# Project 7: Virtual Machine Translator I

CS 220

## Background

When you compile a Java (or C#) program, the compiler generates code written in an intermediate language called Bytecode (or IL). In theory, this code is designed to run in a virtual machine environment like the JVM (or CLR). But, before the VM code can be executed on a real computer, it first has to be translated into the native code of that computer. The program that carries out this latter translation is called a VM Translator. Projects 7 and 8 illustrate how such VM Translators are implemented. This is done by introducing a simple VM language (similar to Java's Bytecode), and building a VM Translator that translates VM code into code written in the Hack assembly language. The VM language, abstraction, and translation process are described in detail in Chapters 7 and 8.

## Objective

Build the first part of a VM translator, focusing on the implementation of the stack arithmetic and memory access commands of the VM language. In Project 8, this basic translator will be extended further into a fully-functioning 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 and test the given VM programs, 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 with the project.

## 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. Your VM translator implementation can be written in any programming language, but check with me before using any programming language other than Java. We've provided starter code for an Eclipse Java project. The CPU Emulator will allow you to execute, and test, the machine code generated by your VM translator. Another tool that comes handy in this project is the supplied visual VM Emulator. The emulator allows experimenting with a working VM implementation and with the given VM programs before you set out to translate them. For more information about this tool, refer to the supplied *VM Emulator Tutorial*.

## Proposed Implementation

Chapter 7 includes a proposed, language-independent VM Translator API, which can serve as your implementation's blueprint. We propose building your VM Translator 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 VM translator should implement the nine arithmetic/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: Handling memory access commands:** The next version of your 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, 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.

## Test Programs

We supply five VM programs, designed to unit-test the staged implementation proposed above. For each program `Xxx` we supply four files, as follows. The `Xxx.vm` file contains the program's VM code. The `XxxVME.tst` script allows running the program on the supplied VM Emulator, to glean 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 then adds them up. | `SimpleAddVME.tst`<br>`SimpleAdd.tst`<br>`SimpleAdd.cmp` |
| `StackTest.vm` | Executes a sequence of arithmetic and logical operations on the stack. | `StackTestVME.tst`<br>`StackTest.tst`<br>`StackTest.cmp` |

**Testing how the VM Translator handles memory access commands:**

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

## Tips

**Initialization:** In order for any translated VM program to start running, the translated program (written in Hack assembly code) must include a preamble startup code that forces the VM implementation to start executing it on the host platform. In addition, in order for the translated code to operate properly, the base addresses of the virtual segments must be stored in the correct locations in the host RAM. Both issues — startup code and segment initializations — are implemented in Project 8. The difficulty of course is that we need these initializations in place in order to run the test programs of 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 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` test 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. 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 your translated `Xxx.asm` program on the supplied CPU Emulator. If there are run-time errors, keep working on your VM translator.

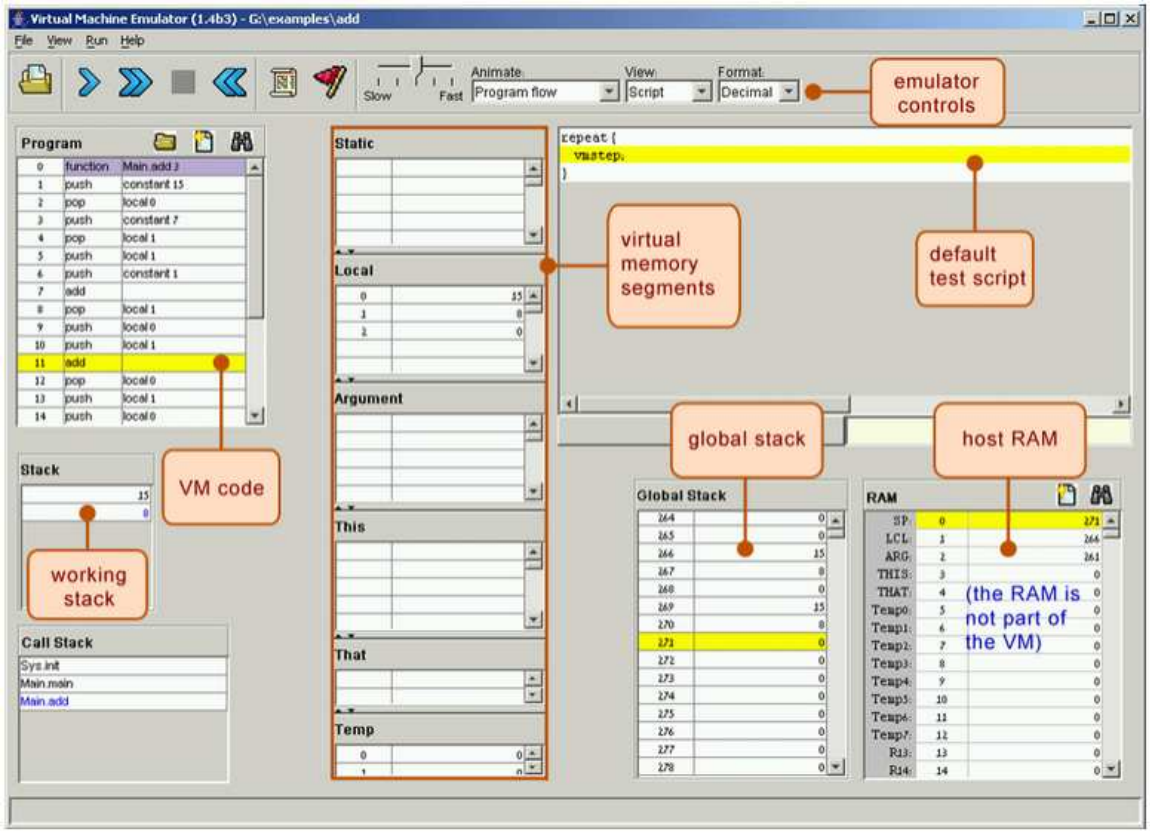
**Implementation Order:** The supplied test programs were carefully planned to test the specific features of each stage in your VM implementation. Therefore, it's important to implement your translator in the proposed order, and to test it using the appropriate test programs at each stage. Implementing a later stage before an early one may cause the test programs to fail.

