Implementing the Hack VM translator

John Lapinskas, University of Bristol

Tokens for the Hack VM

- **Keywords**:
  - ‘push’, ‘pop’, ‘add’, ‘sub’, ‘neg’, ‘and’, ‘or’, ‘not’, ‘eq’, ‘gt’, ‘lt’,
  - ‘local’, ‘constant’, ‘this’, ‘that’, ‘pointer’, ‘argument’, ‘static’, ‘temp’,
  - ‘label’, ‘goto’, ‘if-goto’,
  - ‘function’, ‘call’, ‘return’ (covered next week!)
- **Integer literals**: Any base-10 integer in the range $0 \ldots 32767$.
- **Identifiers**: Any string containing no whitespace that's not a keyword and starts with a letter.
- **Newlines**.

There's nothing new here — you already know how lexers work, so we've written this part of the compiler for you.

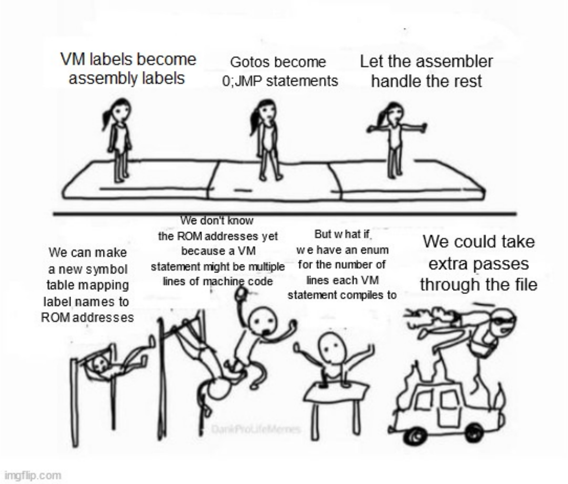
With the assembler, we built a symbol table of labels as part of lexing. Here, we don't need a symbol table of labels unless we're trying for good error handling (which we're not). Can you see why?

Implementing labels and gotos

In Hack assembly, we read labels into a symbol table during lexing. Here...

Implementing labels and gotos

In Hack assembly, we read labels into a symbol table during lexing. Here...


VM labels become assembly labels
Gotos become 0;JMP statements
Let the assembler handle the rest
We can make a new symbol table mapping label names to ROM addresses
We don't know the ROM addresses yet because a VM statement might be multiple lines of machine code
But what if we have an enum for the number of lines each VM statement compiles to
We could take extra passes through the file
DankProLifeMemes
imgflip.com

Source: Generated with imgflip (here)

I prefer the top option myself, but you do you.

A grammar for the Hack VM

There are many possible grammars for Hack VM. Here's one.

⟨instruction⟩ ::= (‘push’, ⟨data⟩ | ‘pop’, ⟨data⟩ |
‘add’ | ‘sub’ | ‘neg’ | ‘and’ | ‘or’ | ‘not’ | ‘eq’ | ‘gt’ | ‘lt’ |
‘label’, identifier | ‘goto’, identifier | ‘if-goto’, identifier |
‘function’, identifier, integerLiteral |
‘call’, identifier, integerLiteral | ‘return’), newline
⟨data⟩ ::= (‘local’ | ‘constant’ | ‘this’ | ‘that’ | ‘pointer’ |
‘argument’ | ‘static’ | ‘temp’), integerLiteral

This is actually far simpler than Hack assembly — it's an LL(1) grammar, and we can tell how to parse every ⟨instruction⟩ based on its first token.

The hard part is actually translating VM instructions into assembly!

You can work most of this out for yourselves, but we'll cover memory and the stack in detail.

Allocating memory

The general purpose of mapping virtual memory to physical memory (**allocating** memory) is the same as with `this` and `that`.

Say we want to map `local` to physical memory. Then:

- We choose an address in RAM, say 300.
- We map `local 0` to RAM[300].
- We map `local 1` to RAM[301].
- We map `local 2` to RAM[302].
- ...
- We map `local whatever` to RAM[$300 + \textit{whatever}$].

Allocating memory

The general purpose of mapping virtual memory to physical memory (**allocating** memory) is the same as with `this` and `that`.

Say we want to map `local` to physical memory. Then:

- We choose an address in RAM, say 300.
- We map `local 0` to RAM[300].
- We map `local 1` to RAM[301].
- We map `local 2` to RAM[302].
- $\cdots$
- We map `local whatever` to RAM[$300 + \textit{whatever}$].

We call 300 the **base address**, and the number after `local` the **offset**. Thus every address in `local` is mapped to the base address plus the offset.

