Project 7: Virtual Machine I - Stack Arithmetic

Background

Java (or C#) compilers generate code written in an intermediate language called bytecode (or IL). This code is designed to run on a virtual machine architecture like the JVM (or CLR). One way to implement such VM programs is to translate them further into lower-level programs written in the machine language of some concrete (rather than virtual) host computer. In projects 7 and 8 we build such a VM translator, designed to translate programs written in the VM language into programs written in the Hack assembly language. The VM language, abstraction, and translation process are described in chapters 7 and 8 of the book. For the purpose of this project, chapter 8 can be ignored.

Objective

Build a basic VM translator, focusing on the implementation of the VM language's stack arithmetic and memory accesscommands. In Project 8, this basic translator will be extended into a full-scale VM translator.

Contract

Write a VM-to-Hack translator, conforming to the VM Specification, Part I (book section 7.2) and to the Standard VM-on-Hack Mapping, Part I (book section 7.3.1). Use your VM translator to translate the VM files supplied below, yielding corresponding programs written in the Hack assembly language. When executed on the supplied CPU emulator, the translated code generated by your translator should deliver the results mandated by the test scripts and compare files supplied below.

Usage

Depending on the programming language that you use, the VM translator should be invoked using something like "VMTranslator fileName.vm", where the string fileName.vm is the translator's input, i.e. the name of a text file containing VM commands. The translator creates an output file named fileName.asm, which is stored in the same directory of the input file. The name of the input file may contain a file path.

Resources

The relevant reading for this project is chapter 7. 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.

Proposed Implementation

We propose implementing the basic VM translator API described in chapter 7 in two stages. This will allow you to unit-test your implementation incrementally, using the test programs supplied below. In what follows, when we say "your VM translator should implement some VM command" we mean "your VM translator should translate the given VM command into a sequence of Hack assembly commands that accomplish the same task".

Stage I: Handling stack arithmetic commands: The first version of your basic VM translator should implement the nine arithmetic / logical commands of the VM language as well as the VM command push constant x.

The latter command is the generic push command for which the first argument is constant and the second argument is some non-negative integer x. This command comes handy at this early stage, since it helps provide values for testing the implementation of the arithmetic / logical VM commands. For example, in order to test how your VM translator handles the VM add command, we can test how it handles the VM code push constant 3, pushconstant 5, add. The other arithmetic and logical commands are tested similarly.

Stage II: Handling memory access commands: The next version of your basic VM translator should include a full implementation of the VM language's push and pop commands, handling the eight memory segments described in chapter 7. We suggest breaking this stage into the following sub-stages:

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

Testing

We supply five VM programs, designed to unit-test the staged implementation proposed above. For each program Xxxwe supply four files. The Xxx.vm file contains the program's VM code. The XxxVME.tst script allows running the program on the supplied VM emulator, to experiment with the program's intended operation. After translating the program using your VM translator, the supplied Xxx.tst script and Xxx.cmp compare file allow testing the translated assembly code on the supplied CPU emulator.

Testing how the VM translator handles arithmetic commands:

Program	Description	Test Scripts
SimpleAdd.vm	Pushes two constants onto the stack and adds them up.	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 how the VM translator handles memory access commands:

Program	Description	Test Scripts
BasicTest.vm	Executes push/pop operations using the virtual memory segments constant, local, argument, this, that, and temp.	BasicTestVME.tst BasicTest.tst BasicTest.cmp
PointerTest.vm	Executes push/pop operations using the virtual memory segments pointer, this, and that.	PointerTestVME.tst PointerTest.tst PointerTest.cmp
StackTest.vm	Executes push/pop operations using the virtual memory segment static.	StaticTestVME.tst StaticTest.tst StaticTest.cmp

Tips

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. In addition, the virtual memory segments must be mapped on selected areas in the host RAM. The bootstrap code and the memory mappings are described in chapter 8 and handled by the final version of the VM translator, implemented in the next project. In the current project you should worry about none of these issues, since all the necessary VM initializations and memory mappings are handled by the supplied test scripts. In other words, the assembly code that your translator generates should contain the translation of the VM commands found in the input file, and nothing else.

Steps:

For each one of the five test programs supplied above, 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.
2. Use your VM translator to translate the supplied Xxx.vm file. The result should be a new text file containing Hack assembly code, named Xxx.asm.
3. Inspect the Xxx.asm program generated by your VM translator. If there are visible syntax (or any other) errors, debug and fix your VM translator.
4. To check if the generated code performs properly, use the supplied Xxx.tst and Xxx.cmp files to run the Xxx.asm program on the supplied CPU emulator. If there are any problems, debug and fix your VM translator.

Implementation Order: The supplied test programs were carefully planned to test the incremental features introduced by each development stage of your basic VM translator. 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.

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 modify and extend this program, adding the handling of more VM commands. If your project 8 modifications will end up breaking some of the working code developed in project 7, you'll be able to resort to a clean backup version.

