Functions in Hack VM: Syntax and implementation

John Lapinskas, University of Bristol

Function syntax

Last video, we relied quite heavily on building symbol tables during the semantic analysis stage of compilation.

The Hack VM syntax is designed to completely avoid the need for them!

Function syntax

Last video, we relied quite heavily on building symbol tables during the semantic analysis stage of compilation.

The Hack VM syntax is designed to completely avoid the need for them!

- The syntax to define a function is `function name x`, where `name` is the function's name and `x` is the size of the function's `local` segment (generally the number of local variables used).

  Rather than using e.g. {}s or indentation, the function definition ends with either the next `function` command or the EOF. (So any code to be executed outside functions must be at the top of the file.)

Function syntax

Last video, we relied quite heavily on building symbol tables during the semantic analysis stage of compilation.

The Hack VM syntax is designed to completely avoid the need for them!

- The syntax to define a function is `function name x`, where `name` is the function's name and `x` is the size of the function's `local` segment (generally the number of local variables used).

  Rather than using e.g. {}s or indentation, the function definition ends with either the next `function` command or the EOF. (So any code to be executed outside functions must be at the top of the file.)
- The syntax to call a function is `call name x`, where `name` is the function's name and `x` is the number of arguments to use. This pops the top `x` values of the stack (for use as arguments), calls the function, and pushes the returned value onto the stack.

Function syntax

Last video, we relied quite heavily on building symbol tables during the semantic analysis stage of compilation.

The Hack VM syntax is designed to completely avoid the need for them!

- The syntax to define a function is `function name x`, where `name` is the function's name and `x` is the size of the function's `local` segment (generally the number of local variables used).

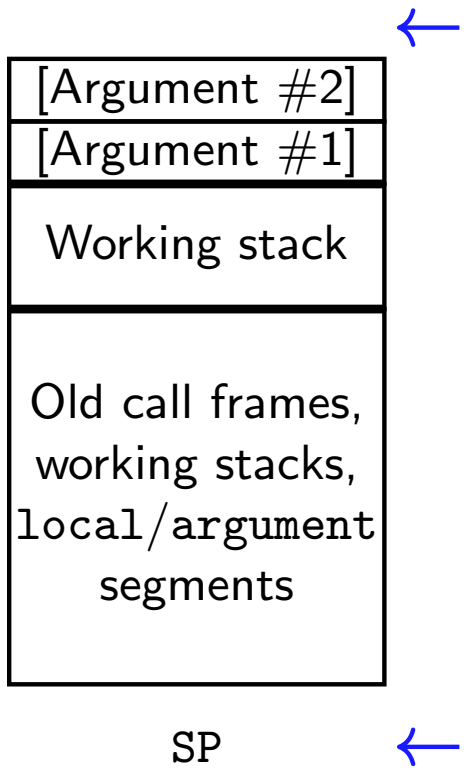
  Rather than using e.g. {}s or indentation, the function definition ends with either the next `function` command or the EOF. (So any code to be executed outside functions must be at the top of the file.)
- The syntax to call a function is `call name x`, where `name` is the function's name and `x` is the number of arguments to use. This pops the top `x` values of the stack (for use as arguments), calls the function, and pushes the returned value onto the stack.
- The syntax to return from a function is `return`, which returns the top value of the stack.

An example of function syntax in use

[See video for a demonstration with the VM simulator with sum.vm.]

Implementing function calls

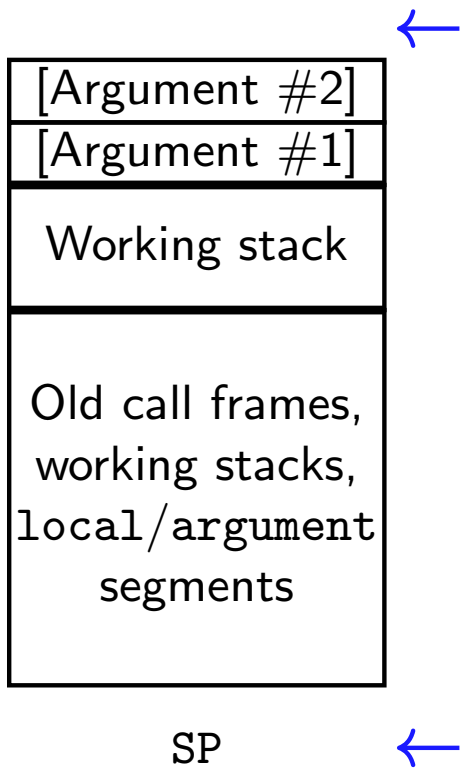
Say our VM translator sees the line `call myFunc 2`. On the right is one possible stack in execution of the VM code, as an example. The assembly code we generate must:



Implementing function calls

Say our VM translator sees the line `call myFunc 2`. On the right is one possible stack in execution of the VM code, as an example. The assembly code we generate must:

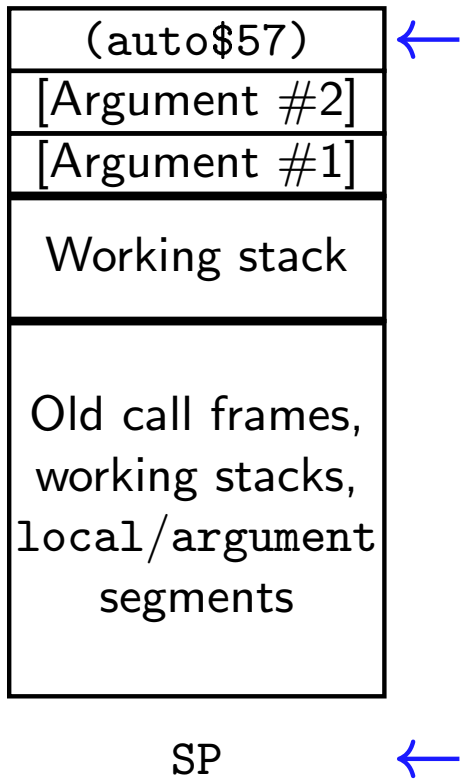
- End with a label, say `auto$57`, which we push onto the stack for the later return statement to jump to. This mustn't clash with any of our other labels in the final assembly file. (Note there's no need to update `SP` yet! It will be easier to do that later.)



Implementing function calls

Say our VM translator sees the line `call myFunc 2`. On the right is one possible stack in execution of the VM code, as an example. The assembly code we generate must:

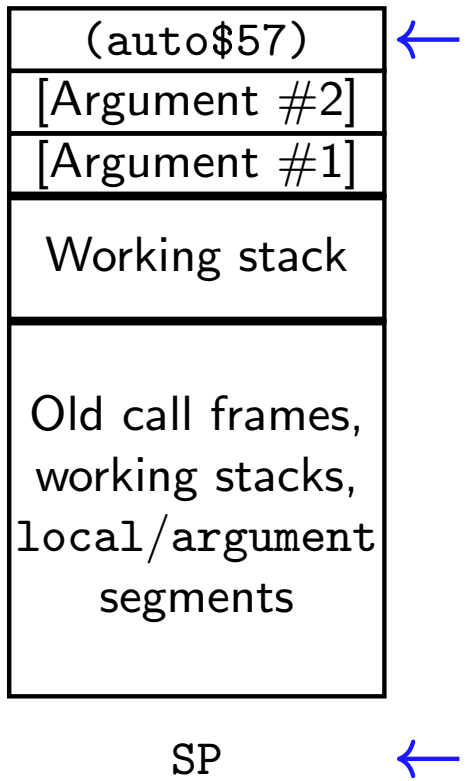
- End with a label, say `auto$57`, which we push onto the stack for the later return statement to jump to. This mustn't clash with any of our other labels in the final assembly file. (Note there's no need to update `SP` yet! It will be easier to do that later.)