For example, `pointer 0` is the base address of `this`, and `pointer 1` is the base address of `that`.

Hardware limitations on memory

The Hack VM has 64KB of memory attached to each of the eight virtual memory segments (in the form of 32,768 16-bit words) and a stack which can be infinitely tall. The Hack CPU supports 64KB of memory in total. Something has to give.[1]

So we give each memory segment a **length** and forbid access to any area of memory past that. For example, if we allocate `local` a segment of length 5, then we allow access to `local 0` through `local 4` only.

We are then free to take the address that would be `local 5` to be (for example) the base address of `argument`.

[1] This isn't unique to Hack VM — most models of computation will assume access to infinite storage. Turing machines have an infinite tape, C programs have no hard limit on how much you can `malloc` before `free`ing.

Hardware limitations on memory

The Hack VM has 64KB of memory attached to each of the eight virtual memory segments (in the form of 32,768 16-bit words) and a stack which can be infinitely tall. The Hack CPU supports 64KB of memory in total. Something has to give.[1]

So we give each memory segment a **length** and forbid access to any area of memory past that. For example, if we allocate `local` a segment of length 5, then we allow access to `local 0` through `local 4` only.

We are then free to take the address that would be `local 5` to be (for example) the base address of `argument`.

Modern systems enforce this carefully. If a process tries to access a value outside one of its allocated segments, then a hardware interrupt is generated — typically leading to a **segmentation fault** crash.

(If the memory segment is for the stack, then the error is a **stack overflow**.)

This requires hardware support Hack doesn't have, so we would have to do this (inefficiently) in software. For now, we just track segment lengths manually.

[1] This isn't unique to Hack VM — most models of computation will assume access to infinite storage. Turing machines have an infinite tape, C programs have no hard limit on how much you can `malloc` before `free`ing.

Example: Memory allocation in C

A `long long` in C is a 64-bit integer, one word of memory on a 64-bit computer.

Example: Memory allocation in C

A `long long` in C is a 64-bit integer, one word of memory on a 64-bit computer.

The C code `long long myArray[128];` allocates a segment of 128 words of memory and stores the base address in the `long long *` variable `myArray`.

Example: Memory allocation in C

A `long long` in C is a 64-bit integer, one word of memory on a 64-bit computer.

The C code `long long myArray[128];` allocates a segment of 128 words of memory and stores the base address in the `long long *` variable `myArray`.

The code `myArray[50]` then returns the `long long` stored at address `myArray` + 50, i.e. the 51st entry of the array.

Example: Memory allocation in C

A `long long` in C is a 64-bit integer, one word of memory on a 64-bit computer.

The C code `long long myArray[128];` allocates a segment of 128 words of memory and stores the base address in the `long long *` variable `myArray`.

The code `myArray[50]` then returns the `long long` stored at address $\texttt{myArray} + 50$, i.e. the 51st entry of the array.

The code `myArray[128]` attempts to access the `long long` stored at address $\texttt{myArray} + 128$. But this is outside the allocated segment (by one word), so we get a segmentation fault.

Example: Memory allocation in C

A `long long` in C is a 64-bit integer, one word of memory on a 64-bit computer.

The C code `long long myArray[128];` allocates a segment of 128 words of memory and stores the base address in the `long long *` variable `myArray`.

The code `myArray[50]` then returns the `long long` stored at address `myArray` + 50, i.e. the 51st entry of the array.

The code `myArray[128]` attempts to access the `long long` stored at address `myArray` + 128. But this is outside the allocated segment (by one word), so we get a segmentation fault.

Arrays of other types are handled analogously, but with allowances for different data sizes. E.g. `short *myArray = malloc(64*sizeof(short));` would allocate a segment of 16 words of memory and store four 16-bit `short`s in each word. `myArray[50]` would retrieve the third `short` stored at address `myArray` + 12. CPUs with modern ISAs can manipulate data stored this way very efficiently.

(Many IDEs for C, including e.g. CLion, support memory and disassembly views. So you can write some simple test code and investigate for yourself!)

Memory allocation in Hack

These details are all *specific to our implementation* of the Hack VM. They're not part of the Hack VM itself, but e.g. the Nand2tetris VM emulator assumes them.

The mystery keywords of Hack assembly (except `SP`) we haven't explained yet are all variables which hold the base addresses of memory segments:

- The base address of `local` is stored in $\mathrm{RAM}[1] = \texttt{LCL}$.

Memory allocation in Hack

These details are all *specific to our implementation* of the Hack VM. They're not part of the Hack VM itself, but e.g. the Nand2tetris VM emulator assumes them.