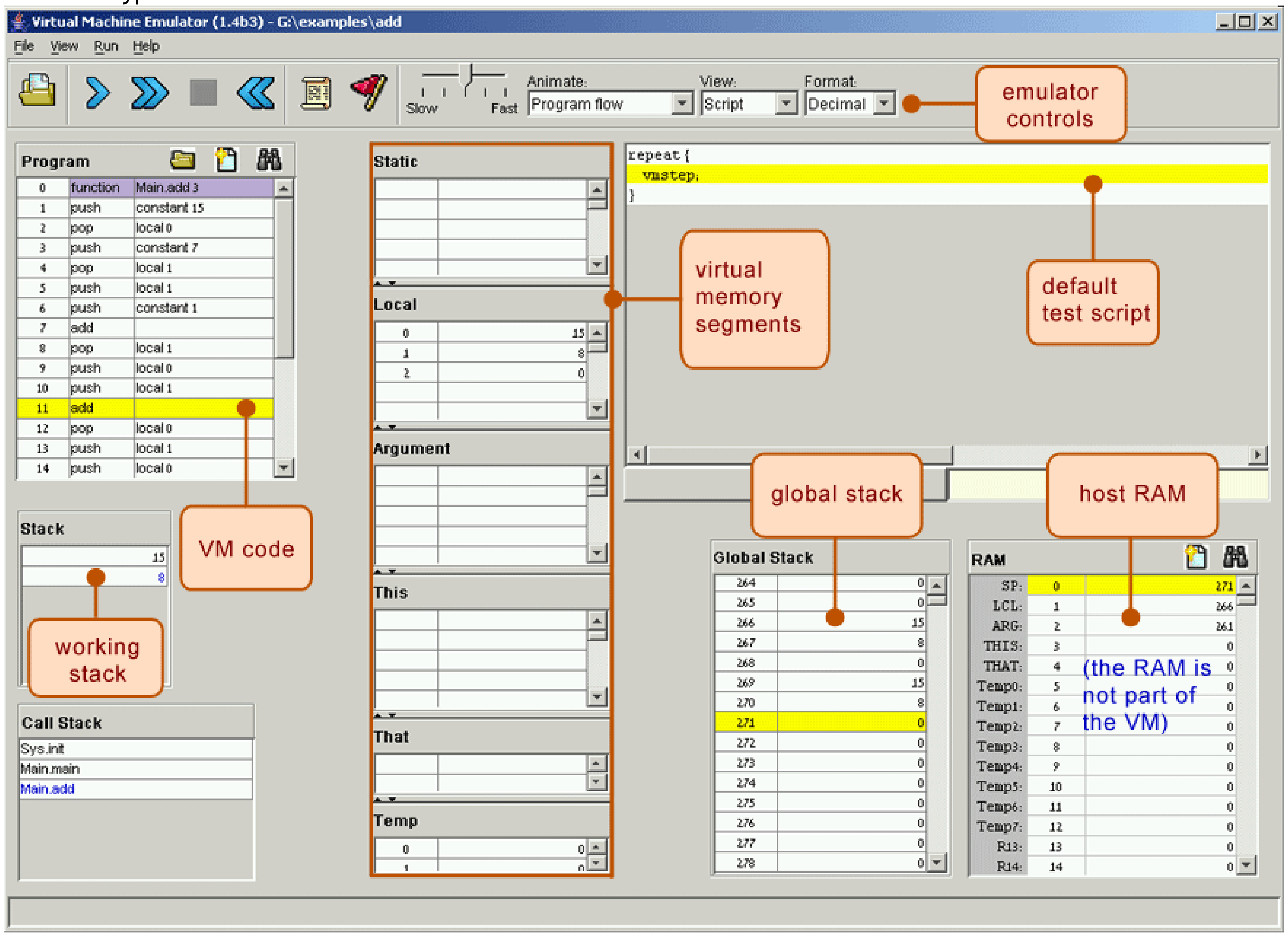
Tools

Before setting out to develop your VM translator, we recommend getting acquainted with the virtual machine architecture model and language. As mentioned above, this can be done by running, and experimenting with, the supplied .vm test programs using the supplied VM emulator.

The VM emulator: This Java program, located 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 push and pop commands effect the stack. And, you can use the simulator to execute any one of the supplied .vm test programs.
For more information, see the VM emulator tutorial (PPT, PDF)

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



Animate: Program flow View: Script Format: Decimal

Program

0	function	Main.add 3
1	push	constant 15
2	pop	local 0
3	push	constant 7
4	pop	local 1
5	push	local 1
6	push	constant 1
7	add	
8	pop	local 1
9	push	local 0
10	push	local 1
11	add	
12	pop	local 0
13	push	local 1
14	push	local 0

Stack

15
8

Call Stack

Sys.init
Main.main
Main.add

Static

Local

0	15
1	8
2	0

Argument

This

That

Temp

0	0
1	0

```
repeat {
  vmstep;
}
```

Global Stack

264	0
265	0
266	15
267	8
268	0
269	15
270	8
271	0
272	0
273	0
274	0
275	0
276	0
277	0
278	0

RAM

SP:	0	271
LCL:	1	266
ARG:	2	261
THIS:	3	0
THAT:	4	0
Temp0:	5	0
Temp1:	6	0
Temp2:	7	0
Temp3:	8	0
Temp4:	9	0
Temp5:	10	0
Temp6:	11	0
Temp7:	12	0
R13:	13	0
R14:	14	0