8.5 Project

Objective Extend the basic VM translator built in Project 7 into a full-scale VM translator. In particular, add the ability to handle the program flow and function calling commands of the VM language.

Resources (same as Project 7) 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 in 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 full-scale VM-to-Hack translator, extending the translator developed in Project 7, and conforming to the VM Specification, Part II (section 8.2) and to the Standard VM Mapping on the Hack Platform (section 8.3.1). Use it to translate the VM programs supplied below, yielding corresponding programs written in the Hack assembly language. When executed on the supplied CPU emulator, these assembly programs should deliver the results mandated by the supplied test scripts and compare files.

Testing Programs

We recommend completing the implementation of the translator in two stages. First implement the program flow commands, then the function calling commands. This will allow you to unit-test your implementation incrementally, using the test programs supplied here.

For each program Xxx, the XxxVME.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.

Test Programs for Program Flow Commands

■ BasicLoop: computes $1 + 2 + \cdots + n$ and pushes the result onto the stack. This program tests the implementation of the VM language's goto and if-goto commands.

■ Fibonacci: computes and stores in memory the first n elements of the Fibonacci series. This typical array manipulation program provides a more challenging test of the VM's branching commands.

Test Programs for Function Calling Commands

■ SimpleFunction: performs a simple calculation and returns the result. This program provides a basic test of the implementation of the function and return commands.

■ FibonacciElement: This program provides a full test of the implementation of the VM's function calling commands, the bootstrap section, and most of the other VM commands.

The program directory consists of two .vm files:

● Main.vm contains one function called fibonacci. This recursive function returns the *n*-th element of the Fibonacci series;

● Sys.vm contains one function called init. This function calls the Main.fibonacci function with $n = 4$, then loops infinitely.

Since the overall program consists of two .vm files, the entire directory must be compiled in order to produce a single FibonacciElement.asm file. (compiling each ● vm file separately will yield two separate .asm files, which is not desired here).

■ StaticsTest: A full test of the VM implementation's handling of static variables. Consists of two .vm files, each representing the compilation of a stand-alone class file, plus a Sys.vm file. The entire directory should be compiled in order to produce a single StaticsTest.asm file.

(Recall that according to the VM Specification, the bootstrap code generated by the VM implementation must include a call to the Sys.init function).

Tips

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 selected locations in the host RAM. The first three test programs in this project assume that the startup code was not yet implemented and include test scripts that effect the necessary initializations "manually." The last two programs assume that the startup code is already part of the VM implementation.

Testing/Debugging For each one of the five test programs, follow these steps:

1. Run the 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(s), yielding a single .asm text file that contains a translated 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.

Note: The supplied test programs were carefully planned to unit-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 Same as in Project 7.