Assignment 2 Description

Assignment 2 - Writing a CPU Emulator

Weighting and Due Dates

- Marks for this assignment contribute 5% of the overall course mark.
- Marks for functionality will be awarded automatically by the web submission system.
- **Due dates: Milestone - 11:55pm Tuesday of week 9**, **Final - 11:55pm Friday of week 9**.
- **Late penalties:** For each part, the maximum mark awarded will be reduced by 25% per day / part day late. If your mark is greater than the maximum, it will be reduced to the maximum.
- **Core Body of Knowledge (CBOK) Areas:** abstraction, design, hardware and software, data and information, and programming.

Project Description

In this assignment you will implement an **emulator** for the Hack CPU in C++. It will have an array of 16 bit integers to represent ROM, an array of 16 bit integers to represent RAM and three 16-bit integer variables to represent the A, D and PC registers. You are required to write an ***emulate_instruction()*** function that reads the PC register, reads the corresponding instruction from ROM and then simulates the instruction's execution. You are also required to write a ***disassemble_instruction()*** function that may be used to generate part of the **emulator** program's output.

SVN Repository

Note: this assignment assumes that you have already created directories for every assignment, workshop, project and exam in your svn repository, as described on the **Startup Files for Workshops and Assignments (https://myuni.adelaide.edu.au/courses/64329/pages/startup-files-for-workshops-and-assignments)** page.

1. If required, checkout a working copy of the assignment2 directory from your svn repository.
2. Change directory to the working copy of the assignment2 directory.
3. Copy the latest startup files from the "**View Feedback**" tab of the "**Assignment 2 - Submit Here**" assignment into the updates sub-directory. **Do not unzip the file.**
4. Run the following command to place the assignment's startup files in the correct locations, this will automatically add them to svn and commit them to your svn repository:

```
% make install
```

5. Goto the Web Submission System and make a submission to the "**Assignment 2 - Submit Here**" assignment.

Assignment 2 Files and Directories

In addition to the generic **Makefile** and **updates** sub-directory, the assignment2 directory should now contain the following files and directories:

- **emulator.cpp** - C++ source file.
- **emulator** - executable script that will run your compiled **emulator** program.
- **bin** - this directory contains precompiled programs and scripts.
- **includes** - this directory contains **.h** files for the library.
- **lib** - this directory contains precompiled library components.
- **originals** - this directory contains the original version of **emulator.cpp**.
- **tests** - this directory contains test data.

Note: you need to edit the **emulator.cpp** file to complete this assignment. All the other files are automatically regenerated every time you run **make**, they must not be changed or added to **svn**.

Note: if a newer version of the startup files is made available, it must be placed in the **updates** sub-directory and added to **svn**. The next time **make** is run, all of the files will be updated except for **emulator.cpp**.

Submission and Marking Scheme

Submissions for this assignment must be made to the **web submission system** (https://cs.adelaide.edu.au/services/websubmission) assignment named: **Assignment 2 - Submit Here**. The assessment is based on "**Assessment of Programming Assignments** (https://myuni.adelaide.edu.au/courses/64329/pages/assessment-of-programming-assignments) ".

Notes:

- the marks for the Milestone Tests will be shown by the **Assignment 2 - Milestone** assignment
- the marks for the Final Tests will be shown by the **Assignment 2 - Final** assignment
- the participation marks for this assignment will be shown by the **Assignment 2 - Pre Milestone** assignment

Your programs must be written in C++. Your programs will be compiled using the **Makefile** included in the startup files. The **emulator** program will be compiled using the file **emulator.cpp**. No other **.cpp** or **.h** files should be present in your **svn** directory. Your programs will then be tested using a small set of test files also included in the startup files. In addition a number of ***secret*** tests will also be run. **Note**: if your program fails any of these ***secret*** tests you **will not** receive any feedback about these ***secret*** tests, even if you ask!

