CS220 – Project 8: Virtual Machine Translator II

Background

We continue building the VM Translator — a program that translates a program written in the VM language into a program written in the Hack machine language. This is a respectable chunk of engineering, so we are doing it in two stages. Welcome to Stage II..

Objective

Extend the basic VM translator built in Project 7 into a full-scale VM translator. In particular, in Project 7 we focused on handling the stack arithmetic and memory access commands of the VM language. We now turn to handle the language's branching and function calling commands.

Criteria for Success

Write a full-scale VM-to-Hack translator, extending the translator developed in Project 7, and conforming to the VM Specification, Part II (Section 8.2) and to the Standard VM-on-Hack Mapping, Part II (Section 8.3.1). Use your VM translator to translate he VM programs supplied below, yielding corresponding programs written in the Hack assembly language. When executed on the supplied CPU Emulator, the translated code generated by your VM translator should deliver the results mandated by the test scripts and compare files supplied below.

Resources

The relevant reading for this project is Chapter 8. You will need two tools: the programming language with which you implement your VM translator, and the supplied CPU Emulator. This emulator will allow you to execute, and test, the machine code generated by your VM translator. Another tool that comes handy in this project is the supplied visual VM Emulator. The emulator allows experimenting with a working VM implementation and with the given VM programs before you set out to translate them. For more information about this tool, refer to the supplied VM Emulator Tutorial.

Testing

We recommend completing the implementation of the VM translator in two stages. First, implement and test the translation of the VM language's branching commands, then implement and test the translation of the function call and return commands. This will allow you to unit-test your implementation incrementally, using the test programs supplied below.

Testing how the VM Translator handles branching commands:

Program	Description	Test Script
BasicLoop.vm	Computes the sum 1+2+...+ *n* and pushes the result onto the stack. This program tests the implementation of the VM language's branching commands: `goto` and `if-goto`.	BasicLoopVME.tst BasicLoop.tst BasicLoop.cmp
FibonacciSeries.vm	Computes and stores in memory the first *n* elements of the Fibonacci series. This typical array manipulation program provides a more challenging test of the VM's branching commands.	FibonacciSeriesVME.tst FibonacciSeries.tst FibonacciSeries.cmp

Testing how the VM Translator handles function call and return commands:

Program	Description	Test Script
SimpleFunction.vm	Performs a simple calculation and returns the result. This program provides a basic test of the implementation of the VM commands `function` and `return`.	SimpleFunctionVME.tst SimpleFunction.tst SimpleFunction.cmp
NestedCall: Sys.vm	An intermediate test (in terms of complexity) intended to be used between the SimpleFunction and FibonacciElement tests. It may be useful when SimpleFunction passes but FibonacciElement fails or crashes. NestedCall also tests several requirements of the Function Calling Protocol that are not verified by the other supplied tests. For more information and debugging advice read this NestedCall Test Guide, written by Mark Armbrust.	NestedCallVME.tst NestedCall.tst NestedCall.cmp
FibonacciElement: Main.vm Sys.vm (Since the program consists of more than one `.vm` file, the entire directory must be translated in order to produce a single `FibonacciElement.asm` file. See the note below for more information.	Tests the handling the VM's function calling commands, the bootstrap section, and most of the other VM commands. The program directory consists of two files: `Main.vm` contains one function named `fibonacci`. This recursive function returns the *n*'th element of the Fibonacci series. `Sys.vm` contains one function named `init`. This function calls the `Main.fibonacci` function with *n*=4, and then loops infinitely. (The bootstrap code of the VM implementation includes a default call to `Sys.init`.)	FibonacciElementVME.tst FibonacciElement.tst FibonacciElement.cmp
StaticsTest: Class1.vm Class2.vm Sys.vm (Ditto, see note below).	Tests the handling of static variables. Note that normally, the VM Translator is used to translate class files compiled by the *Jack Compiler*. Here we have two stand-alone, compiled class files, plus a `Sys.vm` file. The entire directory should be translated in order to produce a single `StaticsTest.asm` file.	StaticsTestVME.tst StaticsTest.tst StaticsTest.cmp

Notes: For `NestedCall`, `FibonacciElement`, and `StaticsTest`, your program will be translating the entire directory. For the remaining tests, you will be translating a single file. For `FibonacciElement`, and `StaticsTest`, your program should insert the bootstrap code at the beginning of the `asm` file. None of the remaining tests need the bootstrap code.

Handling programs consisting of more than one file: VM programs are rarely written by humans; they are normally generated by compilers. For example, Java compilers translate Java class files into intermediate VM code known as Bytecode. As we will see in the next projects, our Jack compilation model is very similar. A Jack program consists of one or more compilation units, known as classes. Each class is stored in a separate `.jack` file, all residing in the same directory — let's call it `MyProg`. Following compilation, the Jack compiler generates a set of corresponding `.vm` files, and stores them in the same `MyProg` directory. At this point the VM Translator enters the picture. If we wish to execute the MyProg code on the Hack platform, we apply the VM Translator to the entire MyProg directory (rather than to the individual `.vm` files). The result will be a single, monolithic `MyProg.asm` file containing the logic of the entire program. Therefore, your VM Translator should be capable of translating both an individual `.vm` file as well as a directory containing one or more `.vm` files.

Proposed Implementation

Chapters 7 and 8 include a proposed, language-independent VM Translator API, which can serve as your implementation's blueprint.

For each one of the six supplied test programs, 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` test script (if the program consists of one ore more files residing in a directory, load the entire directory into the VM Emulator and proceed to execute the code.)

