Assignment 06: LC-3 Assembler

CSci 450: Computer Architecture

Objectives

In this assignment, you will implement the highest level of abstraction for the LC-3 architecture that we discuss in our class, the assembly language level. The assembly language level differs in significant respects from the micro-architecture level, ISA and OS levels in that we are building a translator here instead of an interpreter. Thus the purpose of the final LC-3 assembler is to take a plain ASCII input file of LC-3 assembly code, and translate it into an object or executable file format that is suitable to be interpreted by our previous assignment simulators.

You will find that in many respects, the code is larger here than what we needed to build the previous interpreters. This is partly because of the low-level nature of the plain C language we are using, we will need to build some basic mechanisms such as symbol tables and tokenizers so we can effectively complete the assembler. Also processing string in plain C is done by using arrays of simple characters, which are a lower level of abstraction of strings and handling them than you may be familiar with from using other higher-level languages. We recommend you read/review c array processing, such as C Tutorial: Strings in C a bit before beginning work on the assignment tasks.

Questions

- Why are two passes typically needed in this type of a translator?
- What is the purpose of pass 1 in a two-pass translator such as the LC-3 assembler?
- How are symbols extracted and a symbol table built from pass one of an assembler?
- How does a tokenizer operate in a simple translator to begin parsing a source file for translation?
- How are addresses resolved and calculated in a two-pass translator?
- What is the output/representation that a translator like the LC-3 assembler produces that can be loaded and interpreted by an LC-3 machine?

Objectives

- Understand some of the basics of a two-pass translator.
- Become familiar with symbol table creation and use in assemblers/compilers.
- See the basics of line-based tokenization for a line-oriented translator.
- Understand the purpose and format of binary output formats from such translators that are loaded and interpreted by the computer architecture hardware.

Description

In this assignment you will be completing some missing components of a basic translator for the LC-3 architecture. In some respects the translator from assembly to a suitable binary machine format is actually more complicated than the implementation/simulation of the machine translation we have already done in this class. Though a lot of this complexity is because of the low-level nature of the C language we continue to use for this assignment. So before we can implement the pass one and two assembly processes, we need to create some supporting data structure, such as tokenizers and symbol tables, to be used by the basic two-pass assembly process.

Though we are using plain C in this assignment, much of the code is based on an object-oriented design. Plain C does not supported object oriented programming directly. But you will notice that each `.[c|h]` source/header file defines a structure, and a set of methods that construct and destruct, and apply operations on the structure. For example, the `symbol-table.[c|h]` file includes the declaration of a `symbol_table` structure, and functions like `st_construct()` and `st_destruct()`. The `st_construct()` acts as a class constructor, its purpose is to dynamically allocate and initialize a `symbol_table` structure and return it. The other methods in the `symbol_table.[c|h]` class are mostly member methods, that expect a pointer to the `symbol_table` as the first explicit parameter that they will perform there work on. This basic object-oriented design pattern is repeated for most all of the other files/classes you will use and update in this assignment.

Overview and Setup

For this assignment you will be implementing missing member from many of the separate files/classes of this assignment, culminating in a fully working `lc3asm` assembler that can translate LC-3 assembly files into binary object files ready to be loaded and interpreted. As usual before starting the assignment tasks proper, you should make sure that you have completed the following setup steps:

1. Accept the assignment and copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment 05 LC-3 Assembler' for our current class semester and section.
2. Clone the repository using the SSH URL to your host file system in VSCode. Open up this folder in a Development Container to access and use the build system and development tools.
3. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor. Confirm that you C/C++ Intellisense extension is working, and that your code is being formatted according to class style standards when files are saved.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to `Issues`, and create them from the issue templates for the assignment. Also make sure you are linking each issue you create with the `Feedback` pull request for the assignment.

Assignment Tasks

Task 1: Symbol Table file/class tasks

We will start by completing some unwritten functions for the `symbol-table.[c|h]` class. The `symbol-table` provides a basic open hashing scheme implementation for use by pass one of the assembler when identifying and assigning addresses to symbols found in LC-3 assembly source files. As usual, start by `#define task` in the `assg06-tests.cpp` file. The code should still compile and run, though several of the tests should be failing.

