Agenda

The VM Language (Part 2)

- Branching
  - label, goto, if-goto
- Functions
  - function, call, return

Examples

(With slides from nand2tetris.org)

Example

```
int mult(int x, int y) {
  int sum;
  sum = 0;
  for (int j = y; j != 0; j--) {
    sum += x;
  }
  return sum;
}
```

Original Code

```
function mult
  args x, y
  vars sum, j
  sum = 0
  j = y
loop:
  if j = 0 goto end
  sum = sum + x
  j = j -1
  goto loop
end:
 return sum
```

First approximation

What changes between each step?

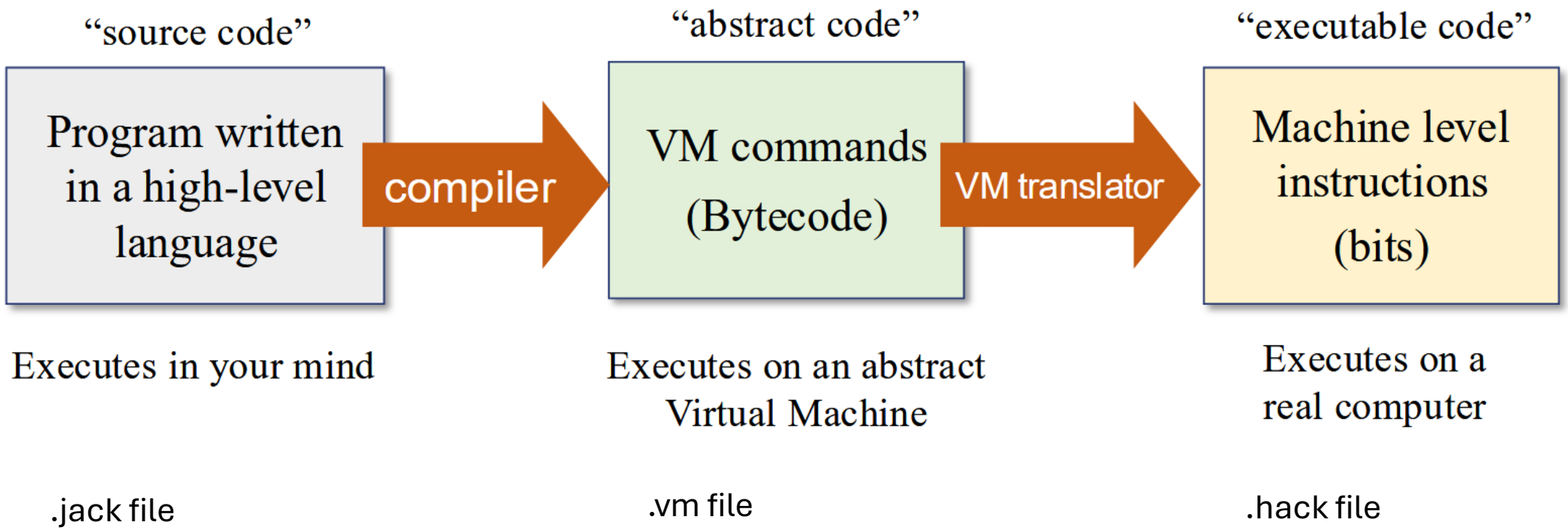
```
function mult(x, y)
  push 0
  pop sum
  push y
  pop j
label loop
  push 0
  push j
  eq
  if-goto end
  push sum
  push x
  add
   pop sum
   push j
   push 1
   sub
   pop j
  goto loop
label end
   push sum
   return
```

VM Pseudocode

```
function mult 2
  push constant 0
  pop local 0 // sum = 0
  push argument 1
  pop local 1  // j=y
label loop
  push constant 0
  push local 1
  eq
  if-goto end
  push local 0 // sum
  push argument 0 // x
  add
  pop local 0 // sum=sum+x
  push local 1
  push constant 1
  sub
  pop local 1 //j=j-1
  goto loop
label end
   push local 0
   return
```

