Extending Hack VM

John Lapinskas, University of Bristol

Our goals for this week

High-level language	
Intermediate representation	Software
Assembly	
Instruction set architecture	
Computer microarchitecture	
Components	
Gates	Hardware
Transistors	
Physics	

This week, we'll be extending Hack VM to include:

- Function calls.

Our goals for this week

Layer	
High-level language	Software
Intermediate representation	Software
Assembly	Software
Instruction set architecture	Software
Computer microarchitecture	Hardware
Components	Hardware
Gates	Hardware
Transistors	Hardware
Physics	Hardware

This week, we'll be extending Hack VM to include:

- Function calls.
- Proper compile-time memory allocation. (Solved at the same time!)

Our goals for this week

Layer	
High-level language	Software
Intermediate representation	
Assembly	
Instruction set architecture	
Computer microarchitecture	Hardware
Components	
Gates	
Transistors	
Physics	

This week, we'll be extending Hack VM to include:

- Function calls.
- Proper compile-time memory allocation. (Solved at the same time!)
- Multi-file compilation support for libraries. (Easy by comparison.)

Our goals for this week

High-level language	
Intermediate representation	Software
Assembly	
Instruction set architecture	
Computer microarchitecture	
Components	
Gates	Hardware
Transistors	
Physics	

This week, we'll be extending Hack VM to include:

- Function calls.
- Proper compile-time memory allocation. (Solved at the same time!)
- Multi-file compilation support for libraries. (Easy by comparison.)

We'll also talk about run-time memory allocation (i.e. `malloc`).

Our goals for this week

Layer	
High-level language	
Intermediate representation	Software
Assembly	
Instruction set architecture	
Computer microarchitecture	
Components	
Gates	Hardware
Transistors	
Physics	

This week, we'll be extending Hack VM to include:

- Function calls.
- Proper compile-time memory allocation. (Solved at the same time!)
- Multi-file compilation support for libraries. (Easy by comparison.)

We'll also talk about run-time memory allocation (i.e. `malloc`).

Next week, we'll be moving on to our high-level language — **Jack**.

How functions should behave: A case study

```c
int fibonacci(int n) {
    int x, y = 0;
    static int times_called = 0;
    static int layers_deep = 0;
    layers_deep++;

    times_called++;

    if (n == 0 || n == 1) {
        x = (n == 0) ? 0 : 1;
        layers_deep--;
        return x;
    }

    x = fibonacci(n-1);
    y = fibonacci(n-2);

    layers_deep--;
    return x+y;
}
```

Recall the (bad!) recursive algorithm from Programming in C to compute the Fibonacci sequence. The sequence is:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

So the algorithm is:

$$\texttt{fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \texttt{fib}(n-1) + \texttt{fib}(n-2) & \text{if } n \geq 2. \end{cases}$$

Here we have an implementation with some static variables to keep track of how many times the function has been called and how many calls deep we are into the recursion.

Summary of desired behaviour

On each call to `fibonacci`:

- Program flow jumps to the start of the function.
- The local variables `x` and `y` are cleared.
- The argument variable `n` is set by the call.
- The static variables `times_called` and `layers_deep` are unchanged.

On function return:

- Program flow returns to the line after the original function call.
- The local variables `x` and `y` return to their old values.
- The argument variable `n` returns to its old value.
- The static variables `times_called` and `layers_deep` are unchanged.

All of this must be robust for an arbitrary number of function calls within function calls (memory permitting), including recursive calls.

[See video for a demonstration in CLion with fibonacci-illustration.c.]

Compiling multiple files

If the VM translator is given a folder, it should operate as follows:

- All `.vm` files in the folder should be translated into one assembly file.

Compiling multiple files

If the VM translator is given a folder, it should operate as follows:

- All `.vm` files in the folder should be translated into one assembly file.
- All VM code should occur within functions.