Task 1 part 1: Complete the `st_hash()` function

We first need to define a hash function that can hash a symbol, represented as an array of `char` types and null terminated with `\0`, into an unsigned 32 bit integer value that can serve as an index into the `entries[]` hash. The function and function documentation has already been given to you, but currently the `st_hash()` function simply always returns 0.

The hash function needs to do the following:

1. Declare a local variable (for example named `key`) and initialize it to 0. This will hold the hash key while we calculate it to be returned. This should be of type `unsigned` just like the return type of the `st_hash()` function.
2. You need a loop that loops over each character of the `symbol` that is passed in as the second parameter. `symbol` is a pointer to a null terminated array of characters. So you should loop until the current character is `\0`.
   - First multiply the current `key` by 31. Initially when `key` is 0, multiplying by 31 will still be 0, but then when you add in the ASCII value of the first character in the `symbol` it will be non zero. After that, multiplying by 31 before adding in each character will increase the key.
   - After the multiply, add in the ASCII value of the current character into the key.
3. The resulting returned key needs to be in the range from 0 to `table_size - 1`. You are passed in the `symbol_table` as the first parameter in this function, named `st`. You can access the size of the symbol table you are hashing by accessing the `st->table_size` member, this holds the allocated size of the `entries` hash table array being used in this `symbol_table`. You need to use modulus `%` arithmetic to rescale the `key` into the correct range from 0 to `table_size - 1` before returning it.

This algorithm is a typical and basic hash function used to create a hash key from an ASCII string.

Once you implement the `st_hash()` function, all of the tests in task 1 that test the return hash key should now pass if you implement the hashing function correctly as described.

Task 1 part 2: Complete the `st_lookup()` function

As mentioned, the symbol table is not yet complete until you also implement the `st_lookup()` function, and the tests of `st_lookup()` for `task1` should not yet be passing until this function is also finished. The purpose of `st_lookup` is to return a pointer to a symbol table entry structure `st_entry` representing the mapping of the `symbol` to its assigned address. This function should return `NULL` if the asked for `symbol` is not currently in the table.

To implement this function, you have to understand a bit of the open hashing scheme being used by this class. This function is given the `symbol_table`, called `st` as the first parameter when it is called. A `symbol_table` is a `struct` that contains an array named `entries`. The `entries` are dynamically allocated based on the symbol table size, so you will see this member declared as an `st_entry**` in the `symbol_table`. This is because it is an array of `st_entry*` pointers. So for example, you can access the `entries` for index/key 0 by doing `st->entries[0]`. This will be a pointer to a linked list of `st_entry` instances. If the linked list is empty, then there will be a `NULL` in the table. Otherwise all symbols that hashed to that key/index will be maintained as a linked list of `st_entry` items.

So to perform a lookup you need to do the following:

1. Use your `st_hash()` function to determine the hash key/index for the given `symbol` that is to be looked up.
2. The `st->entries[key]` will be a pointer to a linked list of `st_entry` items. So you need to search each entry on this list to see if you can find the asked for symbol or not. Start with the entry at the front of the linked list, for example:

```
st_entry* entry = st->entries[key];
```

3. Create a loop that executes as long as the pointer to the entry is not NULL.
    - Check if the `entry->symbol` is the symbol that you are searching for. Use the `match()` function declared in the `assembler.[h|c]` file for this.
    - If they symbol matches the entry, just return the found `st_entry` pointer.
    - otherwise move to the next entry in the linked list `entry = entry->next`;
4. If the symbol is not found in the loop search, it means the search failed.
   So this function should return `NULL` as the result after the loop to indicate a failed lookup.

Once you have implemented `st_lookup()` correctly and as described, and have the `st_hash()` function working, all of the tests in `task1` should now be succeeding. When satisfied with your `task1` work, make a commit and push it to your GitHub classroom repository to be evaluated as usual before moving on to task 2.