Implementing function calls

Say our VM translator sees the line `call myFunc 2`. On the right is one possible stack in execution of the VM code, as an example. The assembly code we generate must:

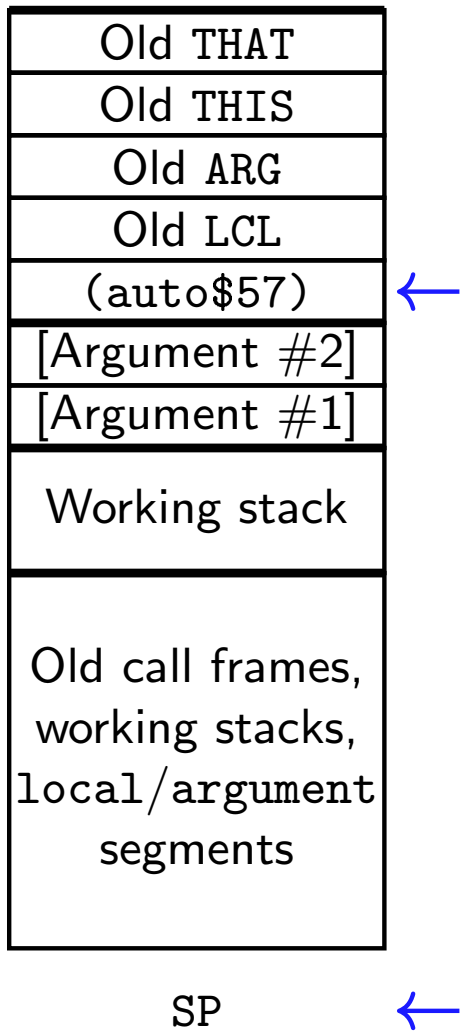
- End with a label, say `auto$57`, which we push onto the stack for the later return statement to jump to. This mustn't clash with any of our other labels in the final assembly file. (Note there's no need to update `SP` yet! It will be easier to do that later.)
- Push `LCL`, `ARG`, `THIS` and `THAT` onto the stack to preserve their current values.



Implementing function calls

Say our VM translator sees the line `call myFunc 2`. On the right is one possible stack in execution of the VM code, as an example. The assembly code we generate must:

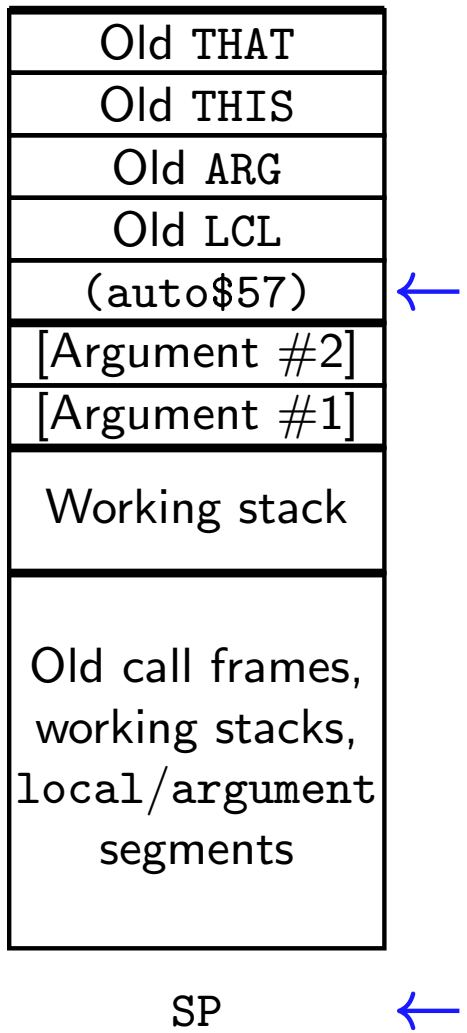
- End with a label, say `auto$57`, which we push onto the stack for the later return statement to jump to. This mustn't clash with any of our other labels in the final assembly file. (Note there's no need to update `SP` yet! It will be easier to do that later.)
- Push `LCL`, `ARG`, `THIS` and `THAT` onto the stack to preserve their current values.



Implementing function calls

Say our VM translator sees the line `call myFunc 2`. On the right is one possible stack in execution of the VM code, as an example. The assembly code we generate must:

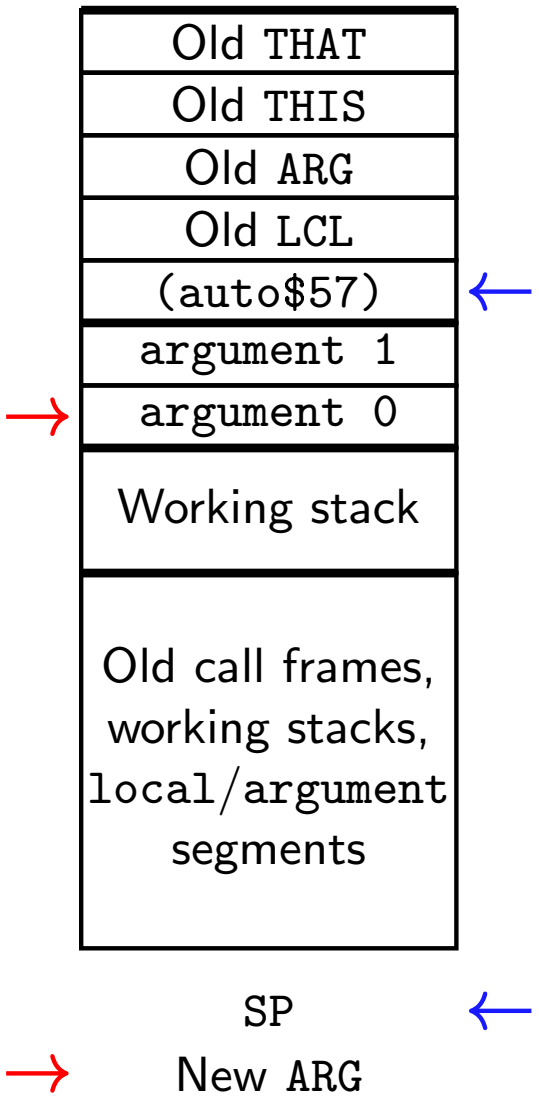
- End with a label, say `auto$57`, which we push onto the stack for the later return statement to jump to. This mustn't clash with any of our other labels in the final assembly file. (Note there's no need to update `SP` yet! It will be easier to do that later.)
- Push `LCL`, `ARG`, `THIS` and `THAT` onto the stack to preserve their current values.
- Set `ARG` to 2 values from the (old) top of the stack, allocating the arguments as the `argument` segment for the function call.



Implementing function calls

Say our VM translator sees the line `call myFunc 2`. On the right is one possible stack in execution of the VM code, as an example. The assembly code we generate must:

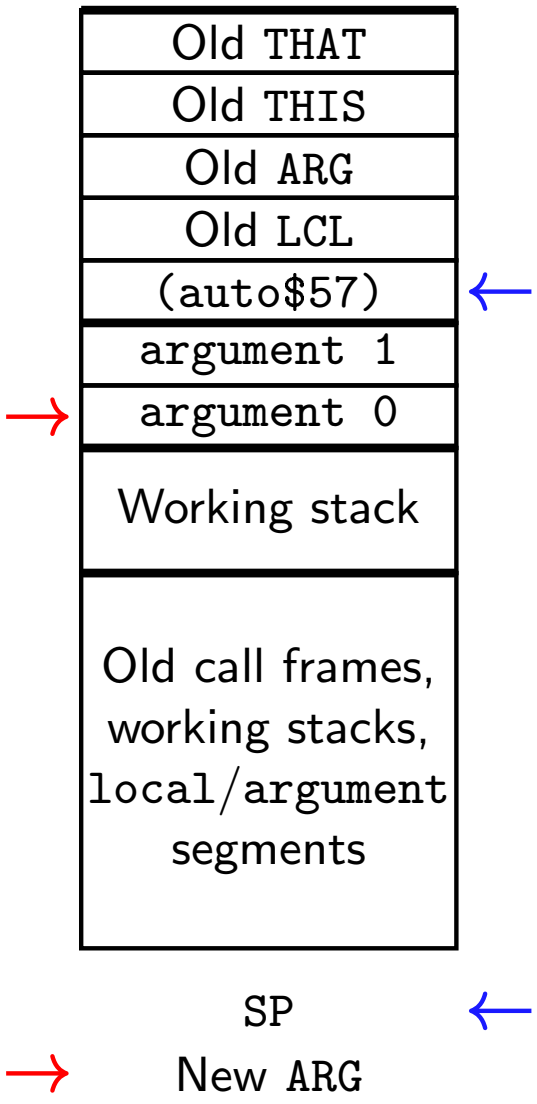
- End with a label, say `auto$57`, which we push onto the stack for the later return statement to jump to. This mustn't clash with any of our other labels in the final assembly file. (Note there's no need to update `SP` yet! It will be easier to do that later.)
- Push `LCL`, `ARG`, `THIS` and `THAT` onto the stack to preserve their current values.
- Set `ARG` to 2 values from the (old) top of the stack, allocating the arguments as the `argument` segment for the function call.