The mystery keywords of Hack assembly (except `SP`) we haven't explained yet are all variables which hold the base addresses of memory segments:

- The base address of `local` is stored in $\mathrm{RAM}[1] = \texttt{LCL}$.
- The base address of `argument` is stored in $\mathrm{RAM}[2] = \texttt{ARG}$.

Memory allocation in Hack

These details are all *specific to our implementation* of the Hack VM. They're not part of the Hack VM itself, but e.g. the Nand2tetris VM emulator assumes them.

The mystery keywords of Hack assembly (except `SP`) we haven't explained yet are all variables which hold the base addresses of memory segments:

- The base address of `local` is stored in $\mathrm{RAM}[1] =$ `LCL`.
- The base address of `argument` is stored in $\mathrm{RAM}[2] =$ `ARG`.
- `pointer` is allocated a fixed segment of length 2 and base address 3. So:
  - The base address of `this`, `pointer 0`, is stored in $\mathrm{RAM}[3] =$ `THIS`.
  - The base address of `that`, `pointer 1`, is stored in $\mathrm{RAM}[4] =$ `THAT`.

Memory allocation in Hack

These details are all *specific to our implementation* of the Hack VM. They're not part of the Hack VM itself, but e.g. the Nand2tetris VM emulator assumes them.

The mystery keywords of Hack assembly (except `SP`) we haven't explained yet are all variables which hold the base addresses of memory segments:

- The base address of `local` is stored in $\mathrm{RAM}[1] = \texttt{LCL}$.
- The base address of `argument` is stored in $\mathrm{RAM}[2] = \texttt{ARG}$.
- `pointer` is allocated a fixed segment of length 2 and base address 3. So:
  - The base address of `this`, `pointer 0`, is stored in $\mathrm{RAM}[3] = \texttt{THIS}$.
  - The base address of `that`, `pointer 1`, is stored in $\mathrm{RAM}[4] = \texttt{THAT}$.
- `temp` is allocated a fixed segment of length 8 and base address 5.

Memory allocation in Hack

These details are all *specific to our implementation* of the Hack VM. They're not part of the Hack VM itself, but e.g. the Nand2tetris VM emulator assumes them.

The mystery keywords of Hack assembly (except `SP`) we haven't explained yet are all variables which hold the base addresses of memory segments:

- The base address of `local` is stored in $\mathrm{RAM}[1] = \texttt{LCL}$.
- The base address of `argument` is stored in $\mathrm{RAM}[2] = \texttt{ARG}$.
- `pointer` is allocated a fixed segment of length 2 and base address 3. So:
  - The base address of `this`, `pointer 0`, is stored in $\mathrm{RAM}[3] = \texttt{THIS}$.
  - The base address of `that`, `pointer 1`, is stored in $\mathrm{RAM}[4] = \texttt{THAT}$.
- `temp` is allocated a fixed segment of length 8 and base address 5.
- `static` is allocated a fixed segment of length 240 and base address 16.
  - If compiling file Foo.vm, the address `static 5` should be mapped to the **Hack assembly variable** `Foo.5`. (Explanation next week!)

Memory allocation in Hack

These details are all *specific to our implementation* of the Hack VM. They're not part of the Hack VM itself, but e.g. the Nand2tetris VM emulator assumes them.

The mystery keywords of Hack assembly (except `SP`) we haven't explained yet are all variables which hold the base addresses of memory segments:

- The base address of `local` is stored in $\mathrm{RAM}[1] = \texttt{LCL}$.
- The base address of `argument` is stored in $\mathrm{RAM}[2] = \texttt{ARG}$.
- `pointer` is allocated a fixed segment of length 2 and base address 3. So:
    - The base address of `this`, `pointer 0`, is stored in $\mathrm{RAM}[3] = \texttt{THIS}$.
    - The base address of `that`, `pointer 1`, is stored in $\mathrm{RAM}[4] = \texttt{THAT}$.
- `temp` is allocated a fixed segment of length 8 and base address 5.
- `static` is allocated a fixed segment of length 240 and base address 16.
    - If compiling file Foo.vm, the address `static 5` should be mapped to the **Hack assembly variable** `Foo.5`. (Explanation next week!)
- `constant` does not appear in physical memory.

Memory allocation in Hack

These details are all *specific to our implementation* of the Hack VM. They're not part of the Hack VM itself, but e.g. the Nand2tetris VM emulator assumes them.