Task 2: Tokenizer file/class tasks

The `tokenizer.[h|m]` file/class provides an object that will open up a file and tokenize the file for the LC-3 assembler in a line-oriented tokenization process. The main member method, besides the constructor and destructors is the `tk_next_line()` function, which scans the open file being tokenized for the next line with a potential set of tokens. This function returns a pointer to a `tokens` instance, which is hardcoded to extract up to 5 tokens from a line. This function skips over any lines that are empty or only contain comments, and skips over any comments on a line after the `;` comment character to the end of the line.

The `tk_next_line()` function uses the `strtokquote()` method to perform most of the work of finding tokens on an input line of text. This method is a modified version of the `strtok()` library function, that returns string literals between opening and closing '"' symbols as a single token. Otherwise whitespace and the `,` character are considered as delimiters between tokens everywhere else besides inside of a string literal. We need a `strtok` implementation that keeps string literals as a single token to make it easier to assemble the `.STRINGZ` pseudo opcode for LC-3.

There is no work to be done for this task. You should enable the `task2` tests and ensure that they all pass successfully at this point. And you should read the code to see if you can understand it, especially the `tk_next_line()` and the `strtokquote()` methods which provide the heart of the file tokenization for the assembler.

Task 3: Opcode file/class tasks

The `opcode.[h|c]` class contains an enumerated type of all of the LC-3 opcodes supported by our assembler, and the `opcode` structure. Every valid line of LC-3 assembly is required to have a single valid opcode or pseudoopcode. So each time a line is identified by the tokenizer as having tokens, we first need to check that the line is a valid line with an opcode token. This file/class provides the `extract_opcode()` method to translate information about the discovered opcode. The opcodes need to ultimately be translated to 16 bits in the binary file, so the enumerated type `opcode` value for the opcodes is defined to directly correspond to these needed 4 bits so that we can use them when outputting the final binary file. Pseudo opcodes do not require this as they can cause output to the binary file, but mainly in the form of reserving memory for local variables. In addition to the opcode binary value, we also need to extract other information, such as the `N`,`Z`,`P` flags for a `BR` instruction and variant information for `JSR`/`JSRR` and `JMP`/`RET` instructions. You will notice if you read the `extract_opcode()` implementation, that the `flags` and `variant` are determined here from the parsed tokens.

The fist thing that needs to be determined when translating a potential line is if an opcode keyword is present on the line. There is a function named `is_keyword()` that is used by `extract_opcode()` that needs to be implemented to get the task 3 tests working.

`is_keyword()` takes a C character array of a parsed token from a line in the input file. This function needs to return a boolean result of `true` if the `token` is an `opcode` keyword, and `false` if not. Immediately above the stub `is_keyword()` function at the bottom of `opcode.c` is an array of all of the 29 valid keywords/opcodes we wish to support with the LC-3 assembler.

The `is_keyword()` is currently a stub function that always returns `false`. You need to enable the `task3` tests and implement this function to get them to pass. You should search through all 29 of the `keywords` in the declared array to see if the given `token` input parameter is an opcode keyword or now. Use the `match()` function again here, defined in the `assembler.[h|c]` file for this. If you find that the token is a keyword, you should return `true`, otherwise if it doesn't match any valid keywords return a `false` result.

If you implement this function correctly, the `task3` tests should pass to translate the `opcode` tokens in the test file. Once you are satisfied with your work, make a commit of task 3 and push it to your GitHub classroom repository.

Task 4: Operand file/class tasks

The `operand.[h|c]` file/class is used to extract and translate operands for the LC-3 assembler. Each valid line of LC-3 assembly contains an optional label/symbol, one and only one opcode, and then 0 to 3 operands. The `RTI` instruction is the only LC-3 opcode that has 0 operands. Some opcodes can have up to 3 operands, for example `ADD R1, R3, #-1`, has 2 register operands and a numeric (decimal) literal value.

If you look in `operand.h` you will see that the `enum oprtype` defines that there are 4 valid types of operands that can be used for LC-3 assembly instructions: `REGISTER`, `NUMERIC`, `STRING` and `SYMBOL`. A lot of operands for LC-3 instructions are register operands, like `R1` and `R3` shown before. The valid set of registers in the LC-3 architecture are from `R0` to `R7`, and we use 3 bits in an assembled binary file for a register operand. Hexadecimal literals like `0xFF` or `xFF`, and decimal literals like `#-1` can be used in some operand locations for immediate operand values.