Compiling multiple files

If the VM translator is given a folder, it should operate as follows:

- All `.vm` files in the folder should be translated into one assembly file.
- All VM code should occur within functions.
- The compiled code should start by setting `SP` to 256 as normal, then call a function **`Sys.init`** in a VM file called **`Sys.vm`**.
  - This is analogous to `main` in C.
  - `Sys.init` will be provided for you in test code.
  - This means the order in which the files are translated won't matter.

Compiling multiple files

If the VM translator is given a folder, it should operate as follows:

- All `.vm` files in the folder should be translated into one assembly file.
- All VM code should occur within functions.
- The compiled code should start by setting `SP` to 256 as normal, then call a function **`Sys.init`** in a VM file called **`Sys.vm`**.
  - This is analogous to `main` in C.
  - `Sys.init` will be provided for you in test code.
  - This means the order in which the files are translated won't matter.
- All functions in the file `abc.vm` (say) must have names starting with "`abc.`", e.g. `abc.print` or `abc.crash`.
  - This prevents name clashes between files, and our Jack compiler will enforce it by compiling a Jack function named `xyz` in a file named `abc` to a VM function named `abc.xyz`.

Compiling multiple files

If the VM translator is given a folder, it should operate as follows:

- All `.vm` files in the folder should be translated into one assembly file.
- All VM code should occur within functions.
- The compiled code should start by setting `SP` to 256 as normal, then call a function **`Sys.init`** in a VM file called **`Sys.vm`**.
  - This is analogous to `main` in C.
  - `Sys.init` will be provided for you in test code.
  - This means the order in which the files are translated won't matter.
- All functions in the file `abc.vm` (say) must have names starting with "`abc.`", e.g. `abc.print` or `abc.crash`.
  - This prevents name clashes between files, and our Jack compiler will enforce it by compiling a Jack function named `xyz` in a file named `abc` to a VM function named `abc.xyz`.
- If two instances of the same `static` address or `label` occur in different files, they should compile to different addresses.
  - Your VM translator already does this!

Jack and the "operating system"

Jack will come with what nand2tetris optimistically calls an "operating system".

Really it's a collection of eight standard libraries written in Jack! (In fairness, this is what an OS *is* at its core — you could build a single-process OS like DOS on this foundation by extending the `Sys` library, viewing processes as function calls.)

Jack and the "operating system"

Jack will come with what nand2tetris optimistically calls an "operating system".

Really it's a collection of eight standard libraries written in Jack! (In fairness, this is what an OS *is* at its core — you could build a single-process OS like DOS on this foundation by extending the `Sys` library, viewing processes as function calls.)

- `Sys.vm` provides `Sys.init` and functions to halt, crash, or wait a certain number of milliseconds.
- `Memory.vm` provides functions for memory allocation (see later).
- `Array.vm` provides functions for an array data type.
- `String.vm` provides functions for a string data type.
- `Keyboard.vm` and `Screen.vm` provide functions for direct input and output.
- `Output.vm` provides functions for displaying and editing text.
- `Math.vm` provides functions for multiplication, division, minimum, maximum, and square root.

You will have access to Hack VM versions of these next week when writing the Jack compiler. The details can be found in Nisan and Schocken appendix 6.

None of them are examinable!

Bootstrapping

In the nand2tetris course, these libraries are actually all written in Jack.
Isn't using Jack code to write a Jack compiler "cheating"?

Bootstrapping

In the nand2tetris course, these libraries are actually all written in Jack. Isn't using Jack code to write a Jack compiler "cheating"? No!

Making a compiler that can compile itself is called **bootstrapping**, and the normal approach is:

- Write a placeholder Jack-to-Hack Compiler A, either:
  - in Hack VM/assembly directly, or
  - in another system entirely, e.g. C on x86-64 (**cross-compiling**).

Bootstrapping

In the nand2tetris course, these libraries are actually all written in Jack.
Isn't using Jack code to write a Jack compiler "cheating"? No!

