Practical Assignment 6

Due 14 Oct by 23:59 **Points** 80 **Submitting** an external tool

Assessment Overview

Weighting:	80 Points (8% of course grade)
Due date:	Friday 14 Oct 11:59 pm (Week 10) Gradescope open for submissions now, with full Autograder available before Week 9.
Task description:	Write VM programs to complete the tasks described below and a Translator to convert those programs to Assembly Code. Doing so should help you to: • Understand how a stack-machine handles computations and logic. • Understand how variable scopes work at a low level. • Understand how a stack-machine is implemented at a low level. Please post your questions on Piazza or ask during your workshop.
Academic Integrity Checklist	Do ✅ Discuss/compare high level approaches ✅ Discuss/compare program output/errors ✅ Regularly submit your work as you progress Be careful ❓ Using online resources to find the solutions rather than understanding them yourself won't help you learn. Do NOT ❌ Submit code not solely authored by you. ❌ Use a public GitHub repository (use a private one instead). ❌ Post/share complete VM/Assembly/Machine code in Piazza/Discord or elsewhere on the Internet etc. ❌ Give/show your code to others

Your Task

Your task for this practical assignment is to write a translator to convert VM language programs into Hack assembly code.

1. Download **this zip file** (https://myuni.adelaide.edu.au/courses/72399/files/11667622/download?download_frd=1) containing template and test files for this assignment.
2. Complete the VM files and VMTranslator as described and as outlined below.
   - Submit your work regularly to Gradescope as you progress.
   - Additional resources and help will be available during your workshop sessions.
3. Test your code and write your **own test cases**.

Testing Requirement

Low level code can be especially prone to errors.

To help you develop, understand, and debug your own code **you'll also need to write several test cases for each task**.

- These test cases will be manually reviewed after the assignment due date.
- Marks for each task may be **scaled down as much as 50%** for poor/missing testing.
- The Gradescope autograder will run your test cases and provide *some* basic feedback.
- The additional resources section below includes basic instructions and guides on writing test cases.
- We also recommend asking your workshop supervisors for advice on testing if you're unsure.

Part 1 - Basic Program (3 points)

In this part you'll be familiarise yourself with Hack VM code by writing a basic arithmetic program.

You'll also need to write your own tests. Take a look at the sample test file provided to see how to write your own test cases.

Task 1.1 - Add and Subtract (3 points)

Write a program in Hack VM code to calculate

Complete the code in `AddSub.vm`

Where:

- `a` & `b` are both **local** variables (supplied in that order)
- `x` is a **static** variable

Test Cases:

- Write at least 2 test cases.
- A sample test case is provided in `AddSub00.tst`
- Each test case should be in a file named `AddSubXX.tst` where `XX` is a number starting at `01`.
- You should also submit any supporting files, such as CMP files.
- Your mark for this task may be **scaled down** for poor/missing testing.

Part 2 - Conditionals & Loops (17 points)

In this part you'll be writing more complex programs that involve gotos.

Task 2.1 - Absolute Value (7 points)

Write a program in Hack VM code to calculate the [absolute value](https://en.wikipedia.org/wiki/Absolute_value) of a given number

Complete the code in `Abs.vm`

Where:

- `x` and `y` are **static** variables (supplied in that order)

Test Cases:

- Write at least 2 test cases.
- A sample test case is provided in `Abs00.tst`
- Each test case should be in a file named `AbsXX.tst` where `XX` is a number starting at `01`.
- You should also submit any supporting files, such as CMP files.
- Your mark for this task may be **scaled down** for poor/missing testing.

Task 2.2 - Multiply (10 points)

Write a program in Hack VM code to multiply 2 numbers .

Complete the code in `Mult.vm`

Where:

- `a` is a **local** variable
- `x` & `y` are both **static** variables (supplied in that order)

Test Cases:

- Write at least 2 test cases.
- A sample test case is provided in `Mult00.tst`
- Each test case should be in a file named `MultXX.tst` where `XX` is a number starting at `01`.
- You should also submit any supporting files such as CMP files.
- Your mark for this task may be **scaled down** for poor/missing testing.

Part 3 - Functions & Arrays (28 points)

It's time to start using functions with differing variable scopes, and pointers to work with array data structures.