The `SYMBOL` enumerated type is used when an token is actually identified to be a symbol/label. For example in `BRp LOOP`, `LOOP` is actually a symbol that should be in the symbol table. A symbol usually represents a relative address that needs to be calculated as the target of a branch or some other operation.

And finally the `STRING` operand can only occur for the `.STRINGZ` pseudo opcode to declare a constant string for use in a program.

You will notice if you look further in `operand.h` that the `operand` structure will hold the decoded/translated value or svalue. The value is a numeric integer, and is used for `REGISTER`, `NUMERIC` and `SYMBOL` operands. The first two are self-explanatory, the value for a symbol operand is calculated in pass 2 when we need the offset amount for an instruction from the current PC to some defined label. The `svalue` is only used for a `STRING` operand, and contains a pointer to the C character array of the string literal that was found for the operand.

Besides the constructor and destructor, the main function for this file/class is `extract_operand()`. This function takes a token and attempts to extract and translate it as one of the valid operand types described. The actual work is done by the `is_*()` and `extract_*()` function.

As usual enable the `task4` tests. We have left the `is_*()` functions unfinished for you to implement. The `task4` tests test each of these individually, and then test the `extract_operand()` as a whole which indirectly call the methods you need to finish in this task.

Task 4 part 1: Implement `is_register()` operand method

All of these methods for task 4 will return a boolean result of `true` if the given `token` is detected to be of the given operand type. For `is_register` we define any token whose first character is `R` to be a (potential) register operand. So test the character at index 0 of the token and return `true` if it is an `R` and `false` if not. You don't need to do any error checking here to ensure that the register number is a valid one in the range 0-7, and in general you don't need to error check anything in these task 4 functions.

Task 4 part 2: Implement `is_string()` operand method

This method should be about the same as the previous one. A string literal is defined in our assembler as a token whose first character is a literal quote `"` character. If the token starts with a `"` return `true` from this function, and if not return `false`.

Task 4 part 3: Implement `is_hex_digit()` operand method

The C library function `isdigit()` returns `true` if a given character is one of `0-9`, and returns false if not. We need to use hexadecimal literal a lot in our LC-3 assembly code, so we want a function that returns true if a given character is `0-9` but also `A-F` or `a-f`. We support either lowercase or uppercase hex letters here in this function. Notice that this function takes a single character, not an array of characters like the other ones. This function should be reused in the next task function.

Task 4 part 4: Implement `is_hex_literal()` operand method

What we really need for the extract function is something that returns true if the `token` string is something like `0XFF` or `x3aaa`. You can maybe think of a more elegant solution, but one approach is to brute force the tests here. First if the first character is `X` or `x` and the second character is a hex digit (use your previous functin), then this is (potentially) a hex literal. This handles cases like `x3aaa` or `X1F`.

But also check for cases like `0x3aaa` and `0X1F` here. Again you could just brute force, and if the first character is `0`, the second is `X` or `x` and the third is a hex digit then you return `true`.

If neither of these is the case, return `false`, this is not a recognized potential hex literal operand token.

Task 4 part 5: Implement `is_decimal_literal()` operand method

We will also support decimal literal values for immediate operands in our LC-3 assembler. By convention usually `#` is used for a decimal literal in many assemblers, like `#-1` or `#255`.

For this method, if the first character is a `#` and the second character is a digit (use the C library `isdigit()` method here), then return `true`. We will first check if a token is a hex literal before checking if it is a decimal literal, so something like `#0987` will give a `false` result from your `is_hex_literal()` since there is no `X` in the token, but should return `true` from this function and be treated as a decimal literal value.

Once you have enabled the `task4` tests and implemented these 5 helper functions, if you implement all of them correctly, the tests should pass for this task. Each `is_*()` helper is tested individually (though not too extensively), so you should pass all of those first, and if any of those do not pass it should indicate which of these helper functions you are not quite getting correct yet.

