Project 8

In project 7 you built a basic VM translator that implements the VM language's arithmetic-logical and push/pop commands. In this project you will extend this translator to handle the VM language's branching and function commands. In addition, you will add the capability of translating multi-file VM programs. This will complete the development of the VM translator that will later serve as the compiler's backend.

Objective

Extend the basic VM translator built in project 7 into a full-scale VM translator, designed to handle multi-file programs written in the VM language. Assume that the source VM code is error-free.

Contract

Complete the construction of a VM-to-Hack translator, conforming to the VM Specification and Standard VM Mapping on the Hack Platform. Use your VM translator to translate the supplied VM test programs, yielding corresponding programs written in the Hack machine language. When executed on the supplied CPU emulator along with supplied test scripts, the code generated by your translator should deliver the results mandated by the supplied compare files.

Resources (same as in project 7)

You will need the programming language in which you implement your VM translator, and the code of the basic translator developed in Project 7. In addition, you will need two tools:

The CPU emulator is used for executing and testing the assembly code generated by your VM translator. If the generated code runs correctly in the CPU emulator, we will assume that your translator performs as expected. This of course is just a partial test of the translator, but it will suffice for our purposes.

The VM emulator is not required, but highly recommended, for this project. The VM emulator is designed to execute VM code in a simulated virtual machine. In the process, it visualizes how the VM code impacts the stack, the virtual memory segments, and the relevant RAM areas in which they are implemented on the host computer. Watching this action "in vivo" helps understand how the code generated by your VM translator should impact the host RAM.

Testing

We recommend completing the implementation of the VM translator in two stages. First, extend the basic VM translator built in project 7 to handle the VM branching commands. Next, extend it to handle the VM function commands. This staged development allows unit-testing your implementation incrementally, using the supplied test programs.

Testing the handling of the VM branching commands

The translation of the commands label, if, and if-goto is tested by the following programs:

BasicLoop: Computes 1 + 2 + ... + argument[0], and pushes the result onto the stack. Tests how the VM translator handles the label and if-goto commands.

FibonacciSeries: Computes and stores in memory the first *n* elements of the Fibonacci series. A more rigorous test of handling the label, goto, and if-goto commands.

Initialization of the stack and the memory segments is normally done by the complete VM implementation (namely, by the assembly code generated by the complete VM translator). In the above tests though, the initialization is done by the supplied test scripts.

Testing the handling of the VM function commands

The translation of the commands function, return and call is tested by the following programs:

SimpleFunction: Performs a simple calculation and returns the result. Tests how the VM translator handles the function and return commands. The supplied test script initializes the stack pointer, the memory segments, and a mock return address, and then calls this function.

NestedCall: An intermediate and optional test that can be used between the SimpleFunction and FibonacciElement tests. It may be useful when the former passes and the latter fails. See the test documentation for more details.

FibonacciElement: This test program consists of two files: Main.vm contains a single function, Main.fibonacci, that returns recursively the n'th element of the Fibonacci series; Sys.vm contains a single function, Sys.init, that calls Main.fibonacci with n=4, and then enters an infinite loop (the VM translator is supposed to generate bootstrap code that calls Sys.init). The resulting setting tests how the VM translator handles multiple .vm files, and how it handles the VM commands function, call, and return, and most of the other VM commands. In addition, this test program tests that the VM translator writes the bootstrap code that initializes the stack pointer and calls Sys.init. Since this test program consists of more than one .vm file, the folder (rather than a single file) must be translated in order to produce a single FibonacciElement.asm file that carries out all these operations.

StaticsTest: Tests the handling of static variables. The test program consists of three files: Class1.vm and Class2.vm contain functions that set and get the values of several static variables; Sys.vm contains a single function, Sys.init, that calls the functions in Class1.vm and in Class2.vm. Since the test program consists of more than one .vm file, the folder (rather than a single file) must be translated in order to produce a single StaticsTest.asm file.

Sys.init: The full-scale VM translator developed in this project must handle single- as well as multi-file programs. By convention, the first function that starts running in a VM program is Main.main. Conforming to this convention, Sys.init is normally programmed to call Main.main. For the purpose of this project though, the Sys.init functions supplied in some of the tests call the testing programs directly.

Implementation

The translation of the *branching* VM commands is relatively simple. The translation of the *function* commands is more challenging. We repeat the suggestion given in the previous project: Start by writing the assembly code that your translator should generate *on paper*. Draw the RAM representation of the global stack on paper, and keep track of the stack pointer and the relevant memory segment pointers. This will help you ensure that your paper-based assembly code successfully implements all the low-level actions associated with handling the function, call, and return commands. At this stage you can generalize your paper-based code and "copy-paste" it into the outputs that your VM translator should generate.

The supplied test programs were carefully planned to test the specific features of each stage in your evolving VM implementation. We recommend implementing your translator in the proposed order, and testing it using the appropriate test programs at each stage. Implementing a later stage before an early one may cause testing failures.

Since project 8 is based on extending the basic VM translator developed in project 7, we advise making a backup copy of the latter before starting this project.

Bootstrap code: In order for any translated VM program to start running, it must include startup code that invokes the program on the host platform. In addition, in order for any VM code to operate properly, the VM implementation must set the base address of the stack to some selected RAM location (on our platform, 256).

Each one of the first three test programs in this project – *BasicLoop*, *FibonaciiSeries*, *SimpleFunction* – include no function calls. Also, each one of these three programs is stored in a single .vm file. When running these three tests, we assume that the VM Translator generates no bootstrap code. We also assume that in these three programs, the VM translator handles no function calls. Therefore, the stack and the virtual segments are not initialized in the RAM. To compensate, the testing of these three programs feature test scripts that affect the necessary initializations "manually" (as you can see by inspecting their .tst files). The last two test programs – *FibonaciiElement* and *StaticTest* – assume that the VM Translator generates startup code and handles all the function-call-and-return VM commands.

So, in this project, there is a simple rule: If the VM Translator is called to translate one file only, it should inform the CodeWriter that it should not write the bootstrap code (that is, ignore the constructor's API guideline beginning with "Writes the assembly instructions that affect the bootstrap code..."). If the VM Translator is called to translate more than one .vm file, it should

inform the CodeWriter that it must generate bootstrap code. This requirement (write / don't write the bootstrap code) can be handled by the CodeWriter constructor.

References (same as in project 7)

You've already used the CPU emulator and VM emulator in previous projects, but we include references to them, for completeness.

CPU emulator demo

CPU emulator tutorial (click Slideshow)

VM emulator tutorial (click Slideshow)