The mystery keywords of Hack assembly (except `SP`) we haven't explained yet are all variables which hold the base addresses of memory segments:

- The base address of `local` is stored in $\mathrm{RAM}[1] = \texttt{LCL}$.
- The base address of `argument` is stored in $\mathrm{RAM}[2] = \texttt{ARG}$.
- `pointer` is allocated a fixed segment of length 2 and base address 3. So:
  - The base address of `this`, `pointer 0`, is stored in $\mathrm{RAM}[3] = \texttt{THIS}$.
  - The base address of `that`, `pointer 1`, is stored in $\mathrm{RAM}[4] = \texttt{THAT}$.
- `temp` is allocated a fixed segment of length 8 and base address 5.
- `static` is allocated a fixed segment of length 240 and base address 16.
  - If compiling file Foo.vm, the address `static 5` should be mapped to the **Hack assembly variable** `Foo.5`. (Explanation next week!)
- `constant` does not appear in physical memory.

For now, we assume that RAM[1–4] are initialised to sensible values for us at the start of code execution. (The provided test scripts do this!)

Implementing the stack

We allocate the stack a fixed segment of memory as well, with length 1792 and base address 256. The address **one word past the top** of the stack is stored in $\mathrm{RAM}[0] = \texttt{SP}$. ($\texttt{SP}$ stands for **Stack Pointer**.)

Implementing the stack

We allocate the stack a fixed segment of memory as well, with length 1792 and base address 256. The address **one word past the top** of the stack is stored in $\mathrm{RAM}[0] = \texttt{SP}$. ($\texttt{SP}$ stands for **Stack Pointer**.)

When we push a new value x onto the stack, we write x to RAM[$\texttt{SP}$], then increment $\texttt{SP}$.

Implementing the stack

We allocate the stack a fixed segment of memory as well, with length 1792 and base address 256. The address **one word past the top** of the stack is stored in $\mathrm{RAM}[0] = \mathtt{SP}$. ($\mathtt{SP}$ stands for **Stack Pointer**.)

When we push a new value x onto the stack, we write x to RAM[$\mathtt{SP}$], then increment $\mathtt{SP}$.

When we pop a value from the stack into RAM[i], we decrement $\mathtt{SP}$, then copy RAM[$\mathtt{SP}$] to RAM[i].

Note that we don't need to zero RAM[$\mathtt{SP}$] here! Since the whole segment is reserved for the stack, it will be overwritten the next time it's accessed anyway.

Implementing the stack

We allocate the stack a fixed segment of memory as well, with length 1792 and base address 256. The address **one word past the top** of the stack is stored in $\mathrm{RAM}[0] = \texttt{SP}$. (SP stands for **Stack Pointer**.)

When we push a new value x onto the stack, we write x to RAM[SP], then increment SP.

When we pop a value from the stack into RAM[i], we decrement SP, then copy RAM[SP] to RAM[i].

Note that we don't need to zero RAM[SP] here! Since the whole segment is reserved for the stack, it will be overwritten the next time it's accessed anyway.

Technically we will also store `local` and `argument` as sub-segments of the stack. But don't worry about that for this week!

Implementing the stack

We allocate the stack a fixed segment of memory as well, with length 1792 and base address 256. The address **one word past the top** of the stack is stored in $\mathrm{RAM}[0] = \texttt{SP}$. (SP stands for **Stack Pointer**.)

When we push a new value *x* onto the stack, we write *x* to RAM[SP], then increment SP.

When we pop a value from the stack into RAM[*i*], we decrement SP, then copy RAM[SP] to RAM[*i*].

Note that we don't need to zero RAM[SP] here! Since the whole segment is reserved for the stack, it will be overwritten the next time it's accessed anyway.

Technically we will also store `local` and `argument` as sub-segments of the stack. But don't worry about that for this week!

[See video for a demonstration of memory handling in the VM emulator.]

Hack VM memory: A summary

Keyword	Addresses	Usage
SP	0	[Address of the topmost stack value] + 1.
LCL	1	Stores base address of local segment.
ARG	2	Stores base address of argument segment.
THIS	3	pointer 0 (i.e. base address of this segment).
THAT	4	pointer 1 (i.e. base address of that segment).
R5–R12	5–12	temp segment (max size 8).
R13–R15	13–15	Temporary variables for VM translator (if needed).
N/A	16–255	static segment (max size 240).
N/A	256–2047	Reserved for the stack, including local and argument segments (max size 1792 combined).
N/A	2048–16383	"Heap" memory for other purposes. Can be allocated to this or that segments.
SCREEN	16384–24575	Memory-mapped output to screen.
KBD	24576	Memory-mapped input from keyboard.

You'll be given this table as a reference in the exam!