Implementing function calls

Say our VM translator sees the line `call myFunc 2`. On the right is one possible stack in execution of the VM code, as an example. The assembly code we generate must:

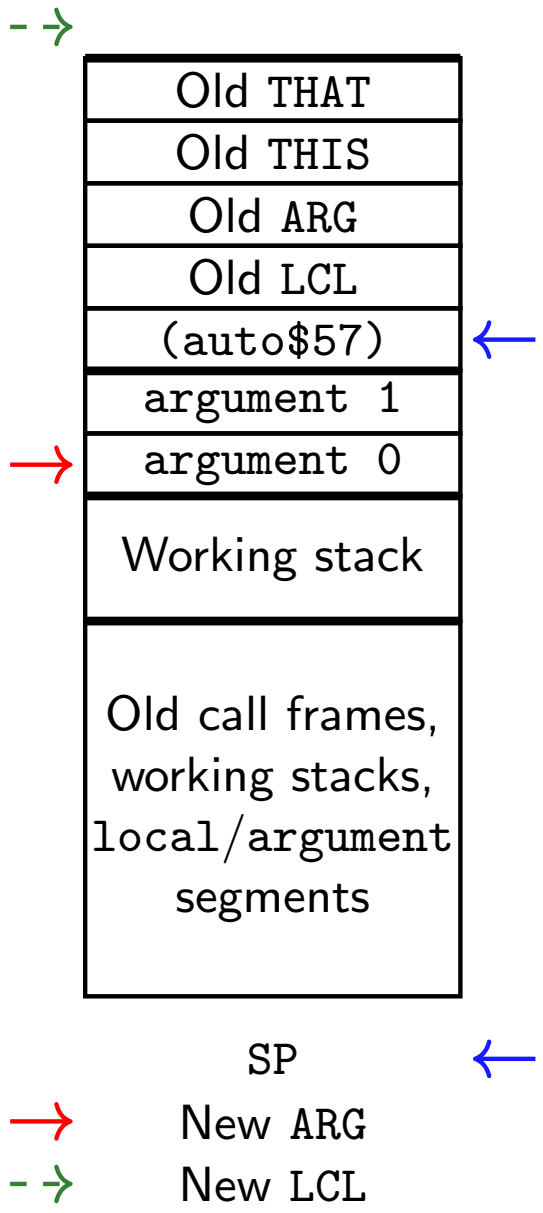
- End with a label, say `auto$57`, which we push onto the stack for the later return statement to jump to. This mustn't clash with any of our other labels in the final assembly file. (Note there's no need to update `SP` yet! It will be easier to do that later.)
- Push `LCL`, `ARG`, `THIS` and `THAT` onto the stack to preserve their current values.
- Set `ARG` to 2 values from the (old) top of the stack, allocating the arguments as the `argument` segment for the function call.
- Set `LCL` to the new top of the stack, which will be the start of the `local` segment for the function call. (The function definition will contain the length of `local`.)



Implementing function calls

Say our VM translator sees the line `call myFunc 2`. On the right is one possible stack in execution of the VM code, as an example. The assembly code we generate must:

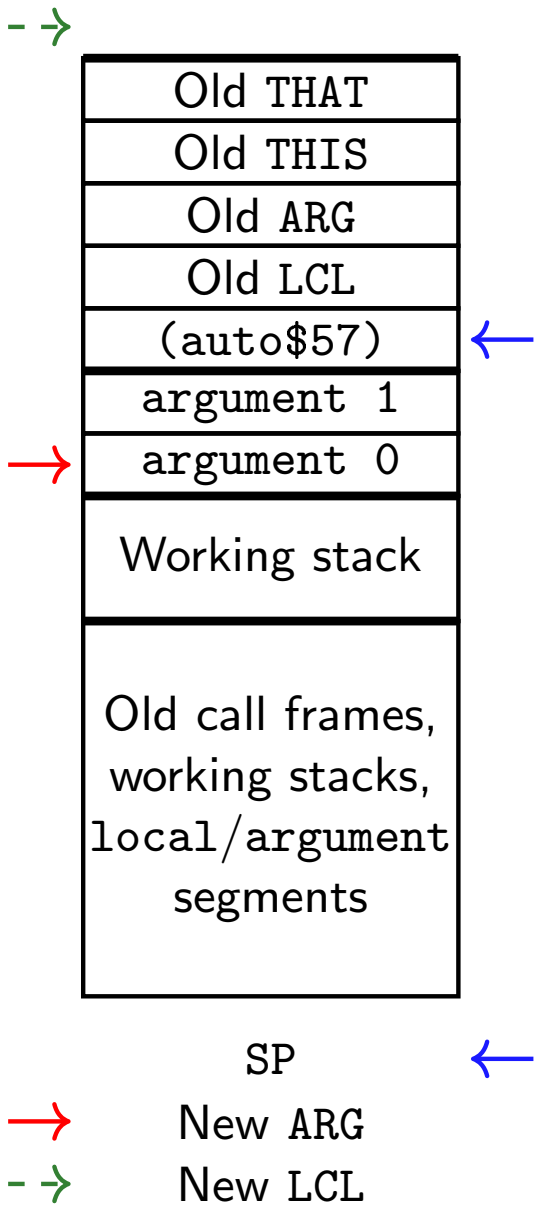
- End with a label, say `auto$57`, which we push onto the stack for the later return statement to jump to. This mustn't clash with any of our other labels in the final assembly file. (Note there's no need to update `SP` yet! It will be easier to do that later.)
- Push `LCL`, `ARG`, `THIS` and `THAT` onto the stack to preserve their current values.
- Set `ARG` to 2 values from the (old) top of the stack, allocating the arguments as the `argument` segment for the function call.
- Set `LCL` to the new top of the stack, which will be the start of the `local` segment for the function call. (The function definition will contain the length of `local`.)



Implementing function calls

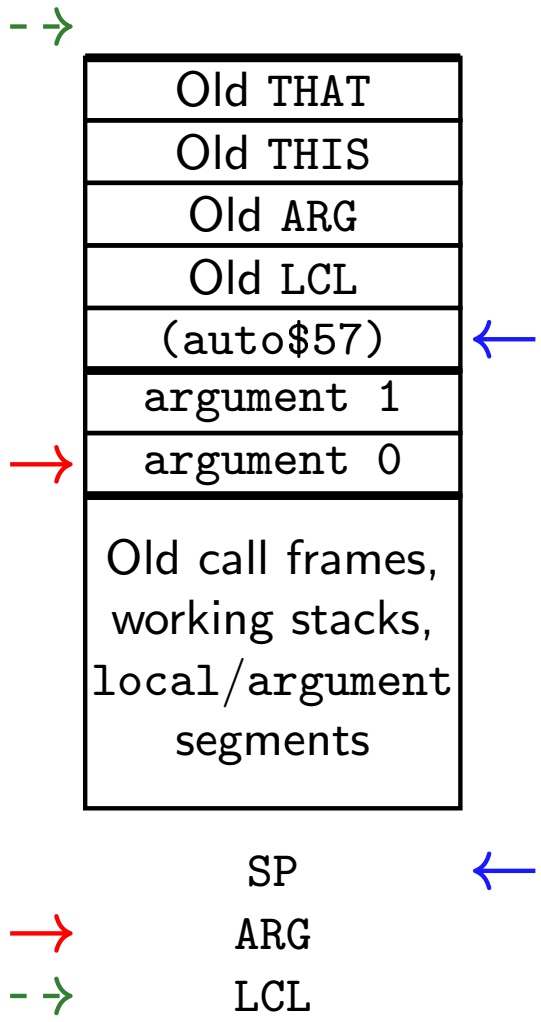
Say our VM translator sees the line `call myFunc 2`. On the right is one possible stack in execution of the VM code, as an example. The assembly code we generate must:

- End with a label, say `auto$57`, which we push onto the stack for the later return statement to jump to. This mustn't clash with any of our other labels in the final assembly file. (Note there's no need to update `SP` yet! It will be easier to do that later.)
- Push `LCL`, `ARG`, `THIS` and `THAT` onto the stack to preserve their current values.
- Set `ARG` to 2 values from the (old) top of the stack, allocating the arguments as the `argument` segment for the function call.
- Set `LCL` to the new top of the stack, which will be the start of the `local` segment for the function call. (The function definition will contain the length of `local`.)
- Jump to the function label (which we will generate from the function definition elsewhere in the VM code).



Implementing function definitions

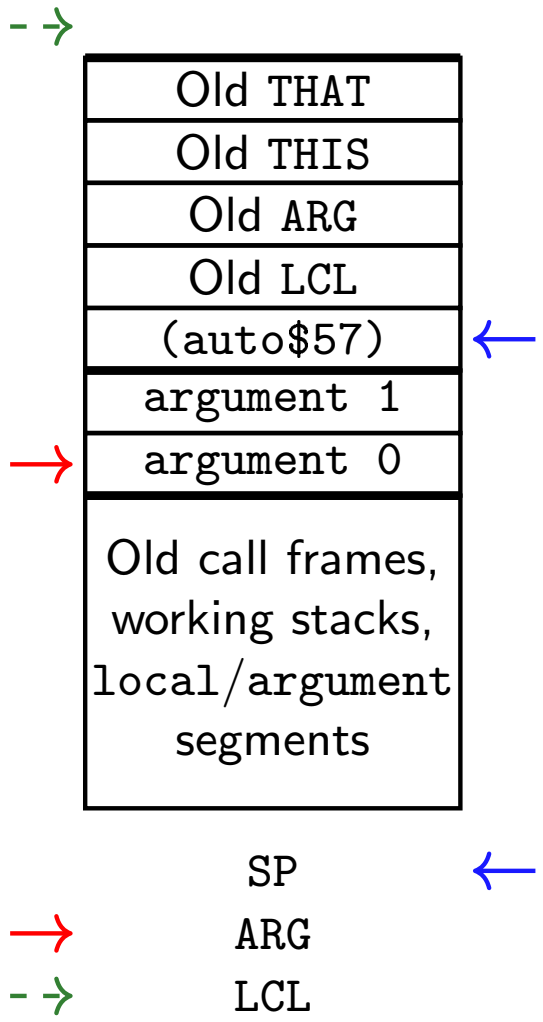
Say our VM translator sees the line `function myFunc 3`. On the right we continue the example from last slide. The assembly code we generate must:



Implementing function definitions

Say our VM translator sees the line `function myFunc 3`. On the right we continue the example from last slide. The assembly code we generate must:

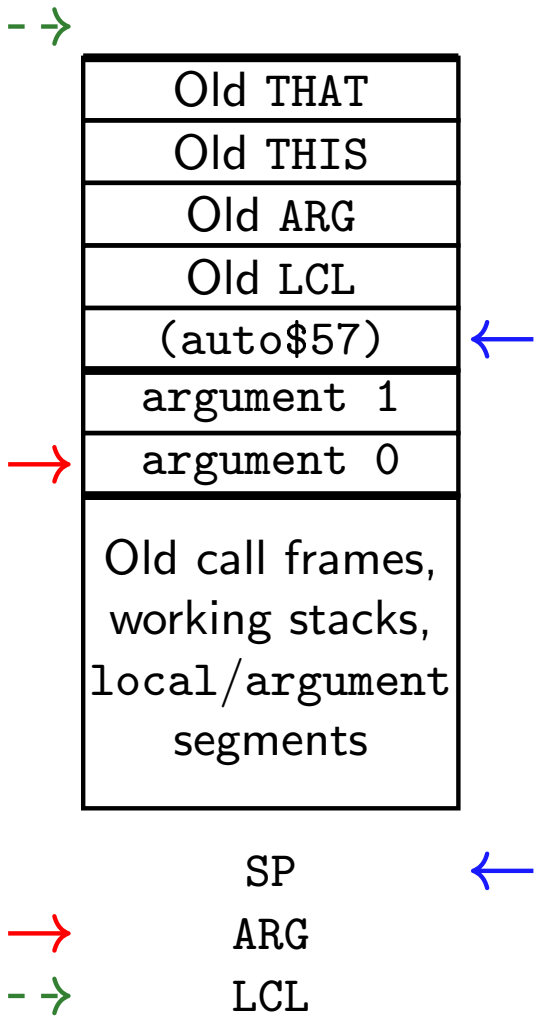
- Begin with a label, which we will jump to each time the function is called. To avoid the need for a symbol table, we should be able to get this label just from the function name (so we can derive it from the `call` statement).



Implementing function definitions

Say our VM translator sees the line `function myFunc 3`. On the right we continue the example from last slide. The assembly code we generate must:

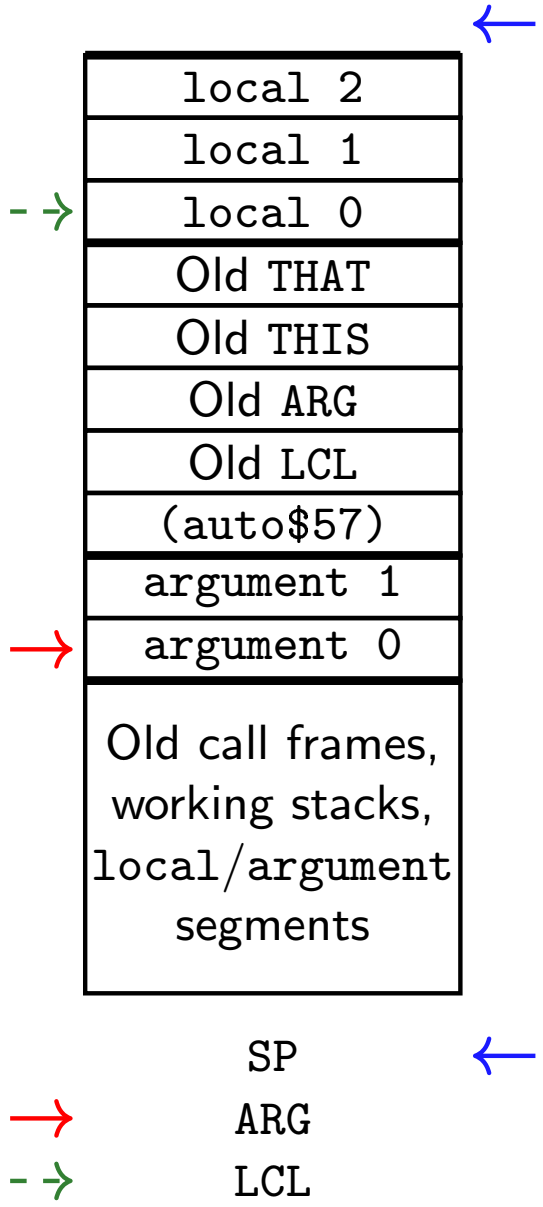
- Begin with a label, which we will jump to each time the function is called. To avoid the need for a symbol table, we should be able to get this label just from the function name (so we can derive it from the `call` statement).
- Set `SP` to `LCL` + 3, getting the length of the `local` segment from the `function` command.