Task 3.1 - Fibonacci (12 points)

Write a **function** `Fib.fib( n )` in Hack VM code to calculate the n-th Fibonacci number (https://en.wikipedia.org/wiki/Fibonacci_number) **recursively**.

Complete the code in `Fib.vm`

Where:

- `Fib.fib` is the name of the function
- `n` is which number in the Fibonacci sequence to calculate,
  Where:
  - `Fib.fib(0) == 0`
  - `Fib.fib(1) == 1`

- The call command for this function is provided in a separate file (See `Sys.vm`)

Test Cases:

- Write at least 3 test cases.
- A sample test case is provided in `Fib00.tst`
- Each test case should be in a file named `FibXX.tst` where `XX` is a number starting at `01`.
- You should also submit any supporting files, such as CMP files.
- Your mark for this task may be **scaled down** for poor/missing testing.

Task 3.2 - Array Largest (16 points)

Write a **function** `ArrMax.arrMax( m , n )` in Hack VM code to calculate the largest value in a given Array.

Complete the code in `ArrMax.vm`

Where:

- `ArrMax.arrMax` is the name of the function
- `m` is a pointer to the Array
- `n` is the number of elements in the Array

- The `pointer` and `that` segments should be used to access the array. *See section 11.1.6 in the text book.*
- The call command for this function is provided in a separate file (See `Sys.vm`)

Test Cases:

- Write at least 3 test cases.
- A sample test case is provided in `ArrMax00.tst`
- Each test case should be in a file named `ArrMaxXX.tst` where `XX` is a number starting at `01`.
- You should also submit any supporting files, such as CMP files.
- Your mark for this task may be **scaled down** for poor/missing testing.

Part 4 - VM Translator (32 points)

We've written programs in VM Code, but do we understand how this relates to the Assembly code we've been working with?

Using your preferred programming language (Python, C++ or Java) implement a VMTranslator as described below.

- Template files are provided for each of these programming languages.
  - Download the Python version **HERE** (https://myuni.adelaide.edu.au/courses/72399/files/11668129/download?download_frd=1) .
  - Download the Java version **HERE** (https://myuni.adelaide.edu.au/courses/72399/files/11668128/download?download_frd=1) .
  - Download the C++ version **HERE** (https://myuni.adelaide.edu.au/courses/72399/files/11668127/download?download_frd=1) .
- You will need to complete the methods provided.
- Submit your completed source and test files in the `VMTranslator` directory.
- Only submit files for 1 programming language.

Task 4.1 - Push & Pop (6 points)

Complete the `vm_push` & `vm_pop` methods.

These methods should return Hack Assembly code that do the following:

`vm_push`

- Read the value from the correct memory segment, then push that value to the stack.
- Constant values need to be emulated.

`vm_pop`

- Pop a value from the stack, then write that value to the correct memory segment.

Test Cases:

- Write at least 2 test cases per method.
- Each test case should be in a file named `METHODTestXX.vm` where `METHOD` is the name of the method and `XX` is a number starting at `01`.
- See the section *Writing Tests* below for details on how to write test cases.
- Your mark for this task may be **scaled down as much as 50%** for poor/missing testing.

Task 4.2 - Arithmetic Operations (up to 2 points)

Complete any 1 of the following methods:

These methods should return Hack Assembly code that do the following:

`vm_add`

- Pop 2 values from the stack, **add** them, then push then result back to the stack.

`vm_sub`

- Pop 2 values from the stack, **subtract** them, then push then result back to the stack.

`vm_neg`

- Pop 1 value from the stack, **negate** it (i.e. flip its sign), then push the result back to the stack.

Test Cases:

- Write at least 1 test case per method.
- Each test case should be in a file named `METHODTestXX.vm` where `METHOD` is the name of the method and `XX` is a number starting at `01`.
- See the section *Writing Tests* below for details on how to write test cases.
- Your mark for this task may be **scaled down as much as 50%** for poor/missing testing.