Assignment 2 - Milestone Submissions: due 11:55pm Tuesday of week 9

A mark out of 100 for the Milestone Tests will be awarded by the **web submission system** (https://cs.adelaide.edu.au/services/websubmission) and will contribute up to 20% of your marks for this assignment. The marks awarded will be automatically modified by a code review program, as described on the **Assessment of Programming Assignments** (https://myuni.adelaide.edu.au/courses/64329/pages/assessment-of-programming-assignments) page. The Milestone Tests are just those where the instructions to be emulated and disassembled are A-instructions.

Assignment 2 - Final Submissions: due 11:55pm Friday of week 9

A mark out of 100 for the Final Tests will be awarded by the **web submission system (https://cs.adelaide.edu.au/services/websubmission)** and will contribute up to 80% of your marks for this assignment. The marks awarded will be automatically modified by a code review program, as described on the **Assessment of Programming Assignments (https://myuni.adelaide.edu.au/courses/64329/pages/assessment-of-programming-assignments)** page. The Final Tests include tests that emulate A-instructions and tests that emulate C-instructions. Tests involving the ***disassemble_instruction()*** function will contribute 25% of the marks for the Final Tests.

Your Final Assignment Mark will be the geometric mean of two components, the weighted marks, **MF**, for the Milestone Tests, **M**, and Final Tests, **F**, and a mark for your logbook, **L**. It will be limited to 10% more than the marks for the Final Tests. ie

MF = M * 0.2 + F * 0.8

FinalMark = MIN(1.1 * MF, SQRT(MF * L))

NOTE - A logbook mark of 0 results in a Final Submission mark of 0.

Logbook Marking

Important: the logbook must have entries for all work in this assignment, including your milestone submissions. See "**Assessment - Logbook Review (https://myuni.adelaide.edu.au/courses/64329/pages/assessment-logbook-review)**" for details of how your logbook will be assessed.

Assignment Weighting and Workload

Depending on your individual programming skills and your understanding of the course material, the assignment weighting may not reflect the workload required to complete it. It is important to recognise this and to make sure that

you do not spend an excessive amount of time trying to complete everything at the expense of your other studies. It is important to know when to stop!

Assignment 2 - Participation Marks

Any submissions to this assignment that are made before the due date for Milestone Submissions may be awarded up to 5 participation marks. The participation marks will be the marks awarded for the Final Tests divided by 20. Your logbook review mark will not affect these participation marks. The participation marks will be allocated to week 9.

A CPU Emulator

Background

In this assignment you will implement an **emulator** for the Hack CPU in C++. It will have an array of 16 bit integers to represent ROM, an array of 16 bit integers to represent RAM and three 16-bit integer variables to represent the A, D and PC registers. You are required to edit the **emulator.cpp** file and complete the ***emulate_instruction()*** function that reads the PC register, reads the corresponding instruction from ROM and then simulates the instruction's execution. You are also required to complete the ***disassemble_instruction()*** function that will is used to generate part of the **emulator** program's output.

Compiling and Running Your Program

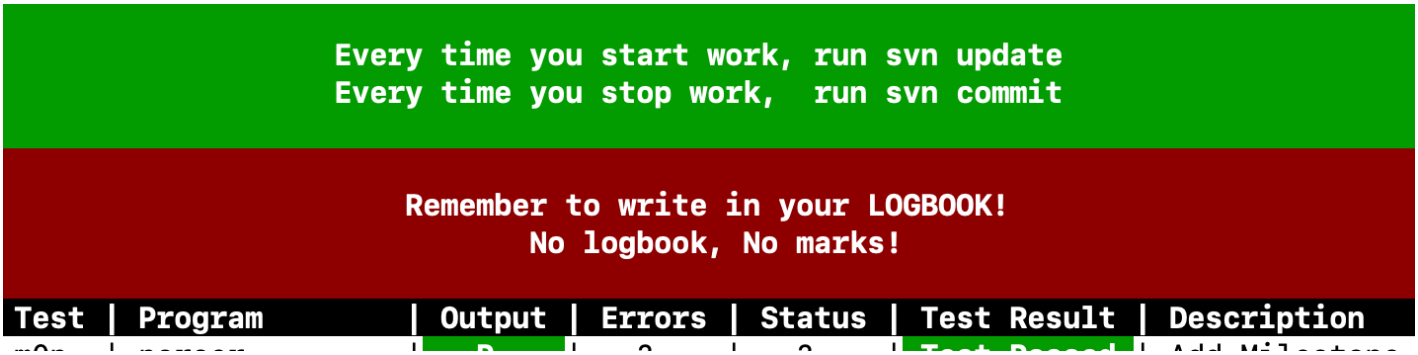
You program can be compiled with the following command.

```
% make notest
```