Once your are satisfied with your implementation and are passing the `task4` tests, create a new commit and push it to your GitHub classroom repository for evaluation.

Task 5: Operation linked list class/file tasks

This is another task that has no work to do for this assignment. At this point you should enable the `task5` tests and ensure that they pass for the `operation-list.[h|c]` methods, and read through and understand the code and the purpose of this class/file.

The `operation-list.[h|c]` is used by the LC-3 assembler to create a linked list of the translated/processed information from the pass 1 of the assembler. The pass 2 simply iterates over this linked list structure of the partially processed operations, instead of reopening and retokenizing the file for a second pass.

The structure of each valid operation line for an LC-3 assembly file has the format:

```
LABEL opcode operand1, operand2, operand3 ; line comments
```

The only required component of a valid LC-3 operation line is the `opcode`. All lines that are blank or only have comments are skipped over by the tokenizer, and all tokens on a line after the first `;` encountered are likewise skipped.

So if you look in the `operation-list.h` file, you will see that there is an `operation_list` structure which maintains and builds a linked list of `opl_entry` items. Each `opl_entry` represent 1 tokenized/translated line of an LC-3 operation. During the first pass, as each line is read in and tokenized, we create 1 entry and append it to an `operation_list`. The `opl_entry` has the `opcode`, which is the only required field of an entry. But if present, the line `label` and up to 3 `operands` for the `opcode` can be present for an `opl_entry`. In addition, the address assigned to this line, and the size (in number of machine words) of this operation line are kept and calculated here. Since this is a simple linked list structure, each tokenized line encountered cause a new `opl_entry` to be created and appended to the end of the linked list during the first pass.

The methods of the `operation-list.[h|c]`, besides the constructor and destructor, are mainly there to append a new entry, with a detected `opcode` and optional `label` (the `opl_append()` method) and then to append individual `operand` items to the entry as they are processed (`opl_append_operand()`).

The remaining methods `opl_begin()`, `opl_next()` are convenience methods to create and easily iterate over the operation linked list.

Task 6: LC-3 Assembler class/file tasks

The `assembler.[h|c]` module uses the `tokenizer`, `symbol-table` and `operation-list` modules to perform the LC-3 translation process. The main functions that do the work in this class/file are the `pass_one()` and `pass_two()` method. Take a moment to read through and understand this code.

Notice in the `pass_one()` function that we use a `tokenizer` (named `tk`) and a `symbol_table` (named `st`) in this pass. An `operation_list` is constructed and appended too in this method, and the result of this pass 1 is saved in the operation list and returned from this function. The main loop of this function is similar to the pseudo code from chapter 7 of our textbook. Each line is tokenized, and the `opcode` and `label` (if present) are first extracted. Then an `entry` is created and appended to the operation list, and any `operand` we find on the line are processed and added to the `entry`. In addition this pass keeps track of the calculated `address` being assigned to each line during the pass. Any `.ORIG` pseudo opcodes encountered will initialize the current `address`. Then the `address` will be incremented depending on the size of the operation. Most LC-3 `opcode` need a single address in memory, so the `size` is 1 and the address usually increases by 1. But some pseudo opcodes, like `BLKW` and `STRINGZ` can be used to allocate and initialize more than 1 memory address in the resulting file.

In `pass_two()` the resulting `operation_list` linked list and `symbol_table` dictionary from pass one are passed in. The main purpose of pass two is to assemble the individual operation lines into the correct machine instruction that needs to be written to the resulting output binary file. Thus this method iterates over the `operation_list` linked list structure, calling `asm_inst()` for each entry to assemble the resulting machine instruction. All operands have already been translated in pass one except for `SYMBOL` operands. So before assembling instructions, we look at all of the `operand` items for this `entry` and calculate the correct relative offset for all symbols.
The result of `calculate_symbol_offset` ends up in the `opr->value` field.