VM Code

Recall: The big picture



Branching

```
// Returns arg 0 * arg 1
function mult 2
   push constant 0
   pop local 0
   push constant 1
   pop local 1
label WHILE_LOOP
   push local 1
   push argument 1
   gt
   if-goto END_LOOP
   push local 0
   push argument 0
   add
   pop local 0
   push local 1
   push constant 1
   add
   pop local 1
   goto WHILE_LOOP
label END_LOOP
   push local 0
   return
```

VM branching commands (syntax)

`label` *label*

`goto` *label*

`if-goto` *label*

label:

Destinations for goto commands

goto:

Jumps the command after label

if-goto:

let condition = pop
if condition, jump to label
else execute the next command

Exercise: Identify the commands associated with the while loop

Branches: Implementation in hack

VM branch commands can be directly translated to hack assembly. For example, convert the commands below to hack assembly

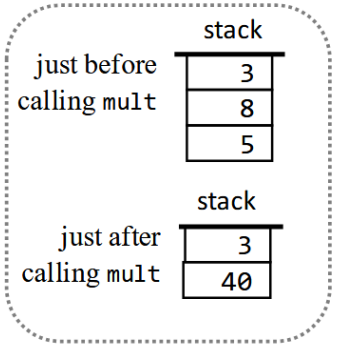
```
label loop
  ...
  eq
  if-goto end
  ...
  goto loop
label end
  ...
```

Functions: call, function, return

caller

```
function bar 4
   ...
   // Computes 3 + 8 * 5
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
   return
```

bar's view:



callee

```
// Returns arg 0 * arg 1
function mult 2
   push constant 0
   pop local 0
   push constant 1
   pop local 1
label LOOP
   push local 1
   push argument 1
   gt
   if-goto END
   push local 0
   push argument 0
   add
   pop local 0
   push local 1
   push constant 1
   add
   pop local 1
   goto LOOP
label END
   push local 0
   return
```

Typical scenario

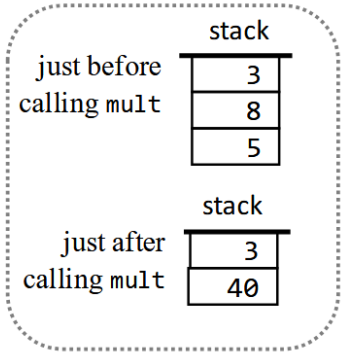
A function (the *caller*) calls a function (the *callee*) for its effect

Functions: call

caller

```
function bar 4
  ...
  // Computes 3 + 8 * 5
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  ...
  return
```

bar's view:



callee

```
// Returns arg 0 * arg 1
function mult 2
   push constant 0
   pop local 0
   push constant 1
   pop local 1
label LOOP
   push local 1
   push argument 1
   gt
   if-goto END
   push local 0
   push argument 0
   add
   pop local 0
   push local 1
   push constant 1
   add
   pop local 1
   goto LOOP
label END
   push local 0
   return
```

VM function commands

- `call`
- `function`
- `return`

Syntax: `call` *functionName nArgs*

Semantics: Calls function *functionName* for its effect, informing that *nArgs* argument values were pushed onto the stack

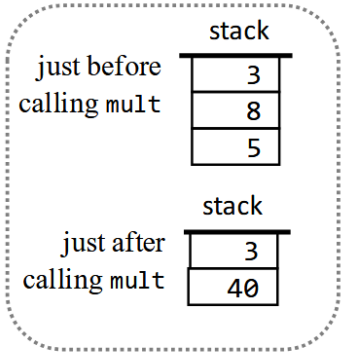
Convention: The caller must push *nArgs* arguments onto the stack before the `call` command.