Task 4.3 - Logic Operations (up to 4 points)

Complete any 2 of the following methods:

These methods should return Hack Assembly code that do the following:

`vm_eq`

- Pop 2 values from the stack, and compare them, then push the result back to the stack.
  - If they are **equal**, then push TRUE (-1) back to the stack, otherwise push FALSE (0)

`vm_gt`

- Pop 2 values from the stack, and compare them, then push the result back to the stack.
  - Compare the second value from the top of the stack to the value at the top of the stack (See chapter 7.3 in the Text book)
  - If the second value is **greater** than the top value, then push TRUE (-1) back to the stack, otherwise push FALSE (0)

`vm_lt`

- Pop 2 values from the stack, and compare them, then push the result back to the stack.
  - Compare the second value from the top of the stack to the value at the top of the stack (See chapter 7.3 in the Text book)
  - If the second value is **less** than the top value, then push TRUE (-1) back to the stack, otherwise push FALSE (0)

`vm_and`

- Pop 2 values from the stack, perform a bit-wise **and** on them, then push the result back to the stack.

`vm_or`

- Pop 2 values from the stack, perform a bit-wise **or** on them, then push the result back to the stack.

`vm_not`

- Pop 1 value from the stack, perform a bit-wise **not/invert** on it, then push the result back to the stack.

Test Cases:

- Write at least 1 test case per method.
- Each test case should be in a file named `METHODTestXX.vm` where `METHOD` is the name of the method and `XX` is a number starting at `01`.
- See the section *Writing Tests* below for details on how to write test cases.
- Your mark for this task may be **scaled down as much as 50%** for poor/missing testing.

Task 4.4 - Jump Operations (8 points)

Complete the `vm_label`, `vm_goto` & `vm_if` methods.

These methods should return Hack Assembly code that do the following:

`vm_label`

- Creates a label that can be used with jump instructions.

`vm_goto`

- Performs an unconditional jump to the location marked by the provided label.

`vm_if`

- Pop a value from the stack. If that value is **not FALSE** (not 0), jump to the location marked by the provided label.

Test Cases:

- Write at least 2 test cases per method.
- Each test case should be in a file named `METHODTestXX.vm` where `METHOD` is the name of the method and `XX` is a number starting at `01`.
- See the section *Writing Tests* below for details on how to write test cases.
- Your mark for this task may be **scaled down as much as 50%** for poor/missing testing.

Task 4.5 - Function Operations (12 points)

Complete the `vm_function`, `vm_call` & `vm_return` methods.

These methods should return Hack Assembly code that do the following:

`vm_function`

- Marks the beginning of a function with a given name and a number of local variables.
- This includes:
  - Generating a label for the program to jump to when the function is called.
  - Initialising the local variables to 0 by pushing the correct number of 0s to the stack.

`vm_call`

- Calls a function with a given name and a number arguments.
- This includes:
  - Generating a label for the program to return to when the function is returns.
  - Saving the stack frame.
  - Updating the memory segment pointers to their new locations.
  - Jumping to the label for the function.

`vm_return`

- Returns from the current function.
- This includes:
  - Copying the return value to the correct location on the stack.
  - Restoring the memory segment pointers with the values from the stack frame.
  - Jumping to the return label (which is stored in the stack frame).

Test Cases:

- Write at least 2 test cases per method.
- Each test case should be in a file named `METHODTestXX.vm` where `METHOD` is the name of the method and `XX` is a number starting at `01`.
- See the section *Writing Tests* below for details on how to write test cases.
- Your mark for this task may be **scaled down as much as 50%** for poor/missing testing.

You're done!

Submit your work to Gradescope using the button below.

- You may submit via file upload or GitHub.
  - If using GitHub, ensure your repository is private.
- Your submission should keep the provided folder structure, where the provided files and folders are either

- In the root of your submission (i.e. no subdirectory)

  ~ or ~
- In a directory named `prac6`

- Your Assembler implementation source files should be:
  - In the `VMTranslator` subdirectory.
  - Only contain the files for 1 programming language

Be sure to submit all files with each submission.

Additional Resources

The following resources may help you complete this assignment:

- **Chapters 7 & 8 of the Text Book** for VM programming and implementation
- Week 9 & 10 Workshops on Hack VM Code
- **Guide to Testing and Writing Test Cases**
- **Appendix 3 of the Text Book** for specification of the test language used in test cases.

This tool needs to be loaded in a new browser window

Load Practical Assignment 6 in a new window