Project 7: Virtual Machine - Stack Arithmetic and Memory Access

HONORS credits: Mandatory Assignment
https://piazza.com/class/jqmr1gc0d6z6jj?cid=244

Non-Honors credits: Optional Assignment
https://piazza.com/class/jqmr1gc0d6z6jj?cid=243

Submission Due Date: 04/21/2019 (Sunday) 11:59pm eCampus
Demo: 75 points + Code Review: 25 points (Date TBA)

Deliverables & Submission: Turn in a zip file in the format ***FirstName-LastName-UIN.zip*** containing all the files needed to compile and run your VM translator.

Put your full name in the introductory comments for each of your files.

Use relevant code comments and indentation in your code. All good programming practice expected and required

Submit this zip file on eCampus.

Late Submission Policy: Refer to the Syllabus

No make-up for missing Demo unless University-approved excuse. Refer to Syllabus for more details.

*************PLAGIARISING CODE WILL BE TAKEN VERY SERIOUSLY*************

Background

Java (or C#) compilers generate code written in an intermediate language called *bytecode* (or *IL*). This code is designed to run on a virtual machine architecture like the JVM (or CLR). One way to implement such VM programs is to translate them further into lower-level programs written in the machine language of some concrete (rather than virtual) host computer. In projects 7 we build such a *VM translator*, designed to translate programs written in the VM language into programs written in the Hack assembly language. The VM language, abstraction, and translation process are described in chapters 7 and 8 of the book, although the primary content needed to complete this project comes from chapter 7.

Objective

Build a ***VM translator*****,** focusing on the implementation of the VM language's *stack arithmetic* and *memory access.*We will NOT be implementing program flow (branches, function calls and returns) in this project.

Contract

Write a VM-to-Hack translator, conforming to the *VM Specification, Part I* (book section 7.2) and the *Standard VM-on-Hack Mapping, Part I* (book section 7.3.1). Use your VM translator to translate the VM programs supplied with this project, yielding corresponding programs written in the Hack assembly language. When executed on the supplied *CPU emulator*, the translated code generated by your translator should deliver the results mandated by the test scripts and compare files supplied with this project.

Resources

The relevant reading for this project is chapter 7 and all related lecture+class notes.
Youtube Video Playlist: **Unit 1.0 - 1.10**
https://www.youtube.com/playlist?list=PLrDd_kMiAuNmllp9vuPqCuttC1XL9VyVh

You will need two tools: the programming language with which you will implement your VM translator, and the supplied *CPU emulator*. This emulator allows executing, and testing, on your PC, the machine code generated by your VM translator. Another tool that comes handy in this project is the supplied *VM emulator*. The VM emulator (described in this document and in the book section 7.5) allows experimenting with the supplied VM programs before setting out to write your VM translator.

Proposed Implementation

We propose implementing the basic VM translator API described in chapter 7 in two stages. This will allow you to unit-test your implementation incrementally, using the test programs supplied below. In what follows, when we say "your VM translator should implement some VM command" we mean "your VM translator should translate the given VM command into a sequence of Hack assembly commands that accomplish the same task".

Stage I: Handling stack arithmetic commands: The first version of your basic VM translator should implement the nine arithmetic / logical commands of the VM language as well as the VM command push constant x.
The latter command is the generic push command for which the first argument is constant and the second argument is some non-negative integer x. This command comes handy at this early stage, since it helps provide values for testing the implementation of the arithmetic / logical VM commands. For example, in order to test how your VM translator handles the VM add command, we can test how it handles the VM code push constant 3, push constant 5, add. The other arithmetic and logical commands are tested similarly.

Stage II: Handling memory access commands: The next version of your basic VM translator should include a full implementation of the VM language's push and pop commands, handling the eight memory segments described in chapter 7. We suggest breaking this stage into the following sub-stages:

1. You have already handled the constant segment;
2. Next, handle the segments local, argument, this, and that;

3. Next, handle the pointer and temp segments, in particular allowing modification of the bases of the this and that segments.
4. Finally, handle the static segment.

Testing

We supply **five VM programs**, designed to unit-test the staged implementation proposed above. For each program Xxx we supply four files. The Xxx.vm file contains the program's VM code. The XxxVME.tst script allows running the program on the supplied *VM emulator*, to experiment with the program's intended operation. After translating the program using your VM translator, the supplied Xxx.tst script and Xxx.cmp compare file allow testing the translated assembly code on the supplied *CPU emulator*.

Testing how the VM translator handles *arithmetic commands*:

Program	Description	Test Scripts
SimpleAdd.vm	Pushes two constants onto the stack and adds them up.	SimpleAddVME.tst SimpleAdd.tst SimpleAdd.cmp
StackTest.vm	Executes a sequence of arithmetic and logical operations on the stack.	StackTestVME.tst StackTest.tst StackTest.cmp

Testing how the VM translator handles *memory access commands*:

Program	Description	Test Scripts
BasicTest.vm	Executes push/pop operations using the virtual memory segments constant, local, argument, this, that, and temp.	BasicTestVME.tst BasicTest.tst BasicTest.cmp
PointerTest.vm	Executes push/pop operations using the virtual memory segments pointer, this, and that.	PointerTestVME.tst PointerTest.tst PointerTest.cmp
StaticTest.vm	Executes push/pop operations using the virtual memory segment static.	StaticTestVME.tst StaticTest.tst StaticTest.cmp

Tips

Initialization: In order for any translated VM program to start running, the translated program (written in Hack assembly code) must include a preamble startup code that forces the VM implementation to start executing it on the host platform. In addition, in order for the translated code to operate properly, the base addresses of the virtual memory segments must be stored

somewhere on the host RAM. Both issues - startup code and segments initializations - are described in section 8.3 of chapter 8.

