Assignment 1 : Hypothetical Machine Simulator

CSci 430: Introduction to Operating Systems

Objectives

In this assignment you will be building an implementation of the hypothetical machine simulator like the one discussed in chapter 1 of our textbook and that you worked on for the first written assignment. The goal is to become better familiar with some fundamental hardware concepts that we rely on when building operating system components in this class. Another goal is to familiarize you with the structure of the assignments you need to complete for this class.

Questions

- What is the purpose of a standard fetch-execute cycle in a computing system?
- How does a computing system operate at the hardware level to translate and execute instructions?
- How can test driven development help you to create and debug your code?

Objectives

- Familiarize ourselves with test driven development and developing software to pass unit tests.
- Become familiar with the class assignment structure of unit tests and system tests.
- Refresh our understanding of basics of how computing systems operate at a hardware level, by studying in more detail the Hypothetical Machine from our Stallings textbook, and implementing a working simulation of this hypothetical computing system.

Description

In this assignment you will be implementing a simulation of the hypothetical machine architecture description given in our Stalling textbook chapter 01. The hypothetical machine described is simple, and is meant to illustrate the basics of a CPU hardware fetch/execute cycle for performing computation, and a basic machine instruction set with some examples processor-memory, data processing, and control type instructions. We will simplify the hypothetical machine architecture in some regards, but expand on it a bit in others for this assignment. You will be implementing the following list of opcodes for this simulation:

opcode	mnemonic	description
0	NOOP / HALT	Indicates system halt state
1	LOAD	Load AC from memory
2	STORE	Store AC to memory
3	JMP	Perform unconditional jump to address
4	SUB	Subtract memory reference from AC
5	ADD	Add memory reference to AC

I have given you a large portion of the simulation structure for this first assignment, as the primary goal of the assignment is to become familiar with using system development tools, like make and the compiler and the unit test frameworks. For all assignments for this class, I will always give you a `Makefile` and a set of starting template files. The files given should build and run successfully, though they will be incomplete, and will not pass all (or any) of the defined unit and system tests you will be given. Your task for the assignments will always be to add code so that you can pass the unit and system tests to create a final working system, using the defined development system and Unix build tools.

All assignments will have 2 targets and files that define executables that are built by the build system. For assg01 the files are named:

- assg01-tests.cpp
- assg01-sim.cpp

If you examine the Makefile for this and all future assignment, you will always have the following targets defined:

- all: builds all executables, including the test executable to perform unit tests and the sim executable to perform the system test / simulations.
- unit-tests: Will invoke the test executable to perform all unit tests, and the sim executable to perform system tests. Notice that this target depends on unit-tests and system-tests, which in turn depend on the sim and test executables being first up to date and built.
- system-tests: Invoke full system tests of the final simulation.
- clean: delete all build products and revert to a clean project build state.

You should start by checking that your development system builds cleanly and that you can run the tests. You will be using the following steps often while working on the assignments to make a clean build and check your tests (you can run the make and make unit-tests target from VS Code as well):

```
$ make clean
rm -rf ./test ./sim *.o *.gch
rm -rf output html latex
rm -rf obj


$ make
mkdir -p obj
g++ -Wall -Werror -pedantic -O2 -g -Iinclude -c
   src/assg01-tests.cpp -o obj/assg01-tests.o
g++ -Wall -Werror -pedantic -O2 -g -Iinclude -c
   src/HypotheticalMachineSimulator.cpp -o obj/HypotheticalMachineSimulator.o
g++ -Wall -Werror -pedantic -O2 -g -Iinclude -c
   src/catch2-main.cpp -o obj/catch2-main.o
g++ -Wall -Werror -pedantic -O2 -g -Iinclude -c
   src/SimulatorException.cpp -o obj/SimulatorException.o
g++ -Wall -Werror -pedantic -O2 -g  obj/assg01-tests.o
   obj/HypotheticalMachineSimulator.o obj/catch2-main.o
   obj/SimulatorException.o -o test
g++ -Wall -Werror -pedantic -O2 -g -Iinclude -c
   src/assg01-sim.cpp -o obj/assg01-sim.o
g++ -Wall -Werror -pedantic -O2 -g  obj/assg01-sim.o
   obj/HypotheticalMachineSimulator.o obj/SimulatorException.o -o sim


$ make unit-tests
././test --use-colour yes

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
test is a Catch v2.13.9 host application.
Run with -? for options

-------------------------------------------------------------------------------
<translateAddress()> HypotheticalMachineController test memory address
translation
-------------------------------------------------------------------------------
src/assg01-tests.cpp:57
...............................................................................

src/assg01-tests.cpp:67: FAILED:
  CHECK( sim.translateAddress(476) == 176 )
with expansion:
```

```
  0 == 176


... skipped output of teests ...

===============================================================================
test cases:  11 |  1 passed |  10 failed
assertions: 171 | 58 passed | 113 failed
```

