, for example, the first operand of the `ADD` instruction is definitely a register operand, without further checks at runtime.

In LC-3, a register operand is encoded in 3 bits. Considering that the binary encoding of a register operand may be placed in different positions, you also need to move the encoding of the register operand to the correct position. We have defined the `translate_register()` function for this conversion. You can follow the documentation comments to implement it.

Similarly, immediate operands are always placed at the end of the instruction, but they need to be truncated to meet the bit requirements of the instruction for immediate operands. You need to implement the `translate_immediate()` function to complete this conversion.

Finally, you need to implement the `translate_label()` function to complete the translation of labels. This function is slightly more complex, and you need to complete the following tasks:

- Check whether the label exists in the symbol table, and report an error if it does not.
- Get the address of the label and correctly calculate the PC-relative offset.
- Check whether this offset is within the legal range according to the number of bits required by the instruction.
- Truncate this offset to the required number of bits and return it.

When you have completed the implementation of all these small functions, you need to implement the `translate_regular_instruction()` function to complete the translation of a complete regular instruction. You need to call the 4 small functions you implemented above to convert each part of an instruction. We have provided the implementation of the translation of some regular instructions, and you can refer to them to complete the translation of the remaining instructions.

4.5.2 Translate Pseudo-instructions

The translation of pseudo-instructions is relatively easy, since we do not need to divide them into different parts for translation. We choose to generate the corresponding translation results directly based on the opcode of the pseudo-instruction:

- For the `.ORIG` and `.END` pseudo-instructions, no translation result needs to be generated.
- For the `.FILL` pseudo-instruction, the binary form of the first operand is directly used as the translation result.
- For the `.BLKW` and `.STRINGZ` pseudo-instructions, multiple translation results are generated according to the meaning of the instruction.

You need to implement the `translate_pseudo()` function according to the above logic. Please read the documentation comments of this function to understand how to handle its input and output.

You have now completed your assembler. You should be able to pass all the test cases, including those we provide and those you write yourself. You can also run `lc3-assembler` directly, provide it with an LC-3 assembly code file, and observe whether its output is correct.

5 Tips

Here are some important tips for you:

- In this lab, the correctness of your implementation accounts for 50% of the score, and the report accounts for the other 50%. If you use our lab framework, you need to ensure that your implementation passes all test cases, including those we provide and those you write yourself.
- You should pay attention to your coding style. You should use appropriate variable names, function names, indentation, etc. For complex parts, you need to include some comments to explain your code. You can also use `clang-format` to format your code, but this is not required.
- If you use our lab framework, we have enabled the highest warning level for you. Please pay attention to every warning message generated by the compiler, as they may indicate potential problems in your code.
- If you use our lab framework, you should not introduce any third-party libraries. You can only use the C/C++ standard library.
- You should explain your implementation ideas in the lab report, including the implementation ideas of each part and explanations of key parts of the code.
- We do not have strict requirements on the format of your lab report, but you should ensure that your report is clear, readable, and formatted consistently.

If you DO NOT use our lab framework and choose to implement it from scratch, you need to pay attention to the following points:

- You can use third-party libraries appropriately, but you need to explain in the lab report which libraries you used. The core part of the assembler must be implemented by yourself.
- We do not limit the build system you use. You can use CMake, `make`, or other build systems, or directly use the compiler for compilation. You need to explain how your code is built in the report. If your project uses third-party libraries, you need to explain how to configure these libraries in the report, or you can include the source code of these libraries directly in your project, as we do in our framework.
- You must include detailed comments in your code to explain your logic.
- You need to write test cases to verify the correctness of your code.

  It is worth noting that we do not strictly define the syntax of LC-3, so different implementations may differ in some details. Therefore, your implementation does not need to pass all the test cases we provide in the lab framework. You can explain your design choices in the lab report.
- Your assembler must be able to accept file input and output, i.e., read LC-3 assembly code from a file and output the binary code to a file. You need to handle command line arguments and explain how to use your assembler in your report.
- Your assembler must be able to output the translated binary code and its address in text form at least.
- Your assembler must be able to exit with an error return value when there are errors in the input. We recommend you to implement simple diagnostic information reporting.

6 Submission

If you use our lab framework, you need to organize your submission content as follows:

```
PB********_Name_laba.zip
|-- ./PB********_Name_report.pdf
`-- ./solution.c / ./solution.cpp
```

You need to submit the correct file according to the programming language you use. If you use C, you need to submit the `solution.c` file; if you use C++, you need to submit the `solution.cpp` file. **You do not need to submit other files in the framework or the compiled files.**

If you choose to implement it from scratch, you need to organize your submission content as follows:

```
PB********_Name_laba.zip
|-- ./PB********_Name_report.pdf
`-- Your code structure
```

Note that you should NOT include build artifacts such as binary files, intermediate files, etc., in your submission.