Steps:
For each one of the five test programs mentioned above and supplied with this project, follow these steps:

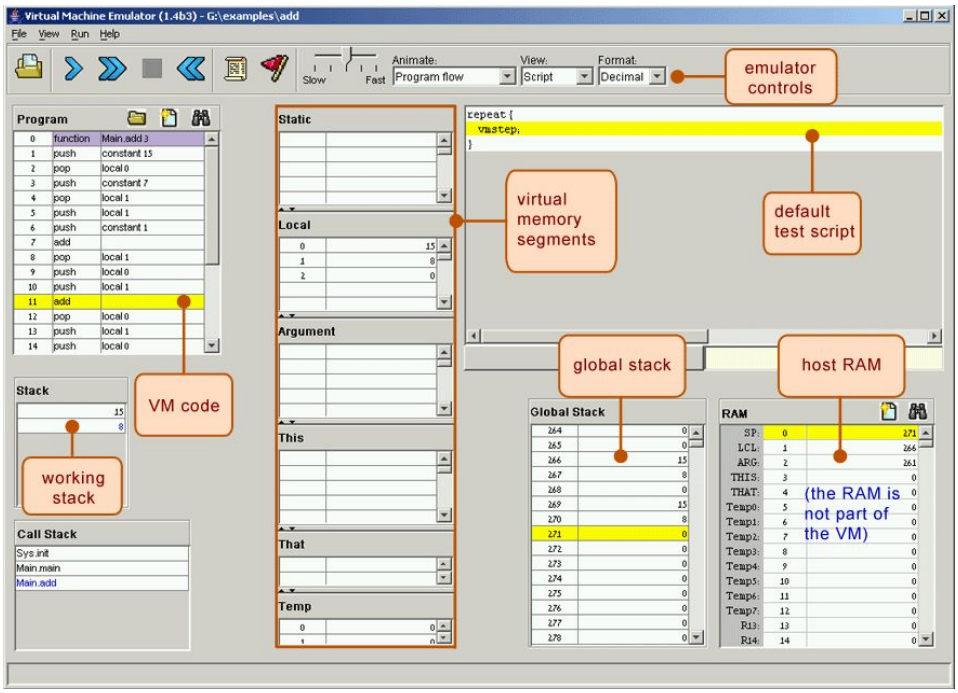
1. To get acquainted with the intended behavior of the supplied test program Xxx.vm, run it on the supplied *VM emulator* using the supplied XxxVME.tst script.
2. Use your VM translator to translate the supplied Xxx.vm file. The result should be a new text file containing Hack assembly code, named Xxx.asm.
3. Inspect the Xxx.asm program generated by your VM translator. If there are visible syntax (or any other) errors, debug and fix your VM translator.
4. To check if the generated code performs properly, use the supplied Xxx.tst and Xxx.cmp files to run the Xxx.asm program on the supplied *CPU emulator*. If there are any problems, debug and fix your VM translator.

Implementation Order: The supplied test programs were carefully planned to test the incremental features introduced by each development stage of your basic VM translator. Therefore, it's important to implement your translator in the proposed order, and to test it using the appropriate test programs at each stage. Implementing a later stage before an early one may cause the test programs to fail.

Tools
Before setting out to develop your VM translator, we recommend getting acquainted with the virtual machine architecture model and language. As mentioned above, this can be done by running, and experimenting with, the supplied .vm test programs using the supplied *VM emulator*.

The VM emulator: This Java program, which should be in your nand2tetris/tools directory, is designed to execute VM programs in a direct and visual way, without having to first translate them into machine language. For example, you can use the supplied VM emulator to see - literally speaking - how push and pop commands use and modify the stack. And, you can use the simulator to execute any one of the supplied .vm test programs.

Here is a typical screenshot of the VM emulator in action:



Rubric(75 points + 25 points code review)

Your program will be tested against 5 test files, each worth 15 points.

SimpleAdd.tst
StackTest.tst
BasicTest.tst
PointerTest.tst
StaticTest.tst

Programming Language Requirements

- If you write the program in C++, please make sure your program compiles using command: *g++ -std=c++11 *.cpp*;

After successful compiling, make sure an executable a.out or a.exe is generated. Then you should run your program with command: *./a.out <vm_file> or ./a.exe file.vm;*

- If you write the program in Java, make sure you put the main function in the file called VMtranslator.java and compile the program with: *javac *.java;* then you can translate a virtual machine code with command: *java VMtranslator <vm_file>*.

- If you write the program in Python, make sure you put the main function in the file called VMtranslator.python and make your program accept a command line argument (which is the vm file). So run your program like: *python VMtranslator.py <vm_file>*.
- Don't assume the vm files are in the same directory of your source files. When we test your program, the argument will be the path to the vm file.