I skipped the output from running the unit tests. As you can see at the end all of the test cases, and most of the unit test assertions are failing initially. But if you look before that, the code is successfully compiling, and the test and sim executable targets are being built.

You will not have to modify the assg01-tests.cpp nor the `assg01-sim.cpp` files that I give you for this assignments. The `assg01-tests.cpp` contains unit tests for the assignment. The `assg01-sim.cpp` file will build a command line executable to perform system tests using your simulator. You should always start by writing code to pass the unit tests, and only after you have the unit tests working should you move on and try and get the whole system simulation working.

Overview and Setup

For this assignment you will be implementing missing member methods of the `HypotheticalMachineSimulator.[hpp|cpp]` file. 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 01 Hypothetical Machine Simulator' for our current class semester and section.
2. Clone the repository using the SSH URL to your local class DevBox development environment.
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++ 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 it/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

You should take a look at the test cases and assertions defined in the `assg01-tests.cpp` file to get started. I will try and always give you the unit tests in the order that it would be best to work on. Thus you should always start by looking at the first unit test in the first test case, and writing the code to get this test to pass. Then proceed to work on the next unit test and so on.

I have given you files named `HypotheticalMachineSimulator.hpp` and `HypotheticalMachineSimulator.cpp` for this first assignment. The `.hpp` file is a header file, it contains the declaration of the `HypotheticalMachineSimulator` class, as well as some supporting classes. You will not need to make any changes to this header file for this assignment. The `.cpp` file is where the implementations of the simulation class member functions will be created. All of your work for this assignment will be done in the `HypotheticalMachineSimulator.cpp` file, where you will finish the code to implement several member functions of the simulator class.

For this assignment, to get all of the functions of the simulator working, you need to perform the following tasks in this order. I give an outline of what should be done here to write each member function of the simulator. There are additional hints in the template files given as comments that you should look at as well for additional tasks you will need to perform that are not described here.

Task 1: Implement the `initializeMemory()` member function

Implement the `initializeMemory()` function. You can pass these unit tests by simply initializing the member variables with the parameters given to this function. However, you also need to dynamically allocate an array of

integers in this function that will serve as the memory storage for the simulation. You should also initialize the allocated memory so that all locations initially contain a value of 0. If you are a bit rusty on dynamic memory allocation, basically you need to do the following. There is already a member variable named `memory` in this class. Memory is a type `int*` (a pointer to an integer) defined for our `HypotheticalMachineSimulator` class. If you know how much memory you need to allocate, you can simply use the `new` keyword to allocate a block / array of memory, doing something like the following

```
memory = new int[memorySize];
```

There are some additional tasks as well for this first function. You should check that the memory to be initialized makes sense in terms of it size for this simulation.

Once you have completed this task and are satisfied with your solution, commit your changes to the `Feedback` pull request of your GitHub classroom repository.

Task 2: Implement the `translateAddress()` member function

Implement the `translateAddress()` function and get the unit tests to work for this test case. The `translateAddress()` function takes a virtual address in the simulation memory address space and translates it to a real address. So for example, if the address space defined for the simulation has a base address of 300 and a bounding (last) address of 1000, then if you ask to translate address 355, this should be translated to the real address 55. The address / index of 55 can then be used to index into the `memory[]` array to read or write values to the simulated memory. There is one additional thing that should be done in this function. If the requested address is beyond the bounds of our simulation address space, you should throw an exception. For example, if the base address of memory is 300, and the bounds address is 1000, then any address of 299 or lower should be rejected and an exception thrown. Also for our simulation, any address exactly equal to the upper bound of 1000 or bigger is an illegal reference, and should also generate an exception.