To compile and test your program at the same time you can use the following command:

% make

The test output should look a bit like this:



Notes:

- The reminder to run svn update and svn commit should appear every hour. It is important to always run svn update when you start work and svn commit when you make logbook entries or take a break.
- The reminder to write in your logbook should appear every half hour. If you logbook does not record your progress as you work your final mark for an assignment may be capped at 25 or in some cases reduced to 0.
- The Test column can be used to refer to a specific test by its **test-id** if you wish to inspect the actual output from the test.
- The Program column tells you what program was running including any command line arguments it was passed.
- The Output column shows you whether or not your program produced the expected standard output (file 1). If it did the column will have a green background, if it did not it will have a red background and if the test does not care, a **?** will be displayed.
- The Errors column shows you whether or not your program produced the expected standard error (file 2).
- The tests are all run twice, once with *write_to_traces()* and *write_to_logs()* enabled and once with them disabled. The Output and Error columns may contain a magenta * to indicate that this resulted in different output. The tests used for marking disable *write_to_traces()* and *write_to_logs()* but you may find them helpful for debugging
- The Status columns shows you the exit status returned when your program terminated.
- The Test Result column shows you whether or not your program passed the test overall. This will either be Test Passed in green or Test Failed in red.

- The Description column may tell you something about the nature of the test or the input.
- You must use **make** to compile and test your programs so that the **updates/log** file can help record your progress and provide some additional information, such as source file changes, that can be later integrated into your logbook.

Emulator

The skeleton **emulator.cpp** file contains the ***emulate_instruction()*** and ***disassemble_instruction()*** functions that you must complete. The ***main()*** function of the **emulator** program uses a while loop to read a test from standard input, then sets up the ROM, RAM and registers to run the test, then calls your ***emulate_instruction()*** to run the test and finally, prints the test inputs, test outputs and expected test outputs. If the emulator program is passed the command line argument "D", it will also use the ***disassemble_instruction()*** function to print a Hack Assembly Language version of the instruction being emulated.

emulate_instruction()

The ***emulate_instruction()*** will need to use the functions described in the **includes/hack_computer.h** file to read from / write to the A, D and PC registers, to read from the ROM and to read from / write to the RAM. The implementation should implement the following steps:

1. call ***read_PC()*** to find the address of the instruction to be emulated
2. call ***read_ROM()*** to read the 16-bit integer representing the instruction from ROM
3. emulate the instruction

Emulating an instruction first requires you to determine if you have an A-instruction or a C-instruction by inspecting bits 13 to 15. Given that all the values in the RAM, ROM and registers are implemented by a 16-bit integer type, uint16_t, you will need to perform some arithmetic to determine the bit's value using some combination of the operators, **~**, **&**, **|**, **>>**, **<<** and **==**. Review the **Bitwise Operators**

(https://myuni.adelaide.edu.au/courses/64329/pages/bitwise-operators) page for more information.

If you do not have an A-instruction or a C-instruction do nothing.

If you have an A-instruction, simply copy the instruction to the A register and add 1 to the PC register.