Functions: function

caller

```
function bar 4
  ...
  // Computes 3 + 8 * 5
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  ...
  return
```

bar's view:



callee

```
// Returns arg 0 * arg 1
function mult 2
   push constant 0
   pop local 0
   push constant 1
   pop local 1
label LOOP
   push local 1
   push argument 1
   gt
   if-goto END
   push local 0
   push argument 0
   add
   pop local 0
   push local 1
   push constant 1
   add
   pop local 1
   goto LOOP
label END
   push local 0
   return
```

VM function commands

- `call`
- ➡ `function`
- `return`

Syntax: `function` *functionName nVars*

Semantics

Here starts the declaration of a function that has name *functionName* and *nVars* local variables

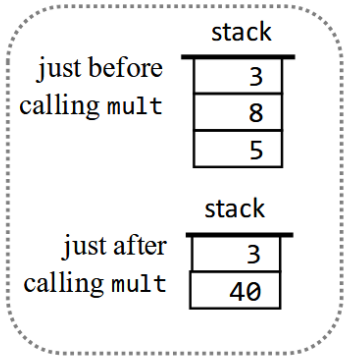
Note: In this example the caller passes 2 arguments, and the function has 2 local variables;
This is just a coincidence;
nArgs had nothing to do with *nVars*.

Functions: return

caller

```
function bar 4
  ...
  // Computes 3 + 8 * 5
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  ...
  return
```

bar's view:



callee

```
// Returns arg 0 * arg 1
function mult 2
   push constant 0
   pop local 0
   push constant 1
   pop local 1
label LOOP
   push local 1
   push argument 1
   gt
   if-goto END
   push local 0
   push argument 0
   add
   pop local 0
   push local 1
   push constant 1
   add
   pop local 1
   goto LOOP
label END
   push local 0
   return
```

VM function commands

- `call`
- `function`
- `return`

Syntax: `return`

Convention: The callee must
(1) push a return value onto the stack, and
(2) execute a `return` command

Semantics

The *return value* will replace (in the stack) the argument values that were pushed by the caller before the `call`;

Control will be transferred back to the caller;

Execution will resume with the command just after the `call`.

Functions: Implementation

1. Assigned function names are *Filename.FnName*

2. The caller must be “put on hold” while the callee executes

3. The callee starts with a fresh stack and its own virtual memory segments

4. When the callee finishes, we should resume the caller where it left off. Any return values should be on the caller’s stack

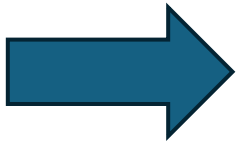
Implementation Example: The High Level

caller

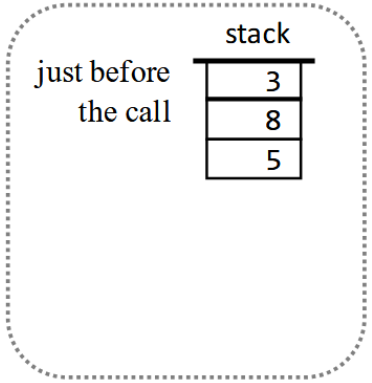
```
function Foo.bar 4
  ...
  // Computes 3 + 8 * 5
  push constant 3
  push constant 8
  push constant 5
  call Foo.mult 2
  add
  ...
  return
```

callee

```
// Returns arg 0 * arg 1
function Foo.mult 2
   push constant 0
   pop local 0
   push constant 1
   pop local 1
   ...
   push local 0
   return
```



Foo.bar's view



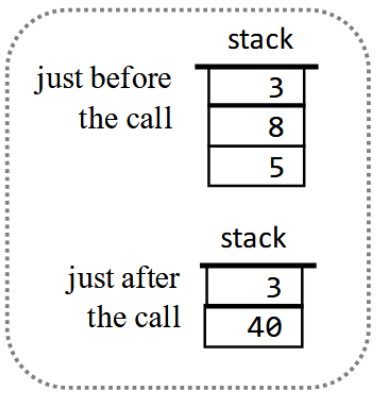
caller

```
function Foo.bar 4
  ...
  // Computes 3 + 8 * 5
  push constant 3
  push constant 8
  push constant 5
  call Foo.mult 2
  add
  ...
  return
```

callee

```
// Returns arg 0 * arg 1
function Foo.mult 2
   push constant 0
   pop local 0
   push constant 1
   pop local 1
   ...
   push local 0
   return
```

Foo.bar's view



Magic!