This function takes an integer virtual address as input, and it returns an integer result, the real address/index into the real memory for the simulation. This function should be defined as a `const` member function. It does not change the state of memory, registers or any other member variables of the hypothetical machine class. Such functions that only return information or provide a utility for other functions, but do not change the state of the class, are declared `const` so that the compiler knows that you are guaranteeing that the object will not change state when this function is called.

Once you have completed this task and are satisfied with your solution, commit your changes to the `Feedback` pull request of your GitHub classroom repository.

Task 3: Implement the `peekAddress()` and `pokeAddress()` simulation member methods

There are two defines for task 3, only define `task3_1` initially when first working on the `peekAddress()` and `pokeAddress()` functions.

Implement the `peekAddress()` and `pokeAddress()` functions and pass the unit tests for those functions. These functions are tested by using poke to write a value somewhere in memory, then we peek the same address and see if we get the value we wrote to read back out again. Both of these functions should reuse the `translateAddress()` function form the previous step. In both cases, you first start by translating the given address to a real address. Then for poke you need to save the indicated value into the correct location of your `memory[]` array. And likewise for peek, you need to read out a value from your `memory[]` array and return it.

The `peekAddress()` function again only returns information when it is called, it should not change the state of the class. So this function also needs to be a `const` member function.

Once your memory peeks and pokes are working, you can try out the tests of the `loadProgram()` function. This function needs to use your `initializeMemory()` from task 1, and the `pokeAddress()` function you just created. Define the `task3_2` tests in the test file. But also, the calls to `initializeMemory()` and `pokeAddress()` have been commented out of the `loadProgram()` to get the code to compile. You should uncomment the calls to these function in the `loadProgram()` member method, then see if the Task 3.2 tests pass when loading full programs for the simulation.

Once you have completed this task and are satisfied with your solution, commit your changes to the `Feedback` pull request of your GitHub classroom repository.

Task 4: Implement the `fetch()` simulation member method

Implement the `fetch()` method for the fetch phase of a fetch/execute cycle. If you are following along in the unit test file, you will see there are unit tests before the `fetch()` unit tests to test the `loadProgram()` function. You have already been given all of `loadProgram()`, but you should read over this function and see if you understand how it works. Your implementation of fetch should be a simple single line of code if you reuse your `peekAddress()` function. Basically, given the current value of the PC, you want to use `peekAddress()` to read the value pointed to by your PC and store this into the IR instruction register.

Once you have completed this task and are satisfied with your solution, commit your changes to the `Feedback` pull request of your GitHub classroom repository.

Task 5: Implement the `execute()` simulation member method

Implement the `execute()` method for the execute phase of a fetch/execute cycle. The execute phase has a lot more it needs to do than the fetch. You need to do the following tasks in the execute phase:

- Test that the value in the instruction register is valid
- Translate the opcode and address from the current value in the instruction register.
- Increment the PC by 1 in preparation for the next fetch phase.
- Finally actually execute the indicated instruction. You will do this by calling one of the functions `executeLoad()`, `executeStore()`, `executeJump()`, `executeSub()` or `executeAdd()`

To translate the opcode and address you need to perform integer division and use the modulus operator `%`. Basically the instruction register should have a 4 digit decimal value such as 1940 in the format `XYYY`. The first decimal digit, the 1000's digit, is the opcode or instruction, a 1 in this case for a LOAD instruction. The last 3 decimal digits represent a reference address, memory address 940 in this case. The translation phase should end up with a 1 opcode in the irOpcode member variable, and 940 in the irAddress member variable. You should use something like a switch statement as the final part of your `execute()` function to simply call one of the 5 member functions that will handle performing the actual instruction execution.

Once you have completed this task and are satisfied with your solution, commit your changes to the `Feedback` pull request of your GitHub classroom repository.