You should also read through the `asm_inst()` and the individual function to assemble the individual opcodes into machine instructions. The `asm_inst()` is really just a big switch statement, the real work to assemble instruction and process pseudo opcodes is done in the individual `asm_*()` functions. For example, look at the `asm_add()` function that assembles an `ADD` operation line. We do minimal error checking in this assembler, if we encounter an unexpected situation we usually give a simple error message on standard error and exit immediately. Most assemblers and compilers try and be nicer to the user, and try to continue translating instead of just stopping. For our `asm_add()` there are two variants, but both expect to have exactly 3 operands. The first operand is always the destination register DR, and the second is always the first source register SR. But the third operand can either be another register, or an immediate value. Notice that we use bit-wise operations to shift and or together values to assemble the machine instruction.

There are two methods that are stub functions that you need to complete to get the `task6` tests to pass. Define the `task6` tests and implement the following to get all of the tests passing.

Task 6 part 1: implement `check_for_symbol()` method

The `check_for_symbol()` method is used in `pass_one()` to determine if the first token on a line is a symbol/label or not. Basically some lines can contain an optional label. So this method needs to check the first token of the list of `tokens`. If the first token on a line IS NOT a keyword (use your `is_keyword()` function to test this), then it must be an optional symbol/label. In that case you should return the first token, as this method expects a `char*` to be returned from it. But if the first token IS a keyword, then the optional label is missing on the line. In that case `NULL` needs to be returned indicating there is no symbol/label on the tokenized line.

Task 6 part 2: implement `calculate_symbol_offset()` method

The `calculate_symbol_offset()` method is used in `pass_two()` when a symbol is encountered while assembling an instruction that represents a relative offset from the PC address that needs to be calculated and assembled into the machine instruction.

This method is passed in the `operand` that contains the symbol that needs to have its offset calculated for, and the `opaddress` which is the current address of the operation line. In addition the `symbol_table` is passed in since we need to look up the address of the symbol label to perform the offset calculation.

Start by using `st_lookup()` to lookup the symbol in the symbol table. you need the `opr->svalue` which contains the name of the symbol of this operand we are trying to offset to. The `st_lookup()` will return a `st_entry*` object.

If the returned entry from the symbol table is `NULL` then there is a problem, there was no label encountered in pass one with the given name in the operand. In that case you should use `fprintf()` to display an appropriate error message, and then use `exit(1)` to just exit immediately (we can't test for this in the unit tests, but I will check it by hand when grading).

If the symbol is found in the table, then you can use the `entry->address` member to find out the address assigned to the symbol/label in pass 1. From there the offset is the difference of the label address and the operand address (`opaddress`) parameter that is passed in. You should add 1 to the `opaddress` when calculating this offset to account for the auto increment of the PC, the relative offset will be relative to the PC after the fetch auto increment. This offset can be positive or negative. The `calculate_symbol_offset()` does not return a result, instead this calculated offset needs to be set for the `opr->value` filed of the `operand` before the function ends.

Implement both functions and get them to pass the `task6` tests. Once you are satisfied with your work, make a commit and push it to your GitHub classroom repository for evaluation.

Task 7: Full System Tests

The final 10 points from the assignment come from successfully passing all of the system tests for this assignment. You again don't need to add anything, but it is possible that some system tests can and will fail even if you get all of your unit tests passing, since the system tests are test full end-to-end translation of several files.

To run the system tests, open up a terminal while running in your DevContainer and do:

```
vscode -> /workspaces/assg06-solution-v2 (main) $ make system-tests
./scripts/run-system-tests
System test test-allopc binary: PASSED
System test test-allopc output: PASSED
System test multiply-by-six binary: PASSED
System test multiply-by-six output: PASSED
System test cin binary: PASSED
System test cin output: PASSED
System test cout binary: PASSED
System test cout output: PASSED
System test halt binary: PASSED
System test halt output: PASSED
System test trap-vector binary: PASSED
System test trap-vector output: PASSED
===============================================================================
All system tests passed     (12 tests passed of 12 system tests)
```

