Week 10 assignment: A Hack VM translator (phase 2)

1 Tasks

1. Finish the Hack VM translator from the previous week and test it with the scripts provided.

If you haven't got `StackTest.vm` compiling from week 9's assignment, you should finish work on that before starting this week's assignment. (It's fine if you haven't finished week 9's assignment completely, though.)

2 Required software

For this lab, you will need the VM emulator and CPU emulator from the nand2tetris software suite. The demonstrations in the video lectures should give you a good idea of how to use this software, and the official documentation is available from the unit page. All of it runs on Windows, Linux and Mac OS.

In order to run this software (and anything else from the nand2tetris suite) on your home computer, you will need to install the Java Runtime Environment, which you can download here. (It's already installed on the lab computers.) If you are getting an error about javaw.exe being missing, the most likely reason is that you don't have the Java Runtime Environment installed.

3 Support for function and return statements

We'll start out by working in single-file mode, so don't worry about the new `compile_folder` function yet. We recommend you base your code for this week on the new skeleton rather than your code from last week's assignment. If you choose to copy your old code into this week's skeleton, there are two things to watch out for:

- The `THIS` enum value and the `TokenType` struct name clashed with imports needed for the folder-scanning code on Windows, so they've been renamed to `KW_THIS` and `HackTokenType` respectively, so you'll need to rename them for `list_vm_files` to work.
- The code in `parse_file` that initialises `SP` to 256 has been commented out, as some of the test scripts this week start with a non-empty stack to simulate previous function calls.

We can compile function and return statements in the same way as we compiled everything last week: by adding an appropriate case statement to `parse_instruction` and defining a function that outputs appropriate assembly code. The skeleton contains outlines of the two key functions `parse_function` and `parse_return`. Like last week, you'll need to fill out one `sprintf` statement per function. **You should assume that the call frame is stored on the stack in exactly the same order as in lectures for the test scripts to work.**

Remember from last week that `get_next_label_name(abc, xyz)` works out an unused Hack assembly label name for a VM file with filename `abc`, then stores that label name in `xyz`. We've also given you a (very simple) similar function `get_function_label(abc, xyz)` which, given a function name `abc` and a buffer `xyz`, works out a standard form for the label that should go at the start of the Hack assembly code for function `abc` and stores it in `xyz`.

To test your code, use the same procedure as last week with the provided `SimpleFunction.vm` test code (Test 1 in the test data). There is no `call` statement in the VM file — instead the accompanying test script sets up the stack as it would be following a `call SimpleFunction.test 2` instruction with return address 9. Like last week, remember that your `.asm` output file will need to to be in the same folder as the test script and `.cmp` comparison file for the test script to work. We also strongly recommend you debug by stepping through the VM code on the VM emulator (with the supplied test script) to see exactly what each statement does to the stack, and then comparing this against the results of your own assembly code.

4 Support for call statements

Now fill out the `sprintf` statement for the `parse_call` function. To test your code, use the `NestedCall.vm` test code (Test 2 in the test data). Again there is no explicit call to `Sys.init`, and the test script sets up the file with a fake call frame. Note that the file to compile is called `Sys.vm`, and you will need to compile it to a file called `NestedCall.asm` for the test script to work.

When debugging, note that You shouldn't expect your call frame to match the VM simulator's call frame in the return address (and the test scripts won't test for this), as the return address will depend on your exact implementation of the VM translator. You may also find the debug information included in the `.html` files useful.

5 Support for compiling a folder

It's now time to go from compiling a single file to compiling a whole folder. The process of scanning through a folder and opening a handle to every `.vm` file is *surprisingly* unpleasant even by C's normal standards. It's also platform-dependent, which is an issue when developing on both Windows and Linux machines. Overall, solving this problem would have limited educational value even as part of the C unit, so we've solved it for you in the new skeleton. The existing `compile_folder` function will add appropriate assembly code to the start of the output file, then compile every file in the folder by calling the usual `lex_file` and `parse_file` functions. **Read through the code of `compile_folder` and make sure you understand it, especially the Hack assembly code.** (Don't bother trying to understand `is_folder` or `list_vm_files`, they're platform-specific black magic.)

Now **add code to the start of `compile_folder`** to initialise `SP` to 256 and simulate a call to `Sys.init`. You can assume this function exists in the folder being compiled. You shouldn't need to know anything about its code except that it can't contain any `return` statements (so you don't have to worry about how your code will handle them).

Now test your code using texttttFibonacciElement.vm (Test 3). When running your code, you should supply the whole folder as an argument rather than a single file. Unlike the previous two sections, the test script won't start by setting up a call frame for you. Finally, try compiling and testing the `StaticsTest.vm` test code (Test 4), again supplying the whole folder as an argument. This shouldn't need any extra code on your part, and should work or fail based on last week's implementation of the `static` segment.