If you have a C-instruction you will need to first simulate the execution of the ALU to generate an output value and the negative and zero outputs. The ALU output value can then be written to some combination of the A register, D register or RAM as determined by the destination bits. Finally the PC register should have 1 added or, if the negative and zero outputs in combination with the jump bits required a jump to be preformed, the PC register should be given the original value of the A register. If an instruction attempts to read from a non-existent address in ROM or RAM, a 0 will be returned. If an instruction attempts to write to the keyboard register or a non-existant address in RAM, the write is ignored.

The instruction emulation will be an implementation of the equivalent HDL code that you wrote for workshop 04.

Note:

- the registers and RAM behave like the real hardware, they only change value at the end of a clock cycle, so
- the read_*() functions always return the old values of the registers or RAM, and
- the write_*() functions only set the future values of the registers or RAM.

disassemble_instruction()

The ***disassemble_instruction()*** function is called when the emulator program is asked to display an Assembly Language representation of the instruction being emulated. The 16-bit integer representation of the instruction is passed to the ***disassemble_instruction()*** and a string value is returned.

Disassembly first requires you to determine if you have an A-instruction or a C-instruction by inspecting bits 13 to 15.

If you do not have an A-instruction or a C-instruction return **"** illegal instruction **"**.

If you have an A-instruction, simply return a string containing "@" followed by the the decimal value of the instruction.

If you have a C-instruction you will need to first separate out the bits corresponding to each of the three components, the destination, the ALU operation and the jump using some combination of the operators, **~**, **&**, **|**, **>>** and **<<**. The 3 destination bits should give you a value in the range 0 to 7. The 3 jump bits should give you a value in the range 0 to 7. The ALU operation bits, if you include the a-bit, should give you a value in the range 0 to 127. The **includes/hack_computer.h** file also includes definitions of the lookup functions, ***destination()***, ***aluop()*** and ***jump()*** that take an integer and return a string representation of the destination, aluop and jump fields of a C-instruction respectively.

Note:

- if the destination bits are all 0, ***destination()*** returns **""**, otherwise it returns a string ending with **"="**
- if the jump bits are all 0, ***jump()*** returns **""**, otherwise it returns a string starting with **";"**
- if the ALU op bits do not match an instruction that can be generated in Hack Assembly Language, ***aluop()*** returns an expression enclosed in dots, eg **".-D-M-1."**

Test Details

If you want to see the actual output or other details of the tests, there are a few options you can try.

- **make test test-id**
  This will only run the test(s) with the matching test-id. If you do not supply a test-id it runs all of the tests.
- **make live test-id**
  If the test-id matches a single test, the test is run as before but then a final run is made that outputs directly to the terminal.

- **make show test-id**
  This does not run any tests but it will show the differences between expected and actual outputs. If no test-id is given all tests are shown.
- **make Show test-id**
  This does not run any tests and is a more detailed version of show that lists the pathnames of all files used in testing as well as differences between your program output with *write_to_traces()* and *write_to_logs()* enabled and disabled. The test scripts disable *write_to_traces()* and *write_to_logs()* so that your debugging output does not interfere with marking. However, this viewing option lets you see you all of your debugging output. If no test-id is given all tests are shown.
- **make less test-id**
  This is the same as **make show** but the output is piped through less so you can view it page by page.
- **make Less test-id**
  This is the same as **make Show** but the output is piped through less so you can view it page by page.

Milestone Submission Tests

Your milestone submission will only be awarded marks for tests that require the correct emulation and disassembly of A-instructions.

Final Submission Tests

Your final submission will be awarded marks for tests that require the correct emulation and disassembly of A-instructions and C-instructions.

Secret Tests

In addition to the tests in the **tests** directory, the web submission system will run a number of ***secret*** tests that may contain various errors. **Note:** these tests are ***secret***, if your programs fail any of these ***secret*** tests you **will not** receive any feedback about these ***secret*** tests, even if you ask!