The system tests cause LC-3 assembly files to be assembled with verbose output turned on, and the resulting binary file and the verbose output status are compared against the expected results. For example, you can run the `lc3asm` assembler that is built by hand on the `progs/halt.asm` file like this:

```
vscode -> /workspaces/assg06-solution-v2 (main) $ ./lc3asm -v -o halt.lc3 progs/halt.asm
verbose: 1
input  file <progs/halt.asm>
output file <halt.lc3>
Pass 1 Symbol Table Results
Symbol            ADDRESS (indx)
----------------------------------
SaveR0.............0x0530 (0270)
SaveR1.............0x0531 (0271)
ASCIINewline........0x052F (0277)
MCR.................0x0547 (0991)
Message............0x0532 (2928)
MASK................0x0548 (3850)


Pass 2 Assembly Results
LABEL              OPCODE    OPERANDS                                 ADDR: INST          BINARY
-----------------------------------------------------------------------------------------------------
                   .ORIG     0x0520                                   0520: 0000 0000000000000000
                   ST        R1, SaveR1                               0520: 3210 0011001000010000
                   ST        R0, SaveR0                               0521: 300E 0011000000001110
                   LD        R0, ASCIINewline                         0522: 200C 0010000000001100
                   TRAP      x21                                      0523: F021 1111000000100001
                   LEA       R0, Message                              0524: E00D 1110000000001101
                   TRAP      x22                                      0525: F022 1111000000100010
                   LD        R0, ASCIINewline                         0526: 2008 0010000000001000
                   TRAP      x21                                      0527: F021 1111000000100001
                   LDI       R1, MCR                                  0528: A21E 1010001000011110
                   LD        R0, MASK                                 0529: 201E 0010000000011110
                   AND       R0, R1, R0                               052A: 5040 0101000001000000
                   STI       R0, MCR                                  052B: B01B 1011000000011011
                   LD        R1, SaveR1                               052C: 2204 0010001000000100
                   LD        R0, SaveR0                               052D: 2002 0010000000000010
                   RTI                                                052E: 8000 1000000000000000
ASCIINewline       .FILL     x000A                                    052F: 000A 0000000000001010
SaveR0             .BLKW     1                                        0530: 0000 0000000000000000
SaveR1             .BLKW     1                                        0531: 0000 0000000000000000
Message            .STRINGZ  "Halting the machine."                   0532: 0048 0000000001001000
                   .         .                                        0533: 0061 0000000001100001
                   .         .                                        0534: 006C 0000000001101100
                   .         .                                        0535: 0074 0000000001110100
                   .         .                                        0536: 0069 0000000001101001
                   .         .                                        0537: 006E 0000000001101110
                   .         .                                        0538: 0067 0000000001100111
                   .         .                                        0539: 0020 0000000000100000
                   .         .                                        053A: 0074 0000000001110100
                   .         .                                        053B: 0068 0000000001101000
                   .         .                                        053C: 0065 0000000001100101
                   .         .                                        053D: 0020 0000000000100000
                   .         .                                        053E: 006D 0000000001101101
                   .         .                                        053F: 0061 0000000001100001
                   .         .                                        0540: 0063 0000000001100011
                   .         .                                        0541: 0068 0000000001101000
```

```
                    .         .                                                  0542: 0069 0000000001101001
                    .         .                                                  0543: 006E 0000000001101110
                    .         .                                                  0544: 0065 0000000001100101
                    .         .                                                  0545: 002E 0000000000101110
                    .         .                                                  0546: 0000 0000000000000000
MCR                 .FILL     xFFFE                                              0547: FFFE 1111111111111110
MASK                .FILL     x7FFF                                              0548: 7FFF 0111111111111111
                    .END                                                         0549: 0000 0000000000000000
                    .END                                                         0549: 0000 0000000000000000
Assembly complete
    bin file: <halt.lc3>
    words written: <0X002B>
    section: <0X0001> address: <0x0520> size: <0X0029>
```

The `-v` flag causes verbose output from the assembly passes. The symbol table from pass one is first shown at the end of that pass, then the pass 2 operation list along with the assembled machine instructions and assigned addresses are shown on standard output. The `-o` flag specifies the name of the binary output file to save the assembled machine instructions into, in this case to a file called `tmp.lc3`

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 50 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 5 to 10 points are awarded for completing each of the remaining 6tasks. However you should note that the autograder awards either all point for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Make sure all system tests are passing, and debug any issues if you have trouble getting one or more of them to pass. You should examine the output from running the assembler on the test file, and look in more detail at the `assembler.[h|c]` pass one and pass two functions, as well as the function that writes the final binary object file.