Task 6: Implement the `execute*()` member methods

Implement the `executeLoad()`, `executeStore()`, `executeJump()`, `executeSub()` and `executeAdd()` functions. Each of these has individual unit tests for them, so you should implement each one individually. All of these should be relatively simple 1 or 2 lines of code function if you reuse some of the previously implemented function. For example for the `executeLoad()` function, you should simply be able to use `peekAddress()` to get the value referenced by the irAddress member variable, then store this value into the accumulator.

Once you have completed this task and are satisfied with your solution, commit your changes to the `Feedback` pull request of your GitHub classroom repository.

Task 7: Enable the full Hypothetical Machine Simulation

Finally put it all together and test a full simulation using the `runSimulation()` method. The final unit tests load programs and call the `runSimulation()` method to see if they halt when expected and end up with the expected final calculations in memory and in the AC. You have been given the code for the `runSimulation()` method. However this method calls your `fetch()` and `execute()` member function implementations, so the code is commented out. Uncomment the implementation of `runSimulation()` and enable the Task 7 tests.

Likewise, there is a small bit of code in the `operator<<()` function near the bottom of the source file that has been commented out. This loop uses the `peekAddress()` member function that you implemented in task 3. You need to uncomment this loop so that full simulations and system tests will work. If you get compilation errors here, you may have failed to declare `translateAddress()` and `peekAddress()` as `const` member functions. These functions do not change the state of a simulation, so they should be safe to be used if a constant reference is passed somewhere.

If your implementations of the previous member functions are correct, you should be able to successfully run full simulations now and pass the Task 7 tests.

Once you have completed this task and are satisfied with your solution, commit your changes to the `Feedback` pull request of your GitHub classroom repository.

System Tests: Putting it all Together

Once all of the unit tests are passing, you can begin working on the system tests. For this first assignment, there is one thing that needs to be done to get your full system tests to pass. There is an `operator<<()` that is called to output the current state of the Hypothetical Machine. At the bottom of this function, a for loop has been commented out, that is making calls to your `peekAddress()` function. This loop is being used to display the contents of memory. Uncomment this loop and then try running full simulations and the system tests.

The sim executable that is built uses the `HypothetheticalMachineSimulation` class you finished implementing to load and execute a program in the simulated machine. The sim targets for the assignments for this class will be typical command line programs that will expect 1 or more command line parameters to run. In this first assignment the sim program needs 2 command line arguments: the maximum number of cycles to simulate and the name of a hypothetical machine simulation file to load and attempt to run. You can ask the sim executable for help from the command line to see what command line parameters it is expecting:

```
$ ./sim -h
Usage: sim maxCycles prog.sim
Run hypothetical machine simulation on the given system state/simulation file

maxCycles          The maximum number of machine cycles (fetch/execute
                     cycles) to perform
file.sim           A simulation definition file containing starting
                     state of machine and program / memory contents.
```

If the sim target has been built successfully, you can run a system test simulation manually by invoking the sim program with the correct arguments:

```
$ ./sim 100 simfiles/prog-01.sim
```

This will load and try and simulate the program from the file `simfiles/prog-01.sim`. The first parameter specifies the maximum number of simulated machine cycles to perform, so if the program is an infinite loop it will stop in this case after performing 100 cycles.

If you are passing all of the unit tests, your simulation should be able to hopefully pass all of the system tests. You can run all of the system tests using the system-tests target from the command line

```
$ make system-tests
./scripts/run-system-tests
System test prog-01: PASSED
System test prog-02: PASSED
System test prog-03: PASSED
System test prog-04: PASSED
System test prog-05: PASSED
System test prog-06: PASSED
System test prog-07: PASSED
System test prog-08: PASSED
System test prog-09: PASSED
System test prog-10: PASSED
================================================================================
All system tests passed     (10 tests passed of 10 system tests)
```

The system tests work by running the simulation on a program and comparing the actual output seen with the correct expected output. Any difference in output will cause the system test to fail for that given input program test.

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 6 tasks. 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.

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. 50 pts for submitting code that compiles and runs tests, and the first task is complete or mostly complete.
3. 5-10 pts for completing each of the remaining tasks 2-7. Each task must be committed to GitHub in a separate commit for credit.

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 `.clang-format` 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 large 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 refdocs` 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.