Making a compiler that can compile itself is called **bootstrapping**, and the normal approach is:

- Write a placeholder Jack-to-Hack Compiler A, either:
  - in Hack VM/assembly directly, or
  - in another system entirely, e.g. C on x86-64 (**cross-compiling**).
- Write a better-quality Jack-to-Hack Compiler B in Jack itself.

Bootstrapping

In the nand2tetris course, these libraries are actually all written in Jack.
Isn't using Jack code to write a Jack compiler "cheating"? No!

Making a compiler that can compile itself is called **bootstrapping**, and the normal approach is:

- Write a placeholder Jack-to-Hack Compiler A, either:
  - in Hack VM/assembly directly, or
  - in another system entirely, e.g. C on x86-64 (**cross-compiling**).
- Write a better-quality Jack-to-Hack Compiler B in Jack itself.
- Use Compiler A to compile Compiler B to Hack machine code.

Bootstrapping

In the nand2tetris course, these libraries are actually all written in Jack. Isn't using Jack code to write a Jack compiler "cheating"? No!

Making a compiler that can compile itself is called **bootstrapping**, and the normal approach is:

- Write a placeholder Jack-to-Hack Compiler A, either:
  - in Hack VM/assembly directly, or
  - in another system entirely, e.g. C on x86-64 (**cross-compiling**).
- Write a better-quality Jack-to-Hack Compiler B in Jack itself.
- Use Compiler A to compile Compiler B to Hack machine code.
- Use Compiler B to re-compile Compiler B to Hack machine code.

Bootstrapping

In the nand2tetris course, these libraries are actually all written in Jack. Isn't using Jack code to write a Jack compiler "cheating"? No!

Making a compiler that can compile itself is called **bootstrapping**, and the normal approach is:

- Write a placeholder Jack-to-Hack Compiler A, either:
  - in Hack VM/assembly directly, or
  - in another system entirely, e.g. C on x86-64 (**cross-compiling**).
- Write a better-quality Jack-to-Hack Compiler B in Jack itself.
- Use Compiler A to compile Compiler B to Hack machine code.
- Use Compiler B to re-compile Compiler B to Hack machine code.
- Throw Compiler A away and use Compiler B from now on.

Bootstrapping

In the nand2tetris course, these libraries are actually all written in Jack.
Isn't using Jack code to write a Jack compiler "cheating"? No!

Making a compiler that can compile itself is called **bootstrapping**, and the normal approach is:

- Write a placeholder Jack-to-Hack Compiler A, either:
  - in Hack VM/assembly directly, or
  - in another system entirely, e.g. C on x86-64 (**cross-compiling**).
- Write a better-quality Jack-to-Hack Compiler B in Jack itself.
- Use Compiler A to compile Compiler B to Hack machine code.
- Use Compiler B to re-compile Compiler B to Hack machine code.
- Throw Compiler A away and use Compiler B from now on.

Even not cross-compiling, the goal is to write as little low-level code as possible.

Bootstrapping

In the nand2tetris course, these libraries are actually all written in Jack. Isn't using Jack code to write a Jack compiler "cheating"? No!

Making a compiler that can compile itself is called **bootstrapping**, and the normal approach is:

- Write a placeholder Jack-to-Hack Compiler A, either:
  - in Hack VM/assembly directly, or
  - in another system entirely, e.g. C on x86-64 (**cross-compiling**).
- Write a better-quality Jack-to-Hack Compiler B in Jack itself.
- Use Compiler A to compile Compiler B to Hack machine code.
- Use Compiler B to re-compile Compiler B to Hack machine code.
- Throw Compiler A away and use Compiler B from now on.

Even not cross-compiling, the goal is to write as little low-level code as possible.

The "OS" functions are mostly straightforward, and there are no new architecture ideas except for `Memory.vm` (which we'll talk about later this week).

That said, if you want practice with assembly/VM, they make good exercises!