Implementing function definitions

Say our VM translator sees the line `function myFunc 3`. On the right we continue the example from last slide. The assembly code we generate must:

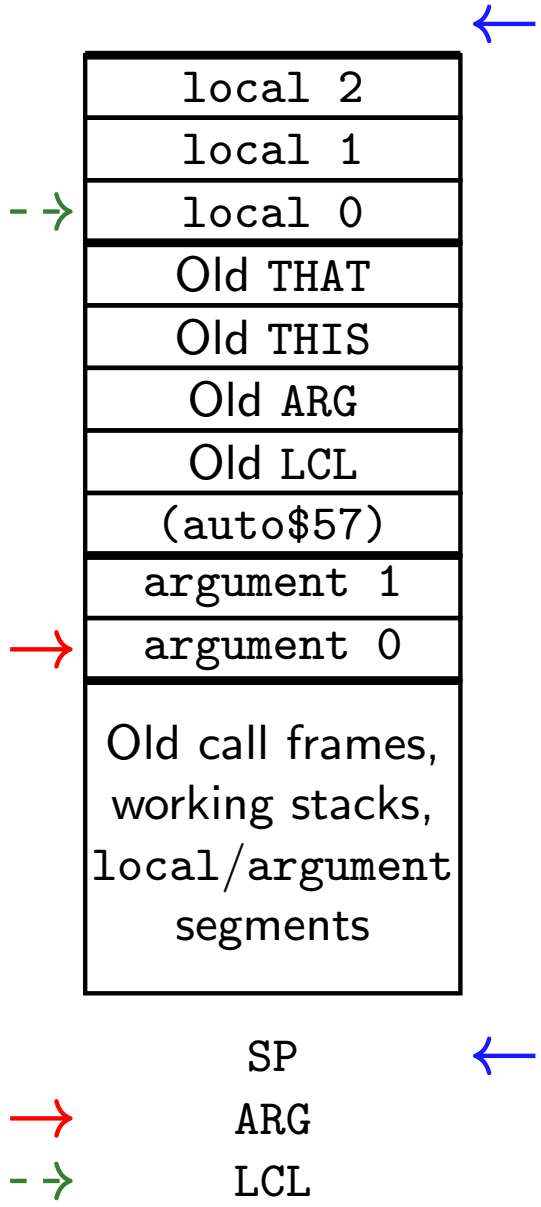
- Begin with a label, which we will jump to each time the function is called. To avoid the need for a symbol table, we should be able to get this label just from the function name (so we can derive it from the `call` statement).
- Set `SP` to `LCL` + 3, getting the length of the `local` segment from the `function` command.



Implementing function definitions

Say our VM translator sees the line `function myFunc 3`. On the right we continue the example from last slide. The assembly code we generate must:

- Begin with a label, which we will jump to each time the function is called. To avoid the need for a symbol table, we should be able to get this label just from the function name (so we can derive it from the `call` statement).
- Set SP to LCL + 3, getting the length of the `local` segment from the `function` command.
- Initialise `local 0`, `local 1` and `local 2` to zero.[a]

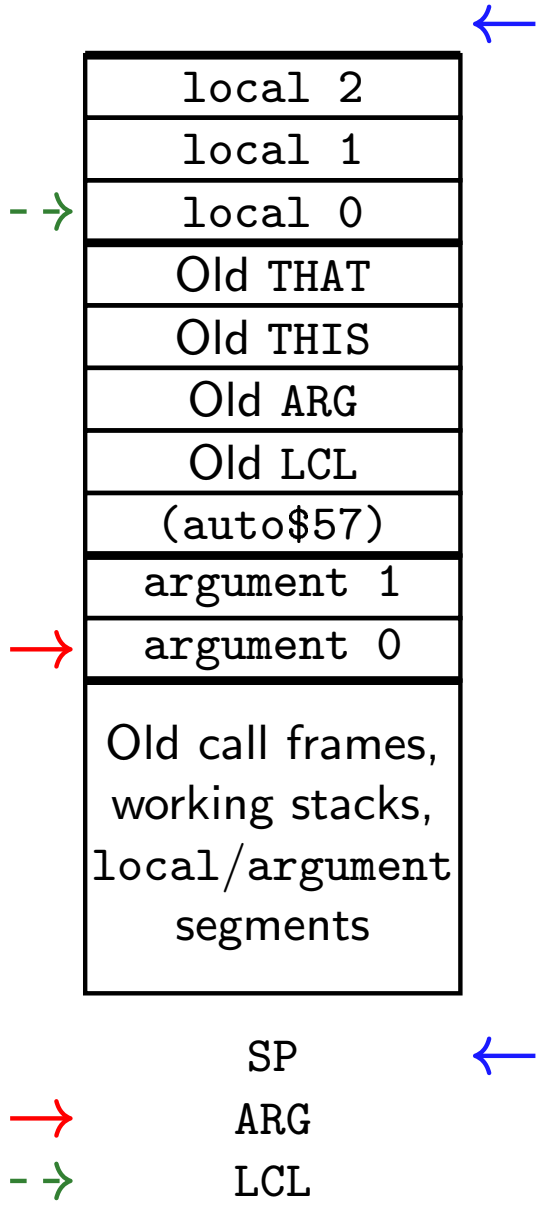


[a] This is part of the Hack VM specification. They don't explain their reasoning, but I think it's for security. Even in a single-process OS like DOS, different function calls may belong to different processes, and it makes sense to prevent them from seeing each other's stale stack memory. They actually can't already do this already via the `this` or `that` segments — the VM emulator only allows `this` to be used for heap memory, and for `that` to be used for heap memory, `SCREEN`, and `KBD`.

Implementing function definitions

Say our VM translator sees the line `function myFunc 3`. On the right we continue the example from last slide. The assembly code we generate must:

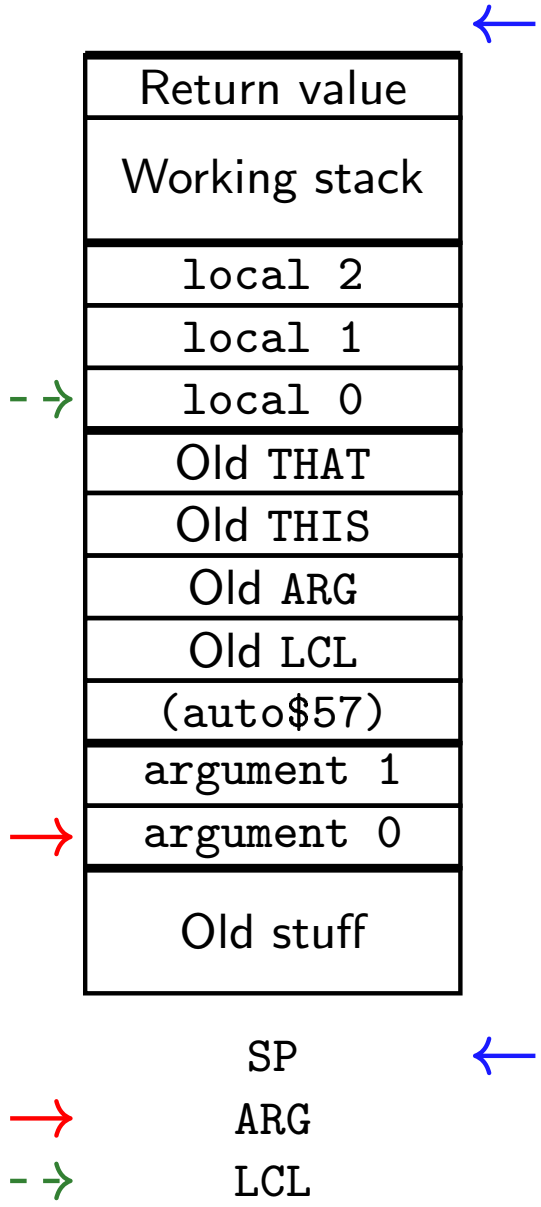
- Begin with a label, which we will jump to each time the function is called. To avoid the need for a symbol table, we should be able to get this label just from the function name (so we can derive it from the `call` statement).
- Set SP to LCL + 3, getting the length of the `local` segment from the `function` command.
- Initialise `local 0`, `local 1` and `local 2` to zero.[a]
- Continue into the first line of actual function code.