Let's open the black box

Implementation Example: Details Step 1

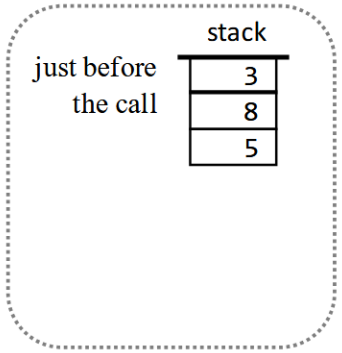
caller

```
function Foo.bar 4
   ...
   // Computes 3 + 8 * 5
   push constant 3
   push constant 8
   push constant 5
   call Foo.mult 2
   add
   ...
   return
```

callee

```
   // Returns arg 0 * arg 1
0  function Foo.mult 2
1     push constant 0
2     pop local 0
3     push constant 1
4     pop local 1
...   ...
20    push local 0
21    return
```

Foo.bar's view



The caller's execution is put on hold

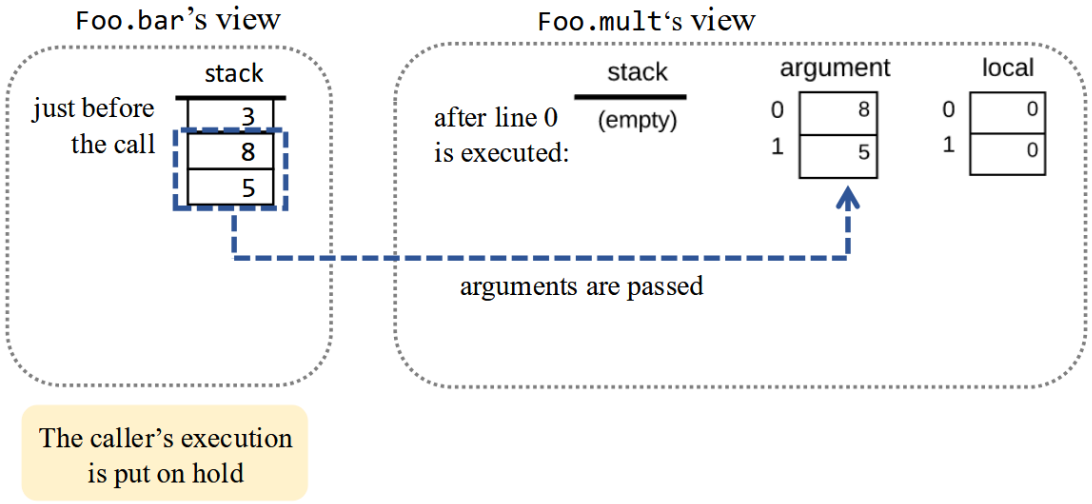
Implementation Example: Details Step 2

caller

```
function Foo.bar 4
  ...
  // Computes 3 + 8 * 5
  push constant 3
  push constant 8
  push constant 5
  call Foo.mult 2
  add
  ...
  return
```

callee

```
  // Returns arg 0 * arg 1
0 function Foo.mult 2
1    push constant 0
2    pop local 0
3    push constant 1
4    pop local 1
...  ...
20   push local 0
21   return
```



The caller's execution is put on hold

Implementation Example: Details Step 3

caller

```
function Foo.bar 4
   ...
   // Computes 3 + 8 * 5
   push constant 3
   push constant 8
   push constant 5
   call Foo.mult 2
   add
   ...
   return
```

callee

```
   // Returns arg 0 * arg 1
0  function Foo.mult 2
1     push constant 0
2     pop local 0
3     push constant 1
4     pop local 1
...   ...
20    push local 0
21    return
```

Foo.bar's view

just before the call

stack
3
8
5

Foo.mult's view

after line 0 is executed:

stack: (empty)

	argument
0	8
1	5

	local
0	0
1	0

The callee's code is executed

Implementation Example: Details Step 4

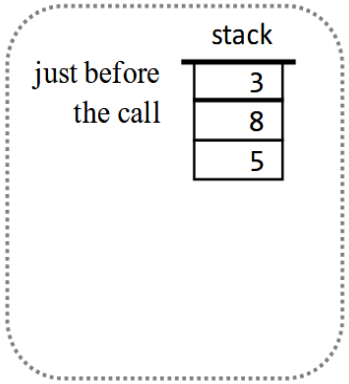
caller

```
function Foo.bar 4
  ...
  // Computes 3 + 8 * 5
  push constant 3
  push constant 8
  push constant 5
  call Foo.mult 2
  add
  ...
  return
```