## Tools

In a psychological experiment run in Israel a few years ago, engineering students were asked to plan a 100-kilometer pipe to transport blood between two cities. The students submitted all sorts of fancy plans, yet no-one bothered to ask "Why such a project is needed in the first place?" Something similar may well happen in this project. You can develop a fancy VM Translator that translates VM code into assembly code perfectly, without stopping to wonder why we need a virtual machine in the first place, and what the translated VM code is actually supposed to do. In order

to avert this situation, we recommend that you get acquainted with the Virtual Machine model, and play with the supplied VM test programs, before you set out to write your VM Translator. This can be done using the supplied VM Emulator.

**The VM Emulator:** This Java program, included in the Nand2Tetris Software Suite, executes VM programs in a direct and visual way, without having to first translate them into machine language. This allows you to experiment with the VM environment before setting out to build your own VM Translator. For example, you can use the supplied VM Emulator to see — literally speaking — how push and pop commands affect the stack. And, you can use it to execute any one of the supplied `*.vm` test programs. Here is a typical screen shot of the VM Emulator in action:




**Confused?** Go through the supplied *VM Emulator Tutorial*.

## Submission and Assessment

If you can't finish the project on time, submit what you've managed to do, and relax. All the projects in this course are highly modular, with incremental test files. Each hardware project consists of many chip modules (`*.hdl` programs), and each software project consists of many software modules (classes and methods). It is best to treat each project as a modular problem set, and try to work out as many problems as you can. You will get partial credit for partial work.

What if your chip or program is not working? It's not the end of the world. Hand in whatever you did, and explain what works and what doesn't in a README file. If you want, you can also supply test files that you developed, to demonstrate working and non-working parts of your project. Instead of trying to hide the problem, be explicit and clear about it. You will get partial credit for your work.

See the next page for the assessment rubric. Submit the following as a single Eclipse ZIP archive in Canvas:

1. A README file containing the names of all group members. This file may also contain other information, as described above. Create this file as a new `Untitled Text File` in your Eclipse project.

2. Your Eclipse project.

3. Nothing else.

# Project 7: Virtual Machine I: Stack Arithmetic

Student name(s): _____

***Grading method:*** As usual with programming assignments, we look for elegance, clarity, reasonable documentation, and neatness.

| ***VM I*** | ***Working?*** | | ***Comments*** |
|---|---|---|---|
| Working? | / 50 | Does the program generate assembly output that will work on all test inputs? | |
| Well built? | / 40 | 12 points for documentation. But, don't over-document! For every method you write, document what it does, what parameters it takes, and what it returns. Use your judgment to add more documentation when needed.<br><br>8 points for a VM translator that produces efficient assembly code. Generally speaking, the fewer the number of generated assembly instructions, the better.<br><br>20 points for a good and clean implementation that we can easily read and understand. | |
| README | / 10 | Names of group members; if project doesn't work, comments on things that don't work and how you've tried to fix them. | |
| Total | / 100 | | |

Total grade: _____