[a] This is part of the Hack VM specification. They don't explain their reasoning, but I think it's for security. Even in a single-process OS like DOS, different function calls may belong to different processes, and it makes sense to prevent them from seeing each other's stale stack memory. They actually can't already do this already via the `this` or `that` segments — the VM emulator only allows `this` to be used for heap memory, and for `that` to be used for heap memory, `SCREEN`, and `KBD`.

Implementing function returns

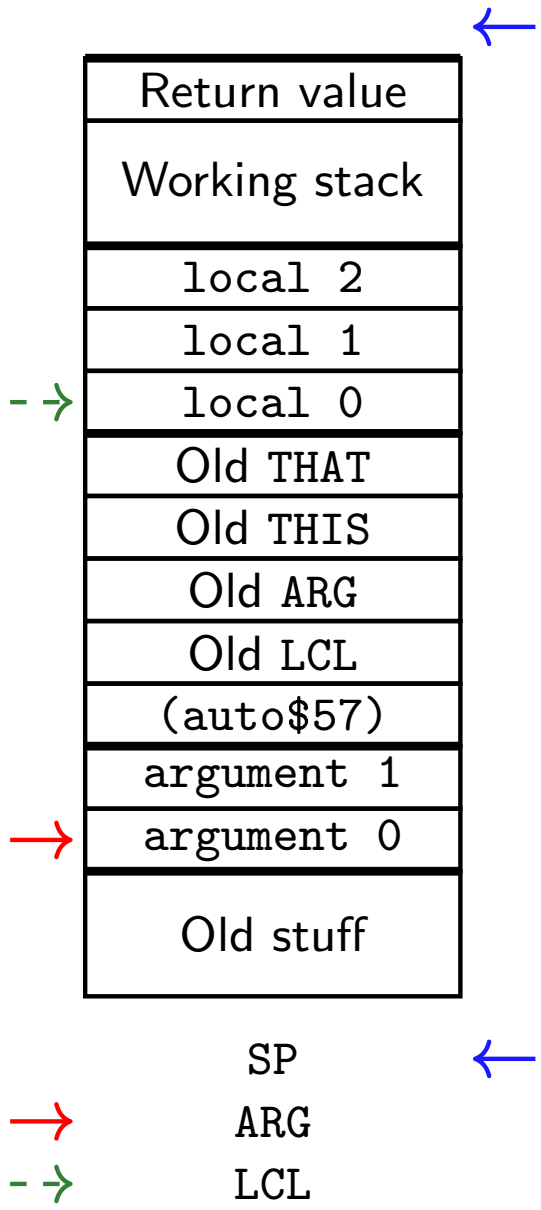
Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:



Implementing function returns

Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:

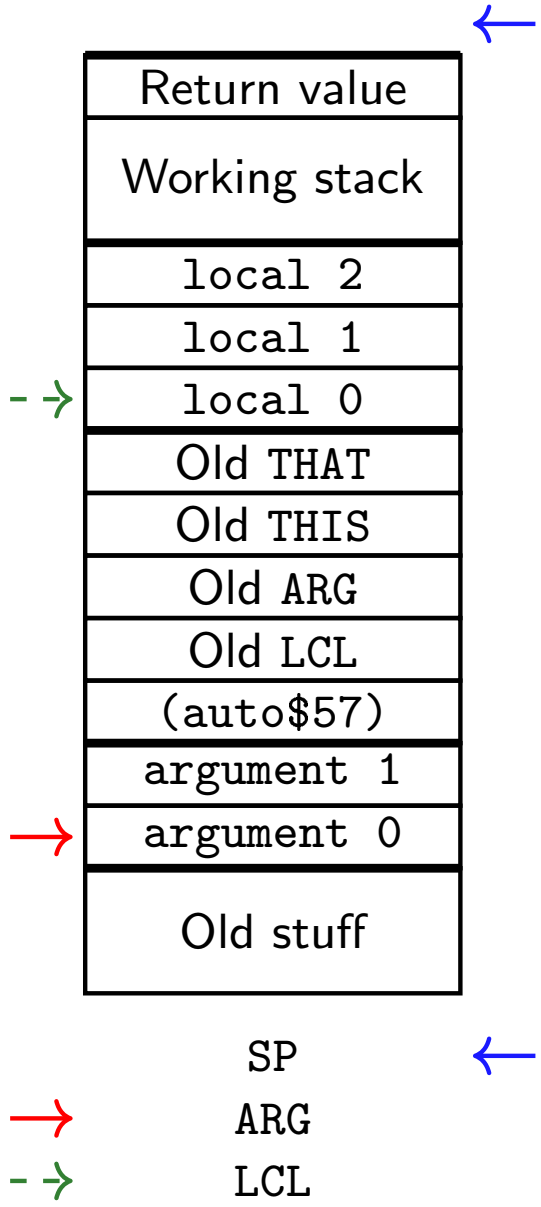
- Temporarily store the return address (e.g. in `R13`).



Implementing function returns

Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:

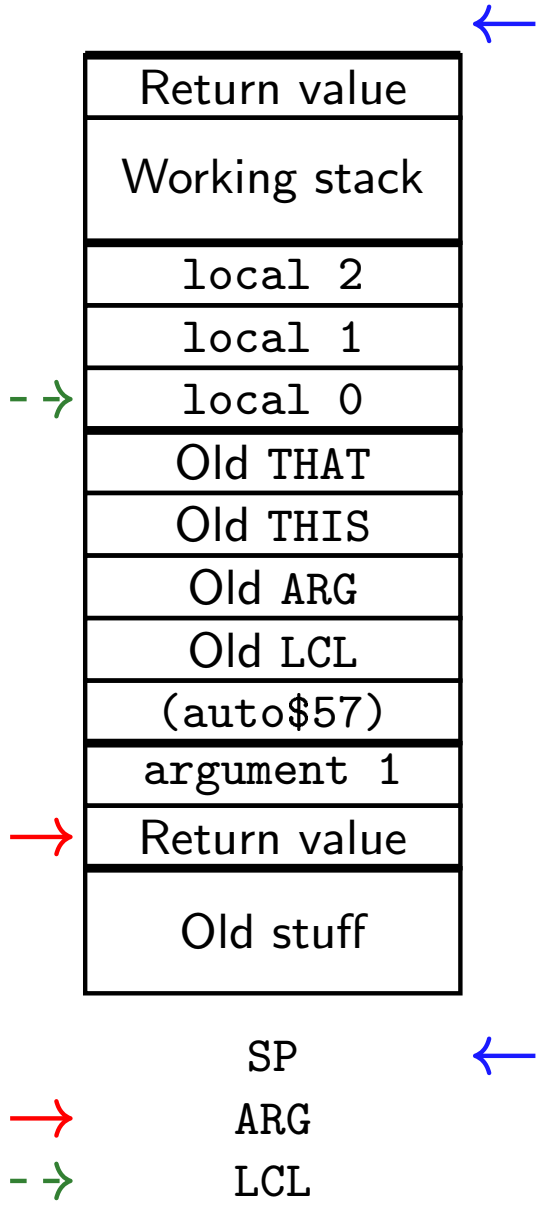
- Temporarily store the return address (e.g. in `R13`).
- Copy the return value to the our new working stack, i.e. the current value of `ARG`.