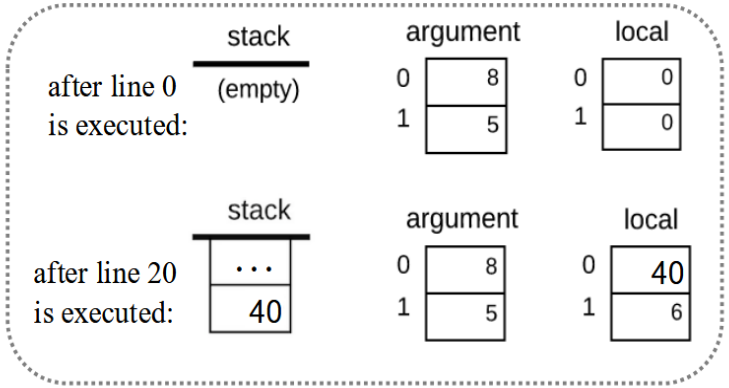
callee

```
  // Returns arg 0 * arg 1
0 function Foo.mult 2
1    push constant 0
2    pop local 0
3    push constant 1
4    pop local 1
...  ...
20   push local 0
21   return
```

Foo.bar's view



Foo.mult's view



The callee's code is executed

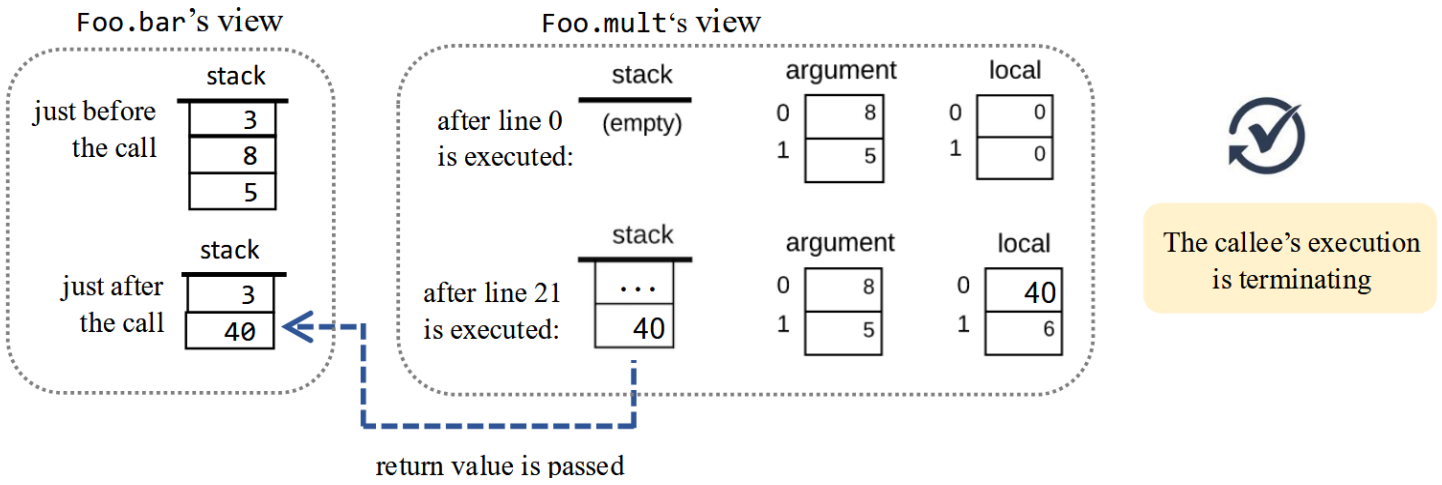
Implementation Example: Details Step 5

caller

```
function Foo.bar 4
  ...
  // Computes 3 + 8 * 5
  push constant 3
  push constant 8
  push constant 5
  call Foo.mult 2
  add
  ...
  return
```

callee

```
  // Returns arg0 * arg1
0 function Foo.mult 2
1    push constant 0
2    pop local 0
3    push constant 1
4    pop local 1
...  ...
20   push local 0
21   return
```



The callee's execution is terminating

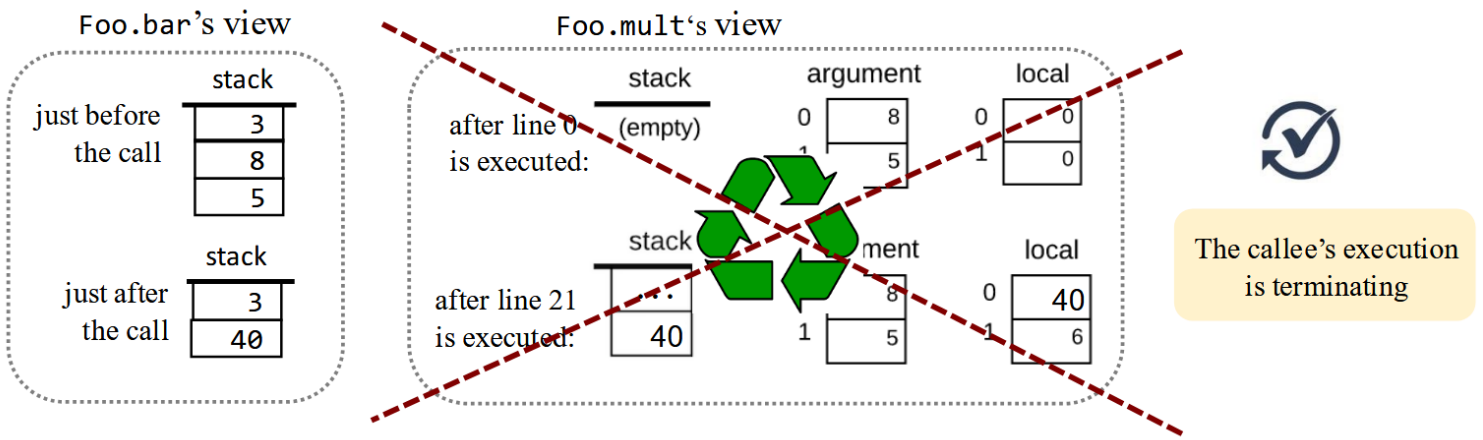
Implementation Example: Details Step 6

caller

```
function Foo.bar 4
   ...
   // Computes 3 + 8 * 5
   push constant 3
   push constant 8
   push constant 5
   call Foo.mult 2
   add
   ...
   return
```

callee

```
   // Returns arg 0 * arg 1
0  function Foo.mult 2
1     push constant 0
2     pop local 0
3     push constant 1
4     pop local 1
...   ...
20    push local 0
21    return
```



Implementation Example: Details Step 7

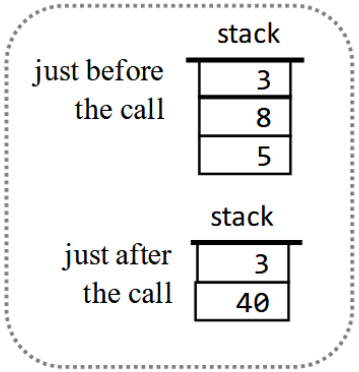
caller

```
function Foo.bar 4
  ...
  // Computes 3 + 8 * 5
  push constant 3
  push constant 8
  push constant 5
  call Foo.mult 2
  add
  ...
  return
```

callee

```
  // Returns arg 0 * arg 1
0 function Foo.mult 2
1    push constant 0
2    pop local 0
3    push constant 1
4    pop local 1
...  ...
20   push local 0
21   return
```

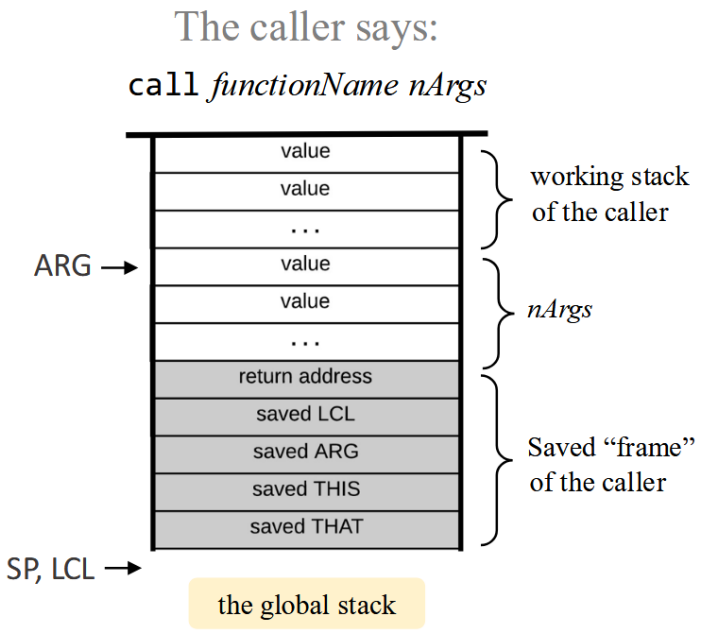
Foo.bar's view



The caller's execution is resumed

(the next command to be executed: `add`)

Function implementation: call



Handling `call` *functionName nArgs*

We have to:

- Save the return address
- Save the caller's segment pointers
- Reposition `ARG` (for the callee)
- Reposition `LCL` (for the callee)
- Go to execute the callee's code

Generated code

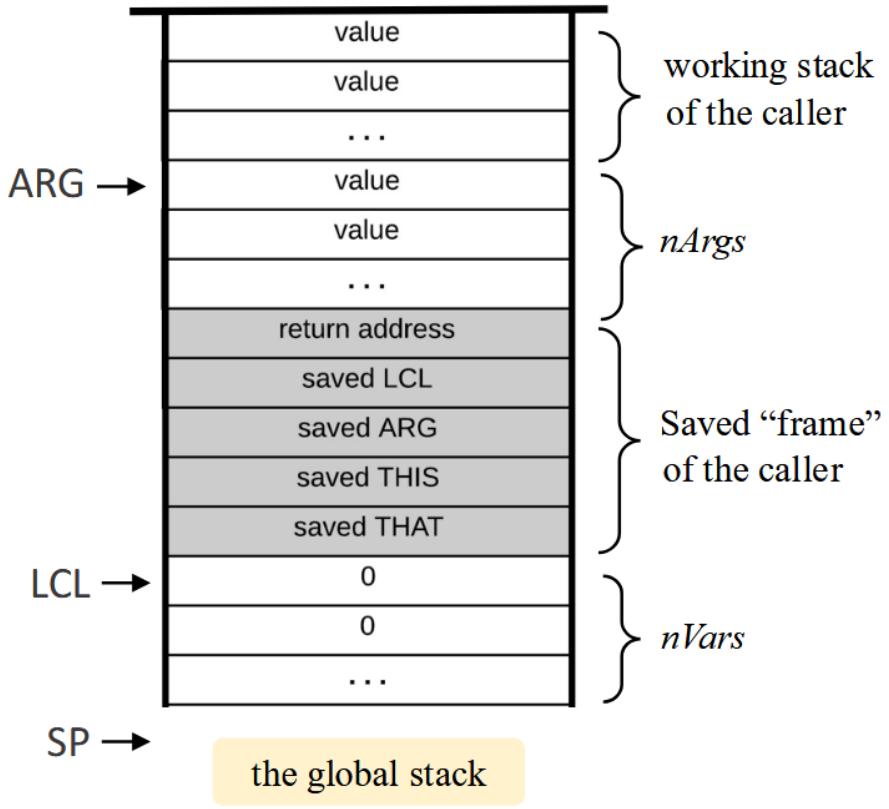
```
// call functionName nArgs
   push retAddrLabel     // Generates and pushes this label
   push LCL              // Saves the caller's LCL
   push ARG              // Saves the caller's ARG
   push THIS             // Saves the caller's THIS
   push THAT             // Saves the caller's THAT
   ARG = SP – 5 – nArgs  // Repositions ARG
   LCL = SP              // Repositions LCL
   goto functionName     // Transfers control to the callee
(retAddrLabel)           // Injects this label into the code
```

