Assignment 1 : Hypothetical Machine Simulator

CSci 430: Introduction to Operating Systems

Fall 2020

Overview

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.

Introduction

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.
- 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.
- 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 tests target from VS Code as well):

```
$ make clean
rm -f  test sim *.o *.gch *~
rm -f  -r output html latex


$ make
g++ -Wall -Werror -pedantic -g -I../../include -c assg01-tests.cpp -o assg01-tests.o
g++ -Wall -Werror -pedantic -g -I../../include -c HypotheticalMachineSimulator.cpp -o HypotheticalMachineSi
g++ -Wall -Werror -pedantic -g assg01-tests.o HypotheticalMachineSimulator.o -L../../libs -lSimulatorExcept
g++ -Wall -Werror -pedantic -g -I../../include -c assg01-sim.cpp -o assg01-sim.o
g++ -Wall -Werror -pedantic -g assg01-sim.o HypotheticalMachineSimulator.o -L../../libs -lSimulatorExceptio


$ make tests
./test -s

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
test is a Catch v2.7.2 host application.
Run with -? for options

-------------------------------------------------------------------------------
<initializeMemory()> HypotheticalMachineController test memory initialization
-------------------------------------------------------------------------------
```

```
assg01-tests.cpp:29
...............................................................................

assg01-tests.cpp:34: FAILED:
  CHECK( sim.getMemoryBaseAddress() == 300 )
with expansion:
  0 == 300 (0x12c)


... skipped output of teests ...

===============================================================================
test cases:  11 |  0 passed |  11 failed
assertions: 170 | 35 passed | 135 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.

Unit Test 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.

1. 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.

2. 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.

3. 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.

4. 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()` funciton. 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.

5. 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.

6. 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.

7. 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. Your `runSimulation()` For this assignment you have been given the code for the `runSimulation()` method, but the code is commented out because it relies on you correctly implementing the above functions first to work correctly. Uncomment the code in the `runSimulation()` method and the final unit tests should now be passing for you.

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 you do not have to do anything to get the simulation working, it has been implemented for you. But in future assignments you may be asked to implement part of the full simulation as well. So you should try out the simulator and understand how it works.

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
./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

In order to document your work and have a definitive version you would like to grade, a MyLeoOnline submission folder has been created named Assignment 01 for this assignment. There is a target in your `Makefile for` these assignments named **submit**. When your code is at a point that you think it is ready to submit, run the submit target:

```
$ make submit
$ make submit
tar cvfz assg01.tar.gz HypotheticalMachineSimulator.hpp HypotheticalMachineSimulator.cpp
HypotheticalMachineSimulator.hpp
HypotheticalMachineSimulator.cpp
```

The result of this target is a tared and gziped (compressed) archive, named assg01.tar.gz in your directory. You should upload this file archive to the submission folder to complete this assignment.

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. 10 pts each (70 pts) for completing each of the 7 listed steps in this assignment to write the functions needed to create the Hypothetical Machine.
3. 20 pts if all given unit tests are passed by your code.
4. 10 pts if all system tests pass and your hypothetical machine 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 `uncrustify` code beautifier on your code files everytime you save the file. You can run this tool manually from the command line as follows:

```
$ make beautify
uncrustify -c ../../config/.uncrustify.cfg --replace --no-backup *.hpp *.cpp
Parsing: HypotheticalMachineSimulator.hpp as language CPP
Parsing: HypotheticalMachineSimulator.cpp as language CPP
Parsing: assg01-sim.cpp as language CPP
Parsing: assg01-tests.cpp as language 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 docs
doxygen ../../config/Doxyfile 2>&1
  | grep warning
  | grep -v "\file statement"
  | grep -v "\pagebreak"
  | sort -t: -k2 -n
  | sed -e "s|/home/dash/repos/csci430-os-sims/assg/assg01/||g"
```

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 docs` 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 docs
doxygen ../../config/Doxyfile 2>&1
  | grep warning
  | grep -v "\file statement"
  | grep -v "\pagebreak"
  | sort -t: -k2 -n
  | sed -e "s|/home/dash/repos/csci430-os-sims/assg/assg01/||g"

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.