The Elements of Computing Systems / *Nisan & Schocken* / www.idc.ac.il/tecs

Project 7: The VM: Stack Arithmetic

Objective: Build the first part of the Virtual Machine translator (the second part is implemented in Project 8), focusing on the implementation of the *stack arithmetic* and *memory access* commands of the VM language.

Resources: You will need two tools: the programming language in which you will implement your VM translator, and the *CPU Emulator* supplied with the book. This emulator will allow you to execute the machine code generated by your VM translator -- an indirect way to test the correctness of the latter. Another tool that may come handy in this project is the visual *VM Emulator* supplied with the book. This program allows experimenting with a working VM implementation before you set out to build one yourself. For more information about this tool, refer to the *VM Emulator Tutorial*.

Contract: Write a VM-to-Hack translator, conforming to the *VM Specification, Part I* (Section 7.2) and the *Standard VM-on-Hack Mapping, Part I* (Section 7.3.1). Use it to translate the test `.vm` programs supplied below, yielding corresponding `.asm` 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.

Proposed Implementation Stages

We recommend building the translator in two stages. This will allow you to unit-test your implementation incrementally, using the test programs supplied below.

Stage I: Stack arithmetic commands: The first version of your VM translator should implement the nine stack arithmetic and logical commands of the VM language as well as the "`push constant x`" command (which, among other things, will help testing the nine former commands). Note that the latter is the generic `push` command for the special case where the first argument is "`constant`" and the second argument is some decimal constant.

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

0. You have already handled the `constant` segment;
1. Next, handle the segments `local`, `argument`, `this`, and `that`;
2. Next, handle the `pointer` and `temp` segments, in particular allowing modification of the bases of the `this` and `that` segments;
3. Finally, handle the `static` segment.

Test Programs

We supply five VM programs, designed to unit-test the proposed implementation stages described above. For each program `Xxx` we supply four files, beginning with the program's code in `Xxx.vm`. The `Xxx_vm.tst` script allows running the program on the supplied VM Emulator, so that you can gain familiarity with the program's intended operation. After translating the program using your VM Translator, the supplied `Xxx.tst` and `Xxx.cmp` scripts allow testing the translated assembly code on the CPU Emulator.

Testing the arithmetic commands implementation:

Program	Description	Test scripts
SimpleAdd.vm	Pushes and adds two constants.	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 the memory access commands implementation:

Program	Description	Test scripts
BasicTest.vm	Executes `pop` and `push` operations using the virtual memory segments.	BasicTestVME.tst BasicTest.tst BasicTest.cmp
PointerTest.vm	Executes `pop` and `push` operations using the `pointer`, `this`, and `that` segments.	PointerTestVME.tst PointerTest.tst PointerTest.cmp
StaticTest.vm	Executes `pop` and `push` operations using the `static` segment.	StaticTestVME.tst StaticTest.tst StaticTest.cmp

Tips

Start by creating a directory named `projects/07` on your computer and extracting project 07.zip to it (preserving the directory structure embedded in the zip file).

Initialization: In order for any translated VM program to start running, it must include a preamble startup 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 segments in the correct locations in the host RAM. Both issues -- startup code and segments initializations -- are implemented in the next project. The difficulty of course is that we need these initializations in place in order to run the test programs given in this project. The good news are that you should not worry about these issues at all, since the supplied test scripts carry out all the necessary initializations in a manual fashion (for the purpose of this project only).

Steps: For each one of the five test programs, follow these steps:

1. Run the `Xxx.vm` program on the supplied VM Emulator, using the `XxxVME.tst` test script, to get acquainted with the intended program's behavior.
2. Use your partial translator to translate the `.vm` file. The result should be a text file containing a translated `.asm` program, written in the Hack assembly language.
3. Inspect the translated `.asm` program. If there are visible syntax (or any other) errors, debug and fix your translator.
4. Use the supplied `.tst` and `.cmp` files to run your translated `.asm` program on the CPU Emulator. If there are run-time errors, debug and fix your translator.

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 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

The VM Emulator: This Java-based VM implementation illustrates how the VM works, using visual GUI and animation effects. Specifically, it allows executing VM programs directly, without having to translate them first into machine language. This practice enables experimentation with the VM environment before you set out to implement one yourself. Here is a typical screen shot of the VM Emulator in action: