NestedCall Test Synopsis

NestedCall.tst is an intermediate test (in terms of complexity) intended to be used between the SimpleFunction and FibonacciElement tests. It may be useful when SimpleFunction passes but FibonacciElement fails or crashes. NestedCall also tests several requirements of the Function Calling Protocol that are not verified by the other supplied tests. NestedCall can be used with or without the VM bootstrap code.

NestedCallVME.tst runs the same test on the VM Emulator.

The NestedCall tests and supporting documentation were written by Mark Armbrust.

Test Structure

Startup

NestedCall is implemented entirely within the Sys.vm file. The first function in Sys.vm is Sys.init(). This allows it to be used before the bootstrap code has been added to the VM Translator since there will be no file processing order issues.

NestedCall loads Sys.asm, sets up the stack to simulate the bootstrap's call to Sys.init(), then begins execution at the beginning of Sys.asm. If the bootstrap is not present, the program begins running with Sys.init() since it is the first function in Sys.vm.

If Sys.asm includes the bootstrap, the bootstrap will (re)initialize the stack and call Sys.init(), so the test should see the same environment either way it gets to Sys.init().

Sys.init()

`THIS` and `THAT` are set to known values so that context save and restore can be tested.

Sys.init() calls Sys.main() and stores the return value in `temp 1`. This tests call to and return from a function with no arguments.

Sys.main()

Sys.init() allocates 5 local variables. It sets `local 1`, `local 2` and `local 3`. `local 0` and `local 4` are intentionally not set.

`THIS` and `THAT` are changed so that context save and restore can be tested.

Sys.main() calls Sys.add12(123) and stores the return value in `temp 0`. This tests call to and return from a function with arguments.

After Sys.add12() returns, Sys.main() sums `local 0` through `local 4` and returns the result. This tests that the local segment was properly allocated on the stack and that the local variables were not overwritten by the call to Sys.main(). It also tests that `local 0` and `local 4` were properly initialized to 0.

Sys.add12()

`THIS` and `THAT` are set to known values so that context save and restore can be tested.

Returns `argument 0` plus 12.

Test Coverage

Functions with no arguments return to correct RIP (Return Instruction Point) with correct return value on stack.
This can fail if the RIP is not correctly pushed on the stack by the calling code, or if the returning code does not store the RIP in a temporary register before overwriting it with the return value.

Functions with arguments return to correct RIP with correct return value on stack.
This can fail if it is assumed that `ARG` points to the RIP.

Functions with local variables allocate space on the stack for the local variables.
This can fail if the function prologue is not written or if the SP is not updated after zeroing the local variables.

All local variables are initialized to 0.
Common errors are to forget this completely, or for the zeroing loop to be off by one.

`THIS` and `THAT` are correctly retained across function calls. Looking ahead, in Project 9 you will be asked to write a simple computer game in the high-level Jack language. You can run your game (following compilation) on the supplied VM Emulator. But, if you choose to translate the VM code that the compiler generates using *your* VM Translator, then code like "`push THIS`, `push THAT` ... `pop THIS`, `pop THAT`" can cause some interesting failures!

Debugging

These comments assume that your VM translator has passed the SimpleFunction test.

If `RAM[0]` is incorrect, you have a stack skew. More data was pushed onto the stack by `call` than was popped by `return`, or vice versa. See *debugging with breakpoints* later in this section.

If one or more of `RAM[1]` through `RAM[4]` is incorrect, the `LCL`, `ARG`, `THIS` and `THAT` pointers are not being correctly saved or restored. Most likely problem is when they are being saved; the SimpleFunction test verified that `return` restored them correctly.

If `RAM[5]` is incorrect there may be a problem with setting up the `ARG` pointer.

If `RAM[4]` is incorrect and `RAM[5]` is correct, there may be a problem with allocation or initialization of local variables.

Debugging with breakpoints

To find tough bugs you can use the "breakpoint" facility in the CPU Emulator (red flag button). You can use breakpoints to have you program stop when it gets to a particular RAM address. For example:

- load the NestedCall.tst file,
- set a PC breakpoint at the ROM address for `(Sys.main)`,
- hit the run button.

When the CPU Emulator stops at the breakpoint you can inspect the RAM to check the stack and pointers values. (If the breakpoint isn't hit, you will need to to single-step debug through your calling code to see why it didn't get there.)

Other useful places to set breakpoints are the entry points to the other functions and at the first and final instructions generated for `return` commands.

NestedCallStack.html shows the expected stack values at various points during the test.

Finding ROM address in your ASM code

It is not easy to find the ROM locations where you want to set breakpoints, because there is no one-to-one correspondence between the ASM file line numbers and the ROM addresses. This is made even more difficult because the supplied CPU Emulator does not display the (LABELS) in its ROM panel.

There are two things that you can do to make this easier.

Modify your assembler to generate a listing file.

A listing file shows all the ASM source lines, including comments, as well as the ROM addresses and the values of the labels and the instructions. For example, here is a snippet of a listing file generated by an assembler written by Mark Armbrust:

```
      20    16      @i      // i -= 1
      21  FC88      M=M-1

      22  FC10      D=M     // if i > 0
      23     6      @LOOP
      24  E301      D;JGT   //      goto LOOP

      25        (STOP)
      25    25      @STOP
      26  EA87      0;JMP

   Data Symbols

      16 D  i

   Code Symbols

       6 C  LOOP
      17 C  SKIP
      25 C  STOP
```

For the Nand2Tetris environment, it is most useful to list the ROM addresses and A-instruction values in decimal. In the above snippet, the C-instruction values are listed in hexadecimal.

The list file is generated during pass 2 of the Assembler, parallel to generating the .hack file. To make it easier to handle blank and comment only lines, Mark has Parser.commandType() return NO_COMMAND for source lines with no command. Mark also added Parser.sourceLine() that returns the unmodified source line.

Have your VM Translator write the VM source lines as comments in the ASM output.

For example:

```
    // label LOOP
(Sys.init$LOOP)
    // goto LOOP
@Sys.init$LOOP
0;JMP
    //
    // // Sys.main()
    //
    // // Sets locals 1, 2 and 3, leaving locals 0 and 4 unchanged to test
    // // default local initialization to 0.  (RAM set to -1 by test setup.)
    // // Calls Main.add12(123) and stores return value (135) in temp 0.
    // // Returns local 0 + local 1 + local 2 + local 3 + local 4 (456) to confirm
    // // that locals were not mangled by function call.
    //
    // function Sys.main 5
(Sys.main)
@5
D=-A
($3)
@SP
```

Note that comments in the VM source become double comments. Looking ahead, in Project 11 you will be asked to write a compiler for the Jack language. If your compiler will write the Jack source lines as comments in the generated VM files, this convention will be quite useful.