Implementing function returns

Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:

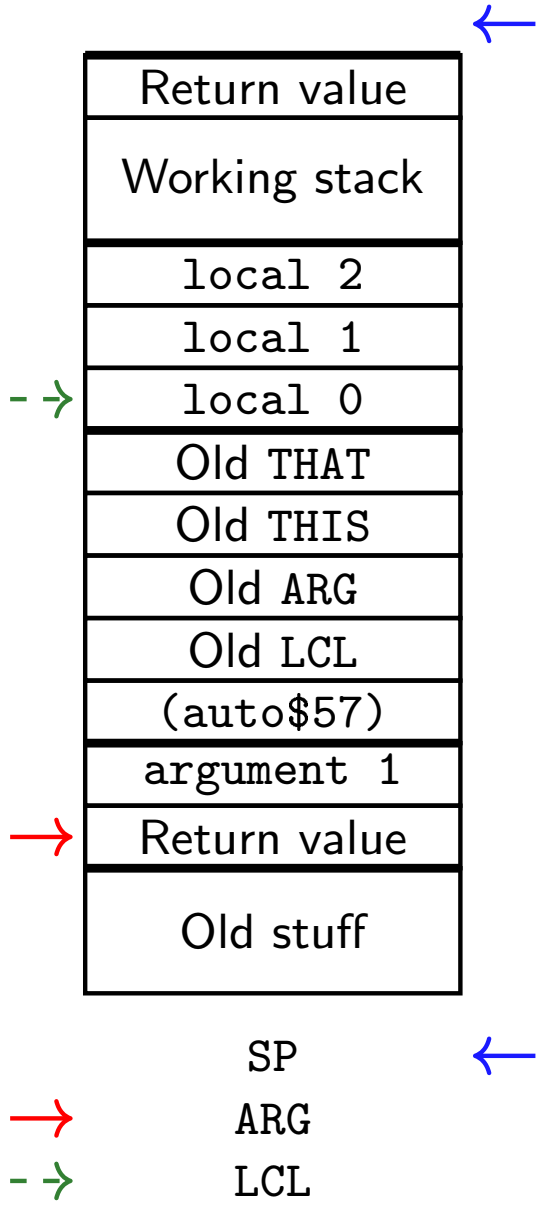
- Temporarily store the return address (e.g. in `R13`).
- Copy the return value to the our new working stack, i.e. the current value of `ARG`.



Implementing function returns

Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:

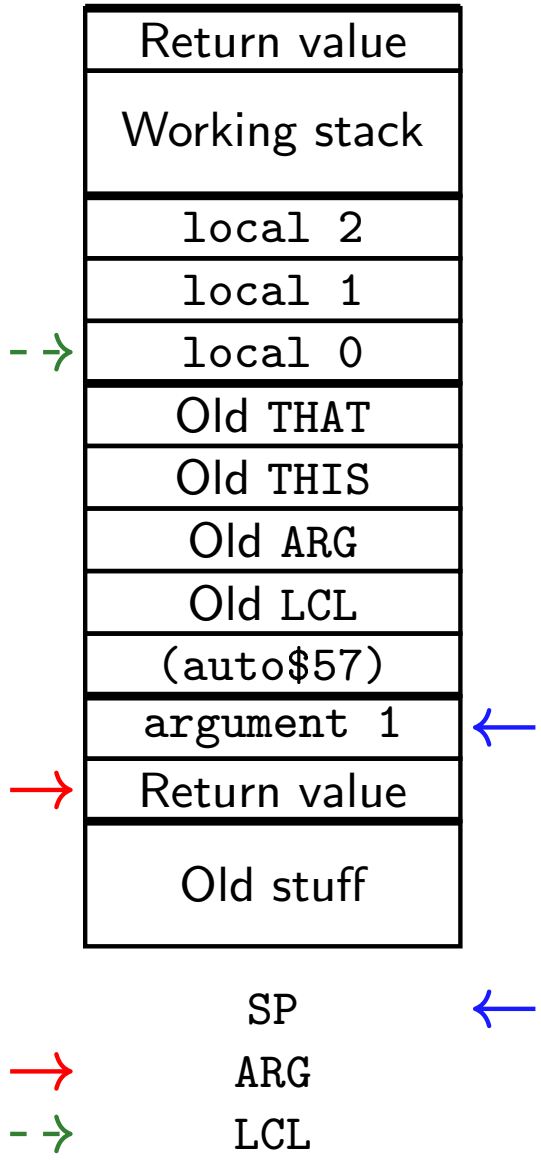
- Temporarily store the return address (e.g. in `R13`).
- Copy the return value to the our new working stack, i.e. the current value of `ARG`.
- Set `SP` to the top of our new working stack, i.e. `ARG` $+ 1$.



Implementing function returns

Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:

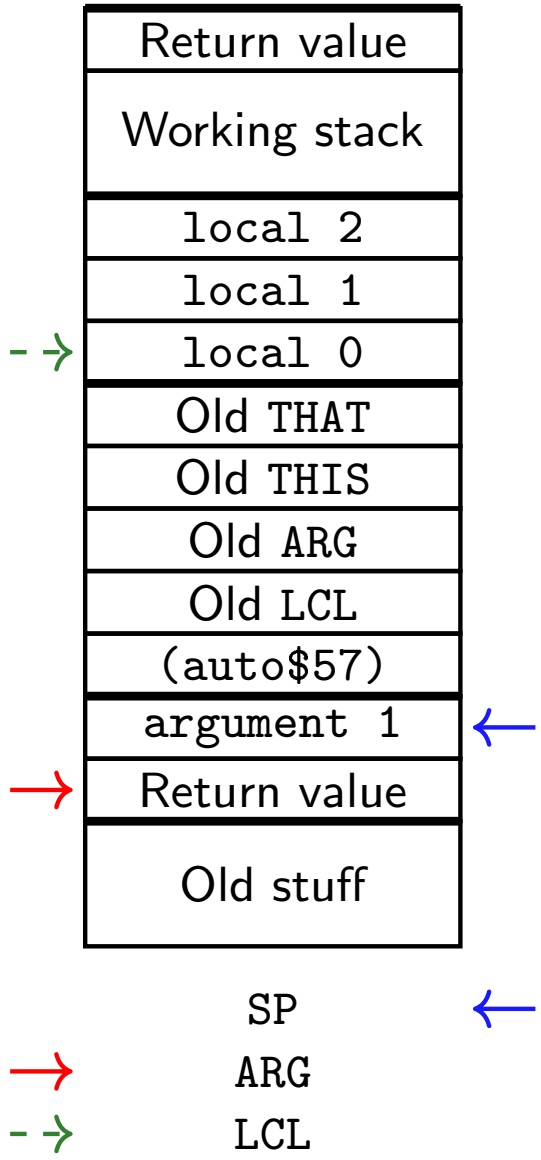
- Temporarily store the return address (e.g. in `R13`).
- Copy the return value to the our new working stack, i.e. the current value of `ARG`.
- Set `SP` to the top of our new working stack, i.e. `ARG` $+ 1$.



Implementing function returns

Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:

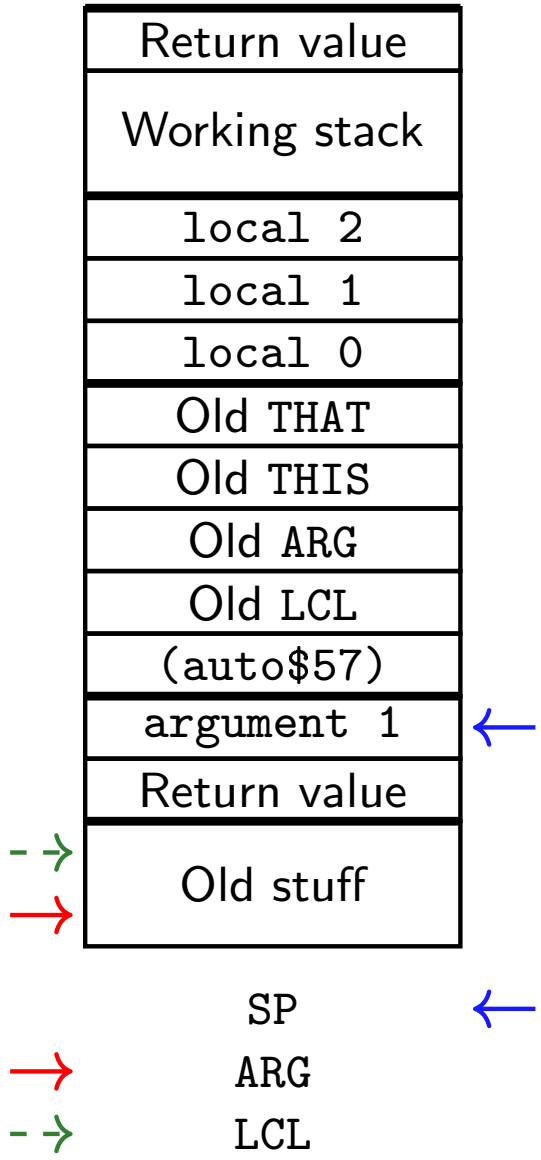
- Temporarily store the return address (e.g. in `R13`).
- Copy the return value to the our new working stack, i.e. the current value of `ARG`.
- Set `SP` to the top of our new working stack, i.e. `ARG` + 1.
- Restore the old values of `THAT`, `THIS`, `ARG` and `LCL`, counting down from the current value of `LCL` to find them on the stack.



