Project 8: Virtual Machine II - Program Control

Background

We continue building the VM translator - a program that translates programs written in the VM language into programs 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 VM language's branching and function calling commands.

Contract

Write a full-scale VM-to-Hack translator, extending the basic translator developed in project 7, and conforming to the VM Specification, Part II (book section 8.2) and to the Standard VM-on-Hack Mapping, Part II (book section 8.3.1). Use your VM translator to translate the 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.

Note that a VM program may span either a single .vm file or a directory containing one or more .vm files. In the former case, the VM translator is invoked using the command VMTranslator fileName.vm. The translator creates an output file named fileName.asm, which is stored in the same directory of the input file. In the latter case, the VM translator is invoked using VMTranslator dirName. The translator creates a single output file named dirName.asm, which is stored in the same directory. This single file contains the assembly code resulting from translating all the .vm files in the input directory. The name of the input file or directory may contain a file path. If no path is specified, the VM translator operates on the current directory by default.

Resources

The relevant reading for this project is chapter 8. 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 at the bottom of this page) allows experimenting with the supplied VM programs before setting out to write your VM translator.

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. Next, 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 Scripts
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.tstFibonacciSeries.cmp

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

Program	Description	Test Scripts
SimpleFunction.vm	Performs a simple calculation and returns the result. This program provides a basic test of	SimpleFunctionVME.tst SimpleFunction.tstSimpleFunction.cmp

the implementation of the VM commands function and return.

Program	Description	Test Scripts
NestedCall: Sys.vm An optional and intermediate test, which may be useful when SimpleFunction (the previous test) passes but FibonacciElement (the next test) fails.	Tests several requirements of the function calling protocol. For more information about this optional test, see this guide and this stack diagram. Can be used with or without the VM bootstrap code.	NestedCallVME.tst NestedCall.tst NestedCall.cmp
Program	Description	Test Scripts
FibonacciElement: Main.vm Sys.vm Since the program consists of more than one .vm file, the entire directory must be translated. The translation should yield a single assembly file named FibonacciElement.asm. See the note below for more information on handling multiple .vm files.	Tests the handling of the VM's function calling commands, the bootstrap section, and most of the other VM commands. The program directory consists of two files, as follows. Main.vm contains one function named Main.fibonacci. This recursive function returns the n'th element of the Fibonacci series, and is unrelated to the Fibonacci series program described previously in this project. Sys.vm contains a single function named Sys.init. This function calls the Main.fibonacci function with n=4, and then loops indefinitely (the Sys.init function, in turn, will be called by the VM implementation's bootstrap code).	FibonacciElementVME.tst FibonacciElement.tst FibonacciElement.cmp Unlike the previous tests, this test assumes that the VM translator initializes the VM implementation. If the generated assembly file will not begin with bootstrap code, the test will fail.
Program	Description	Test Scripts
StaticsTest: Class1.vm Class2.vm Sys.vm	Class1.vm and Class2.vm contain VM functions designed to set and get various static values; this is done in order to test the handling of the static memory segment. Sys.vm contains a single Sys.init function that calls the get/set functions of Class1 and Class2	StaticsTestVME.tst StaticsTest.tst StaticsTest.cmp This test assumes that the VM translator initializes the VM implementation; if the generated assembly file will not begin with bootstrap code, the test will fail.

Proposed Implementation

For each one of the five 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 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 the translated Xxx.asm program on the supplied CPU emulator. If there are any problems, debug and fix your VM translator.

API: Chapter 8 includes a proposed, language-independent VM translator API. This API can serve as your implementation's blueprint.

Implementation order: The supplied test programs were carefully planned to test the incremental features introduced by 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 the translated VM code to execute on the host computer platform, the translated code stream (written in the machine language of the host platform) must include some bootstrap code that maps the stack on the host RAM and starts executing the code proper. The first three test programs in this project assume that the bootstrap code was not yet implemented, and include test scripts that effect the

necessary initializations (as was done in project 7). The last two test programs assume that the bootstrap code is generated by the VM translator. In other words, the assembly code that the final version of your VM translator generates must start with some code that sets the stack pointer and calls the Sys.init function. Sys.init will then call the Main.main function, and the program will start running (similar to how Java's JVM always looks for, and starts executing, a method named main).

Tools

Before setting out to extend your basic VM translator, we recommend playing with the supplied .vm test programs. This will allow you to experiment with branching and function call-and-return commands, 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 one function calls another, and how the called function does its act and returns a value to the caller. For more information, see the VM emulator tutorial (PPT, PDF)

Here is a typical screen shot of the VM emulator in action: