The Hack VM II: Branching and memory
COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

Labels and gotos in Hack VM

How do we handle loops and conditionals in the Hack VM?

Labels and gotos in Hack VM

How do we handle loops and conditionals in the Hack VM?

The same way as in assembly — with jumps, which we now call gotos as they need no longer correspond to only a single machine code instruction.

The syntax is:

- `label LABEL_NAME` declares a label at that point of the code.
- `goto LABEL_NAME` jumps to that label from anywhere in the code.[1]
- `if-goto LABEL_NAME` pops the stack and executes `goto LABEL_NAME` if the result is non-zero (i.e. if it is not `false`).

[1] Strictly speaking, the `goto` and label must be in the same function — see next week.

Labels and gotos in Hack VM

How do we handle loops and conditionals in the Hack VM?

The same way as in assembly — with jumps, which we now call gotos as they need no longer correspond to only a single machine code instruction.

The syntax is:

- `label LABEL_NAME` declares a label at that point of the code.
- `goto LABEL_NAME` jumps to that label from anywhere in the code.[1]
- `if-goto LABEL_NAME` pops the stack and executes `goto LABEL_NAME` if the result is non-zero (i.e. if it is not `false`).

We should use `if-goto` in the same way that we would use `D;JNE` in assembly. The differences are:

- The value we compare to zero is the top of the stack instead of D.
- We have proper logical operators `gt`, `eq`, `lt`, `and`, `or` and `not` built into the language to replace the various jump conditions.

[1] Strictly speaking, the `goto` and label must be in the same function — see next week.

pointer and this

With `local`, we can only push from or pop to memory addresses known before runtime. This can present problems.

Say we have an array stored in memory and want to access the `local 0`'th element of it. But e.g. `push local (local 0)` is not valid VM code!

pointer and this

With `local`, we can only push from or pop to memory addresses known before runtime. This can present problems.

Say we have an array stored in memory and want to access the `local 0`'th element of it. But e.g. `push local (local 0)` is not valid VM code!

Instead, we can use **`pointer`** and **`this`**, two special memory segments. The map from `this` addresses to physical RAM is not fixed in advance, but determined at run-time. We are guaranteed that:

`this 0` maps to RAM[`pointer 0`],
`this 1` maps to RAM[(`pointer 0`) + 1],
`this 2` maps to RAM[(`pointer 0`) + 2]

and so on.

Implementing arrays in Hack VM

We are guaranteed that `this` i maps to RAM[(`pointer 0`) + i] for all i.

We will still need to decide in advance which segments of physical memory will hold our array, just like with assembly. But if we have decided it will be stored in RAM[0x0800]–RAM[0x08FF] (say), then e.g.:

```
push constant 2048
push local 0
add             // Stack now contains (local 0) + 0x0800
pop pointer 0 // this i now maps to (local 0) + 0x0800 + i
push this 0
```

will load the `local 0`'th element of the array onto the stack (counting from 0).

Implementing arrays in Hack VM

We are guaranteed that `this` i maps to RAM[(`pointer 0`) + i] for all i.

We will still need to decide in advance which segments of physical memory will hold our array, just like with assembly. But if we have decided it will be stored in RAM[0x0800]–RAM[0x08FF] (say), then e.g.:

```
push constant 2048
push local 0
add             // Stack now contains (local 0) + 0x0800
pop pointer 0 // this i now maps to (local 0) + 0x0800 + i
push this 0
```

will load the `local 0`'th element of the array onto the stack (counting from 0).

Of course, if `local 0` is 256 or more then we'll run into problems!

Our high-level language will handle this memory allocation automatically, but for now we do it manually. Life is suffering.

Implementing I/O in Hack VM

For I/O, we can use the same trick to access RAM[0x4000–0x5FFF] for the screen and RAM[0x6000] for the keyboard... right?

Implementing I/O in Hack VM

For I/O, we can use the same trick to access RAM[0x4000–0x5FFF] for the screen and RAM[0x6000] for the keyboard... right?

Yes, but annoyingly we can't use `this` for it. We'll see later that `this` has a special role in compiling abstract data types like C's structs, so it can't map to arbitrary segments of memory. This will make sense in week 11!

For now, you only need to know that the `this` segment can only be used to access RAM[0x0800–0x3FFF]. For anything outside that, we must instead use the **`that`** memory segment.

Implementing I/O in Hack VM

For I/O, we can use the same trick to access RAM[0x4000–0x5FFF] for the screen and RAM[0x6000] for the keyboard... right?

Yes, but annoyingly we can't use `this` for it. We'll see later that `this` has a special role in compiling abstract data types like C's structs, so it can't map to arbitrary segments of memory. This will make sense in week 11!

For now, you only need to know that the `this` segment can only be used to access RAM[0x0800–0x3FFF]. For anything outside that, we must instead use the **`that`** memory segment.

`that` behaves almost exactly like `this`. The only differences are:

- The map is from `that 0` to RAM[`pointer 1`] not RAM[`pointer 0`].
- `that` can be used to access any address of physical RAM, not just RAM[0x0800–0x3FFF].

We'll discuss memory mapping in more detail next video.

The eight virtual memory segments

We've seen `local`, `constant`, `this`, `that` and `pointer`. What are the other three virtual memory segments and why do they exist?

The eight virtual memory segments

We've seen `local`, `constant`, `this`, `that` and `pointer`. What are the other three virtual memory segments and why do they exist?

- **`static`** and **`argument`** will behave differently to `local` in function calls, which we discuss next week. More detail then, but in short:

The eight virtual memory segments

We've seen `local`, `constant`, `this`, `that` and `pointer`. What are the other three virtual memory segments and why do they exist?

- **`static`** and **`argument`** will behave differently to `local` in function calls, which we discuss next week. More detail then, but in short:
  - At the start of a function call, `local` will be empty. Its contents will be discarded on a function return.

The eight virtual memory segments

We've seen `local`, `constant`, `this`, `that` and `pointer`. What are the other three virtual memory segments and why do they exist?

- **`static`** and **`argument`** will behave differently to `local` in function calls, which we discuss next week. More detail then, but in short:
  - At the start of a function call, `local` will be empty. Its contents will be discarded on a function return.
  - At the start of a function call, `argument` will hold the arguments of that call. It cannot be written to.

The eight virtual memory segments

We've seen `local`, `constant`, `this`, `that` and `pointer`. What are the other three virtual memory segments and why do they exist?

- **`static`** and **`argument`** will behave differently to `local` in function calls, which we discuss next week. More detail then, but in short:
  - At the start of a function call, `local` will be empty. Its contents will be discarded on a function return.
  - At the start of a function call, `argument` will hold the arguments of that call. It cannot be written to.
  - The contents of `static` persist between function calls. (It will be used later for static and global variables in our high-level language.)

The eight virtual memory segments

We've seen `local`, `constant`, `this`, `that` and `pointer`. What are the other three virtual memory segments and why do they exist?

- **`static`** and **`argument`** will behave differently to `local` in function calls, which we discuss next week. More detail then, but in short:
  - At the start of a function call, `local` will be empty. Its contents will be discarded on a function return.
  - At the start of a function call, `argument` will hold the arguments of that call. It cannot be written to.
  - The contents of `static` persist between function calls. (It will be used later for static and global variables in our high-level language.)
- **`temp`** behaves exactly like `local`, but is mapped to a much smaller area of memory. It's intended as "working space" for use by a compiler from a high-level language for compiling an individual instruction without needing to disrupt the contents of `local`.

Putting it all together: fill.asm as VM code

Recall our assembly program fill.asm, which filled every pixel of the screen black. While any key was held, the screen would instead be filled white.

We implement the same program in Hack VM as fill.vm, for comparison.

[See video for live coding and demonstration.]