2. Use your VM translator to translate the supplied `Xxx.vm` file, or directory, as needed. The result should be a new text file containing Hack assembly code. The name of this file should be `Xxx.asm`.

3. Inspect the translated `Xxx.asm` program. If there are visible syntax (or any other) errors, debug and fix your VM translator.

4. To check if the translated code performs properly, use the supplied `Xxx.tst` and `Xxx.cmp` files to run your translated `Xxx.asm` program on the supplied CPU Emulator. If there are run-time errors, keep working on your VM translator.

API: Chapter 8 includes a proposed, language-independent VM Translator API. This API can be the blueprint of your VM Translator implementation.

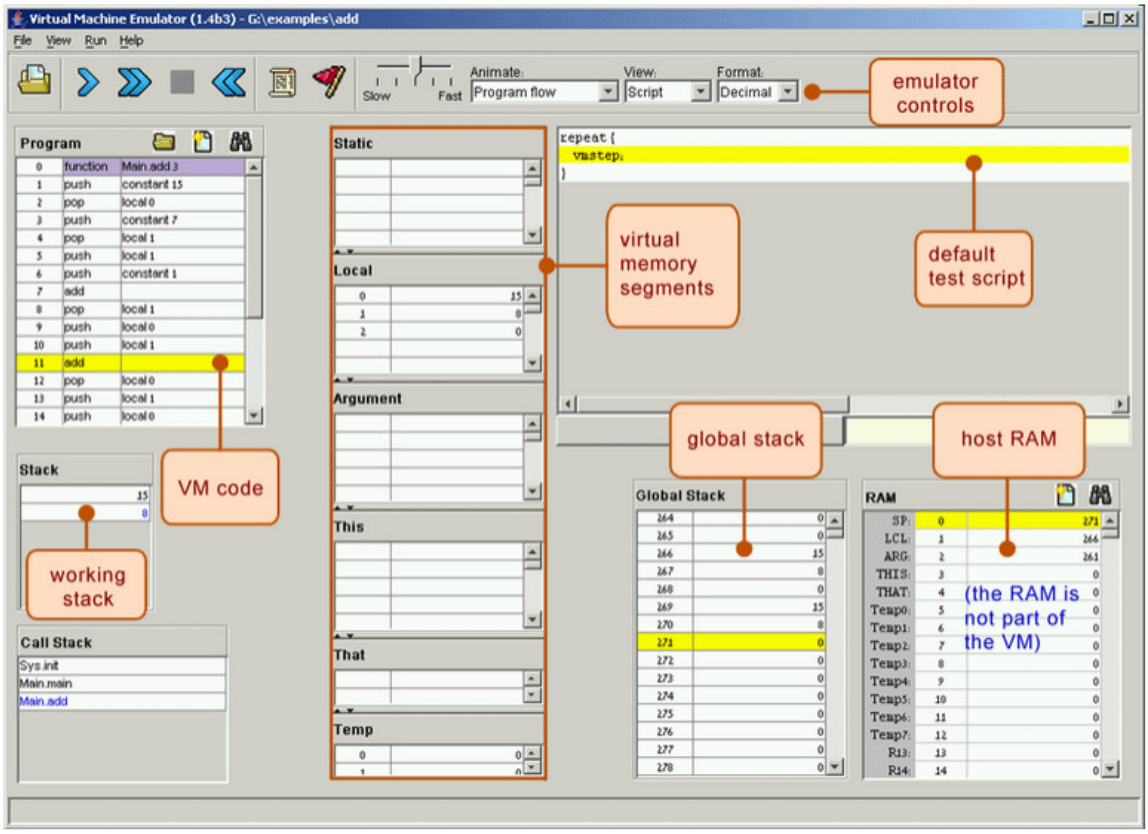
Implementation order: The supplied test programs were carefully planned to test the specific features of each stage in your VM implementation. Therefore, it's important to implement your VM translator in the proposed order, and to test it using the supplied test

programs at each stage. Implementing a later stage before an early one may cause the test programs to fail.

Initialization: In order for any translated VM program to start running, it must include a preamble startup code (boot code) that forces the VM implementation to start executing it on the host platform. In addition, in order for any VM code to operate properly, the VM implementation must store the base addresses of the virtual memory segments in the correct locations in the host RAM. The first four test programs in this project assume that the startup code was not yet implemented, and include test scripts that effect the necessary initialization "manually." The last two programs assume that the startup code is already part of the VM implementation.

Tools

The VM Emulator: This Java program, included in the Nand2Tetris Software Suite, executes VM programs in a direct and visual way, without having to first translate them into machine language. This allows you to experiment with the VM environment before setting out to build your own VM Translator. For example, you can use the supplied VM Emulator to see — literally speaking — how push and pop commands effect the stack. And, you can use it to execute any one of the supplied `.vm` test programs. Here is a typical screen shot of the VM Emulator in action:



Confused? Go through the supplied *VM Emulator Tutorial*.

Submission and Assessment

If you can't finish the project on time, submit what you've managed to do, and relax. All the projects in this course are highly modular, with incremental test files. Each hardware project consists of many chip modules (`*.hdl` programs), and each software project consists of many software modules (classes and methods). It is best to treat each project as a modular problem set, and try to work out as many problems as you can. You will get partial credit for partial work.

What if your chip or program is not working? It's not the end of the world. Hand in whatever you did, and explain what works and what doesn't in a README file. If you want, you can also supply test files that you developed, to demonstrate working and non-working parts of your project. Instead of trying to hide the problem, be explicit and clear about it. You will get partial credit for your work.

Submit all of your Eclipse project as a single ZIP archive.