Implementing function returns

Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:

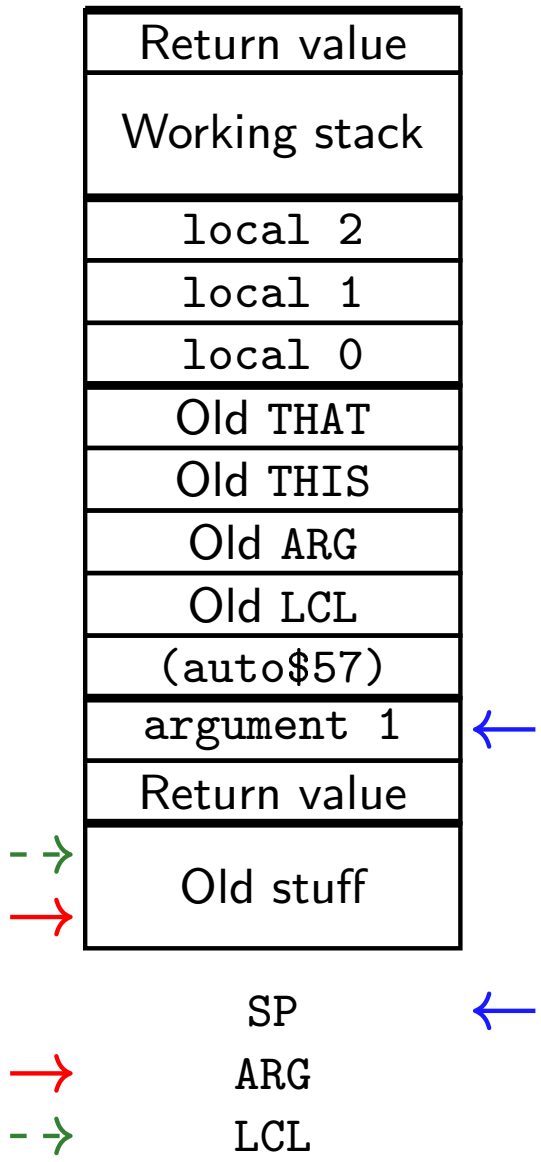
- Temporarily store the return address (e.g. in `R13`).
- Copy the return value to the our new working stack, i.e. the current value of `ARG`.
- Set `SP` to the top of our new working stack, i.e. `ARG` + 1.
- Restore the old values of `THAT`, `THIS`, `ARG` and `LCL`, counting down from the current value of `LCL` to find them on the stack.



Implementing function returns

Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:

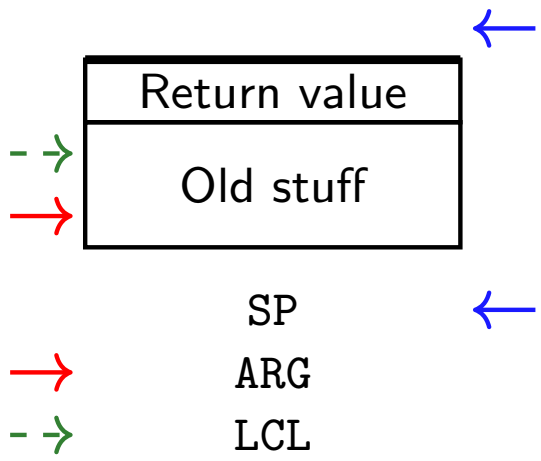
- Temporarily store the return address (e.g. in `R13`).
- Copy the return value to the our new working stack, i.e. the current value of `ARG`.
- Set `SP` to the top of our new working stack, i.e. `ARG` + 1.
- Restore the old values of `THAT`, `THIS`, `ARG` and `LCL`, counting down from the current value of `LCL` to find them on the stack.
- Jump to the return address, effectively discarding everything above `SP`.



Implementing function returns

Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:

- Temporarily store the return address (e.g. in `R13`).
- Copy the return value to the our new working stack, i.e. the current value of `ARG`.
- Set `SP` to the top of our new working stack, i.e. `ARG` $+ 1$.
- Restore the old values of `THAT`, `THIS`, `ARG` and `LCL`, counting down from the current value of `LCL` to find them on the stack.
- Jump to the return address, effectively discarding everything above `SP`.

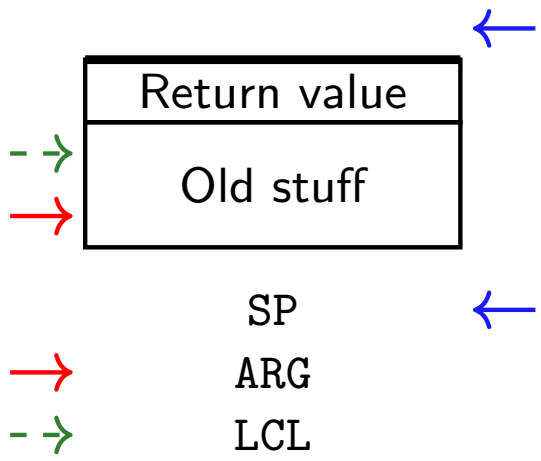


Implementing function returns

Say our VM translator sees the line `return`. On the right we continue the example from last slide. The assembly code we generate must:

- Temporarily store the return address (e.g. in `R13`).
- Copy the return value to the our new working stack, i.e. the current value of `ARG`.
- Set `SP` to the top of our new working stack, i.e. `ARG` $+ 1$.
- Restore the old values of `THAT`, `THIS`, `ARG` and `LCL`, counting down from the current value of `LCL` to find them on the stack.
- Jump to the return address, effectively discarding everything above `SP`.

Notice that our code didn't ever need to know which function we called or where we called it from!



Initialisation

But how do we set `LCL` and `ARG` at the start of the program?

Initialisation

But how do we set `LCL` and `ARG` at the start of the program?

Remember, multi-file Hack VM programs start by calling `Sys.init`. The values they have before this call don't matter. After the call, `LCL` and `ARG` will be set correctly in the usual way.

The default behaviour of the official `Sys.init` function is to call initialisation functions from all the other libraries, then call a function called `Main.main`, then enter an infinite loop.

(When compiling from Jack, `Main.main` will be the compiled analogue of the `main` function in C — the function the program starts in.)

Initialisation

But how do we set `LCL` and `ARG` at the start of the program?

Remember, multi-file Hack VM programs start by calling `Sys.init`. The values they have before this call don't matter. After the call, `LCL` and `ARG` will be set correctly in the usual way.

The default behaviour of the official `Sys.init` function is to call initialisation functions from all the other libraries, then call a function called `Main.main`, then enter an infinite loop.

(When compiling from Jack, `Main.main` will be the compiled analogue of the `main` function in C — the function the program starts in.)

You now have everything you need for this week's assignment. Next video we discuss heap memory allocation, i.e. implementing `malloc` and `free`.