Week 9 assignment: A Hack VM translator (phase 1)

1 Tasks

1. Write a Hack VM translator in C covering this week's material and test it on the scripts provided.

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 Code skeleton

A list of tokens and a grammar for Hack VM can be found in the slides for the fourth video this week. However, most of the lexing and parsing process is very similar to the lexing and parsing process for the Hack assembler last week, so we have written most of the code for you. You won't need symbol tables for this week's assignment, but we've included a tweaked version of token.c and token.h from last week. The existing code in main.c will lex the given input file to a file called `temp.lex` using the `lex_file`, `lex_line` and `lex_token` functions provided. **Spend a few minutes to understand the code that's already here.** It's similar to the lexing code from week 8, but without the need to add labels to a symbol table.

After creating `temp.lex`, the translator parses the result, generating Hack assembly code for each instruction as it's parsed. The `parse_file` function first outputs code to set the stack pointer to 256, then breaks `temp.lex` into instructions by scanning for newline tokens using the `get_next_instruction` function. It sends each instruction to the `parse_instruction` function, which looks at the first token in it to tell what type of instruction it is, then calls an appropriate function such as `parse_push` or `parse_add`. This function then copies correct assembly code to the provided `dest` string, which `parse_instruction` writes to the appropriate output file. When every instruction has been parsed, the end of `parse_file` appends an infinite loop to the output file. **Spend a few minutes to understand the code that's already here.** Again, this is very similar to the parsing code from week 8 — all that's missing is the Hack assembly code to output for each instruction.

Your job will be to write this Hack assembly code and finish the VM translator.

4 Pushing and adding constants

First, you will implement commands of the form `push constant [number]` and `add`. Load `SimpleAdd.vm` into the VM emulator and step through the code. Observe what changes if you load another value into RAM[0] in the lower-right corner of the window. (This value must be between 256 and 2045 to avoid `SimpleAdd.vm` crashing with a stack overflow.)

Write Hack assembly code to simulate the effects of each instruction of `SimpleAdd.vm` on RAM one by one. To match `SimpleAdd.vm`, your code should:

- add 7 to the stack by writing it to RAM[SP];
- increase the stack size by one by incrementing SP;
- add 8 to the stack by writing it to RAM[SP];
- increase the stack size by one by incrementing SP;

- decrease the stack size by one by decrementing SP;
- add the previous top two elements of the stack (stored in RAM[SP] and RAM[SP-1]);
- store the result in the new top of the stack at RAM[SP-1].

When your code is complete, save it as `SimpleAdd.asm` in the same folder as `SimpleAdd.tst` and `SimpleAdd.cmp` from the test data on the unit page (in the Test 1 subfolder). Then in the CPU emulator, go to File → Load Script (**not** Load Program) and load `SimpleAdd.tst`. Run it, and you should see "End of script - Comparison ended successfully" at the bottom of the window. If you instead see an error, then your assembly file is not behaving in the same way as the VM file and you have a bug. This script is taken from the original Nand2Tetris course, as are the others in this section.

When your code is working, using the ideas behind it, go into `main.c` and **fill in the missing code** in `parse_add` and in the `constant` case of `parse_push`. This code uses the `sprintf` function to put the correct code into `code`, then copy it into `dest`; `sprintf` allows you to specify a string with format specifiers, like `printf`. For example, the following code:

```
sprintf(code, "@%d\n
               D=A\n", x);
```

copies the two assembly instructions `@[x]` and `D=A` into `code`, replacing `[x]` with the value of the C variable x. Unlike last week, there is very little C coding in this assignment — you will "just" be filling in strings of `sprintf` statements to the appropriate strings of Hack code. When filling in the code for the `constant` case of `parse_push`, the integer argument (i.e. 7 or 8 in `SimpleAdd.vm`) is stored in `address->value.int_val`.

WARNING: In the long run, we want your C program to be able to compile arbitrary VM code, and that VM code might use any part of RAM. The one exception is RAM[13], RAM[14] and RAM[15], which are reserved for use in expanding VM instructions to Hack assembly — if you ever need to store data in memory, then it should be stored in one of these three addresses.

USEFUL TIP: You should include Hack assembly comments at the start and end of each VM instruction you compile, e.g. "`// BEGIN PUSH CONSTANT`" and "`// END PUSH CONSTANT`". This will make it much easier to read the output later as you debug your code.

When you have added code to these two functions, **compile your code and run it on the provided test script** `SimpleAdd.vm` (in the test 1 folder) — the first input is the VM code to compile, and the second input is the desired output file, which should be `SimpleAdd.asm`. As before, load the `SimpleAdd.tst` test script in the CPU emulator to check you have compiled `SimpleAdd.vm` correctly.

5 Arithmetic and logical operations

Next you will implement commands of the form `sub`, `neg`, `eq`, `lt`, `gt`, `and`, `or` and `not` in the corresponding functions `parse_sub`, `parse_neg`, `parse_eq` and so on. When writing code for `eq`, `gt` and `lt` in particular, you may feel the need to generate a label in assembly. The `get_next_label_name` function provided at the end of the file will always write a new valid label name into the provided string `dest`; **take a few minutes to understand the code**. You may notice that there is a fair amount of duplicated Hack assembly code between some of these functions — feel free to create some new C functions to cut down on this duplication, but the scale is small enough that copy-pasting is also a viable approach.

When you have implemented all these commands, you should run your code on `StackTest.vm` (in the test 2 folder) to test it.

6 Pushing and popping to other memory segments

Next you will implement `push` commands to, and `pop` commands from, all the memory segments other than `constant` — that is, `local`, `argument`, `this`, `that`, `temp`, `pointer` and `static`. (We don't need to implement `pop` commands to `constant`, since this memory segment can't be written to.) Fill out the remaining code in the `parse_push` and `parse_pop` functions — details on how each memory segment should be mapped to RAM can be found in the lecture videos. Again, you may want to create some new C functions to cut down on code duplication, but this is optional.

When you have implemented all these commands, you should run your code on `BasicTest.vm` (test 3), `PointerTest.vm` (test 4) and `StaticTest.vm` (test 5) in that order. As the names suggest, `PointerTest.vm` tests the `pointer` segment and `StaticTest.vm` tests the `static` segment. If you would like to see how the original VM code behaves under the same memory mapping as in the test script, then the VM emulator test scripts (e.g. `BasicTestVME.tst`) will set up the VM emulator with appropriate values in e.g. `SP` and `LCL` for you.

7 Branching

Finally, implement `goto`, `label` and `if-goto` commands by filling in code for the `parse_label`, `parse_goto` and `parse_ifgoto` functions. Be careful here — your code should be robust to e.g. a user-specified label that just happens to be of the same form as the auto-generated labels outputted by `get_next_label_name`.

When you have implemented these commands, you should run your code on `BasicLoop.vm` (test 6), then `FibonacciSeries.vm` (test 7) to test it. At this point, you have a functional VM translator! There are only a few features missing, which we'll talk about next week and which you'll implement in a lab similar to this one.