Requirements and Grading Rubrics

Program Execution, Output and Functional Requirements

1. Your program must compile, run and produce some sort of output to be graded. 0 if not satisfied.
2. 40 points for keeping code that compiles and runs. A minimum of 50 points will be given if at least the first task is completed and passing tests.
3. 5 to 10 points are awarded for completing each subsequent task 2-10.
4. +5 bonus pts if all system tests pass and your process simulator produces correct output for the given system tests.

Program Style and Documentation

This section is supplemental for the first assignment. If you uses the VS Code editor as described for this class, part of the configuration is to automatically run the `clang-format` code style checker/formatter on your code files every time you save the file. You can run this tool manually from the command line as follows:

```
$ make format
clang-format -i include/*.hpp src/*.cpp
```

Class style guidelines have been defined for this class. The `uncrustify.cfg` file defines a particular code style, like indentation, where to place opening and closing braces, whitespace around operators, etc. By running the beautifier on your files it reformats your code to conform to the defined class style guidelines. The beautifier may not be able to fix all style issues, so I might give comments to you about style issues to fix after looking at your code. But you should pay attention to the formatting of the code style defined by this configuration file.

Another required element for class style is that code must be properly documented. Most importantly, all functions and class member functions must have function documentation proceeding the function. These have been given to you for the first assignment, but you may need to provide these for future assignment. For example, the code documentation block for the first function you write for this assignment looks like this:

```
/**
 * @brief initialize memory
 *
 * Initialize the contents of memory.  Allocate array larget enough to
 * hold memory contents for the program.  Record base and bounds
 * address for memory address translation.  This memory function
 * dynamically allocates enough memory to hold the addresses for the
 * indicated begin and end memory ranges.
 *
 * @param memoryBaseAddress The int value for the base or beginning
 *   address of the simulated memory address space for this
 *   simulation.
 * @param memoryBoundsAddress The int value for the bounding address,
 *   e.g. the maximum or upper valid address of the simulated memory
 *   address space for this simulation.
 *
 * @exception Throws SimulatorException if
 *   address space is invalid.  Currently we support only 4 digit
 *   opcodes XYYY, where the 3 digit YYY specifies a reference
 *   address.  Thus we can only address memory from 000 - 999
 *   given the limits of the expected opcode format.
 */
```

This is an example of a `doxygen` formatted code documentation comment. The two `**` starting the block comment are required for `doxygen` to recognize this as a documentation comment. The `@brief`, `@param`, `@exception` etc. tags are used by `doxygen` to build reference documentation from your code. You can build the documentation using the `make docs` build target, though it does require you to have `doxygen` tools installed on your system to work.

```
$ make refdocs
Generating doxygen documentation...
doxygen config/Doxyfile 2>&1 | grep -A 1 warning | egrep -v "assg.*\.md" | grep -v "Found unknown command"
Doxygen version used: 1.9.1
```

The result of this is two new subdirectories in your current directory named `html` and `latex`. You can use a regular browser to browse the html based documentation in the `html` directory. You will need `latex` tools installed to build the `pdf` reference manual in the `latex` directory.

You can use the `make refdocs` to see if you are missing any required function documentation or tags in your documentation. For example, if you remove one of the `@param` tags from the above function documentation, and run the docs, you would see

```
$ make refdocs
doxygen config/Doxyfile 2>&1 | grep -A 1 warning | egrep -v "assg.*\.md" | grep -v "Found unknown command"

HypotheticalMachineSimulator.hpp:88: warning: The following parameter of
```

```
HypotheticalMachineSimulator::initializeMemory(int memoryBaseAddress,
    int memoryBoundsAddress) is not documented:
  parameter 'memoryBoundsAddress'
```

The documentation generator expects that there is a description, and that all input parameters and return values are documented for all functions, among other things. You can run the documentation generation to see if you are missing any required documentation in you project files.