Project 7

In this project you will build the first half of the first half of a typical two-tier compiler. Specifically, the Jack compiler generates VM code for a virtual machine, much like Java's compiler generates Bytecode. The VM code is then translated further into machine language, using a program named VM translator. In project 7 you will develop a basic version of this translator, and in project 8 you will complete it. This VM Translator is sometimes referred to as the compiler's backend.

Basically, you have to write a program that reads and parses VM commands, one command at a time, and generates Hack instructions that execute the command's semantics on the Hack computer. For example, how should the VM translator handle an input like "push constant 7"? Answer: it should output a sequence of Hack assembly instructions that implement this stack operation on the host RAM. Code generation – coming up with a sequence of Hack instructions that realize each one of the VM commands – is the very essence of this project.

Objective

Build a basic VM translator that implements the arithmetic-logical and push/pop commands of the VM language. For the purpose of this project, This version of the VM translator assumes that the source VM code is error-free. Error checking, reporting and handling can be added to later versions of the VM translator, but are not part of this project.

Resources

You will need three tools: the programming language in which you will implement your VM translator, and the supplied VM emulator and CPU emulator, available in your nand2tetris/tools folder.

The CPU emulator enables executing and testing the assembly code generated by your VM translator. If the generated assembly 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 a program that visualizes how running VM code impacts the stack, the memory segments, and the relevant RAM areas on the host RAM. For example, consider a test program that pushes a few constants onto the stack, and then pops them into some memory locations. You can run this test program in the VM emulator, inspect how the stack grows and shrinks, and see the impact on the relevant RAM area. Seeing how the VM commands impact the host RAM will help you figure out how to realize the same impact using assembly code – a critical requirement for writing the VM translator.

Contract

Write a VM-to-Hack translator, conforming to the Standard VM Mapping on the Hack Platform. Use your translator to translate the supplied test VM programs, yielding corresponding programs written in the Hack assembly language. When executed on the supplied CPU emulator, the assembly programs generated by your translator should deliver the results mandated by the supplied test scripts and compare files.

Testing

Test your evolving VM translator by translating the following test VM programs.

SimpleAdd: This program pushes two constants onto the stack, and adds them up. Tests how your implementation handles the commands "push constant i", and "add".

StackTest: Pushes some constants onto the stack, and tests how your implementation handles all the VM arithmetic-logical commands.

BasicTest: Executes push, pop, and arithmetic commands using the memory segments constant, local, argument, this, that, and temp. Tests how your implementation handles these memory segments (you've already handled constant).

PointerTest: Executes push, pop, and arithmetic commands using the memory segments pointer, this, and that. Tests how your implementation handles the pointer segment.

StaticTest: Executes push, pop, and arithmetic commands using constants and the memory segment static. Tests how your implementation handles the static segment.

Initialization: In order for any translated VM program to start running, it must include startup code that forces the generated assembly code to start executing on the host platform. And, before this code starts running, the VM implementation must anchor the base addresses of the stack and the virtual memory segments in selected RAM locations. Both issues – startup code and segment initializations – will be implemented by the final version of the VM translator, developed in project 8. Alas, these initializations are also needed in order to execute the test programs in project 7. The good news is that you need not worry about these details: Before starting to execute your generated assembly code, the supplied test scripts carry out all the necessary initializations.

Implementation

The VM translator is a program that generates assembly code. In order to write it, you must know how to write Hack assembly code. If necessary, review the assembly program examples in chapter 4, and the programs that you wrote in project 4. In particular, focus on code examples that manage pointers using Hack instructions.

For each VM command, your VM translator must write the Hack assembly code that implements it. We recommend starting by writing and testing these assembly code snippets *on paper*. Draw a RAM segment, draw a trace table that records the values of, say, SP and LCL, and initialize these variables to some arbitrary memory addresses. Now, track on paper the assembly code that you think realizes, say, "push local 2". Does the assembly code impact the relevant RAM areas correctly (on paper)? Did you remember to update the stack pointer? Once you feel confident that your assembly code snippet does it job correctly, you can have your VM translator generate it, almost as is.

We advise developing and testing your evolving translator on the test programs in the order in which they are listed above. This way, you will build the translator's code generation capabilities gradually, according to the demands presented by each test program.

We supply five sets of test programs, test scripts, and compare files. For each test program Xxx.vm we recommend following these steps:

0. Use the XxxVME.tst script to execute the test program Xxx.vm on the VM emulator. This will familiarize you with the operations of the test program. Inspect the stack and the virtual segments, and make sure that you understand what the test program is doing. Now execute the program again, interactively and step-wise, one VM command at a time. Inspect the impact of each VM command on the RAM implementations of the stack and the segments (bottom right of the VM simulator's GUI). The assembly code that your VM translator generates should have the same impact on the Hack RAM.

1. Use your partially implemented translator to translate Xxx.vm. The result should be a text file named Xxx.asm, containing the Hack assembly code generated by your translator.

2. Inspect the generated Xxx.asm code produced by your translator. If there are visible syntax (or any other) errors, debug and fix your translator.

3. Use the supplied Xxx.tst script to load, run and test, on the CPU emulator, the Xxx.asm program created by your VM translator. If there are any errors, debug and fix your translator.

When you are done with this project, be sure to save a copy of your VM translator. In the next project you will be asked to extend this program, so it's a good idea to keep a copy of a version of it that works properly.

References

You've already used the CPU emulator in previous projects, but we include references to it for completeness. The VM emulator is used in this project for the first time. It's important to get acquainted with this program, for this project and for all the subsequent projects in the course.

CPU emulator demo

CPU emulator tutorial (click Slideshow)

VM emulator tutorial (click Slideshow)