(The VM translator must generate all this pseudocode in assembly)

Function implementation: function

The callee is entered:

`function` *functionName nVars*



Handling `function` *functionName nVars*

We have to:

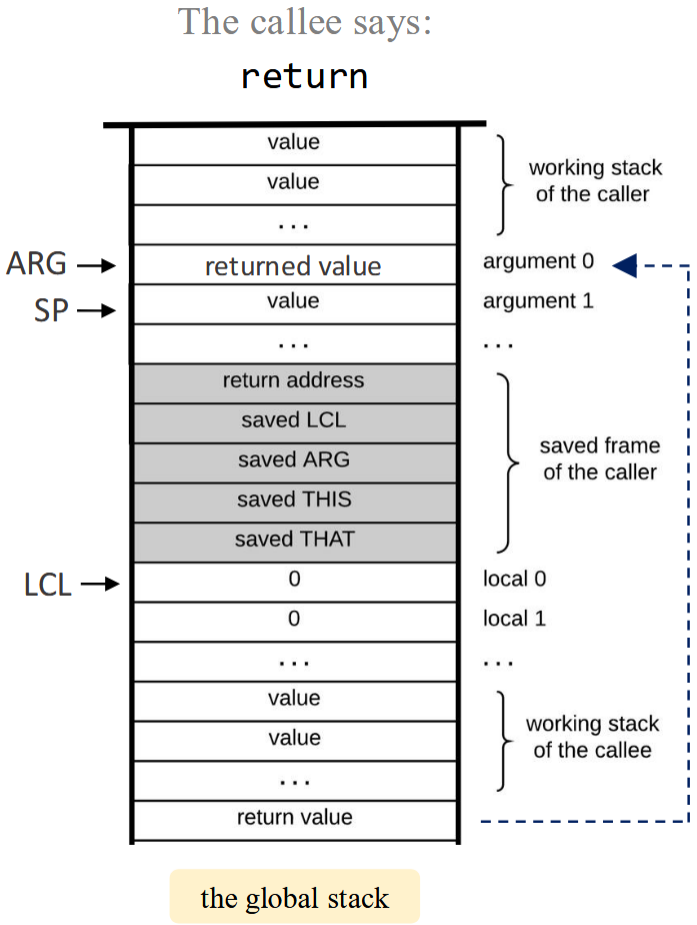
- Inject an entry point label into the code
- Initialize the `local` segment of the callee

Generated code

```
// function functionName nVars
(functionName)          // function's entry point (injected label)
// push nVars 0 values (initializes the callee's local variables)
push 0
...
push 0
```

(The VM translator must generate all this pseudocode in assembly)

Function implementation: return



the global stack

Handling `return`:

We have to:

1. Replace the arguments that the caller pushed with the value returned by the callee
2. Recycle the memory used by the callee
3. Restore the caller's segment pointers
4. Jump to the return address

Generated code

```
// The code below creates and uses two temporary variables:
// endFrame and retAddr;
// The pointer notation *addr is used to denote: RAM[addr]
endFrame = LCL                  // gets the address at the frame's end
retAddr = *(endFrame – 5)       // gets the return address
*ARG = pop()                    // puts the return value for the caller
SP = ARG + 1                    // repositions SP
THAT = *(endFrame – 1)          // restores THAT
THIS = *(endFrame – 2)          // restores THIS
ARG = *(endFrame – 3)           // restores ARG
LCL = *(endFrame – 4)           // restores LCL
goto retAddr                    // jumps to the return address
```

(The VM translator must generate all this pseudocode in assembly)

Example: Factorial

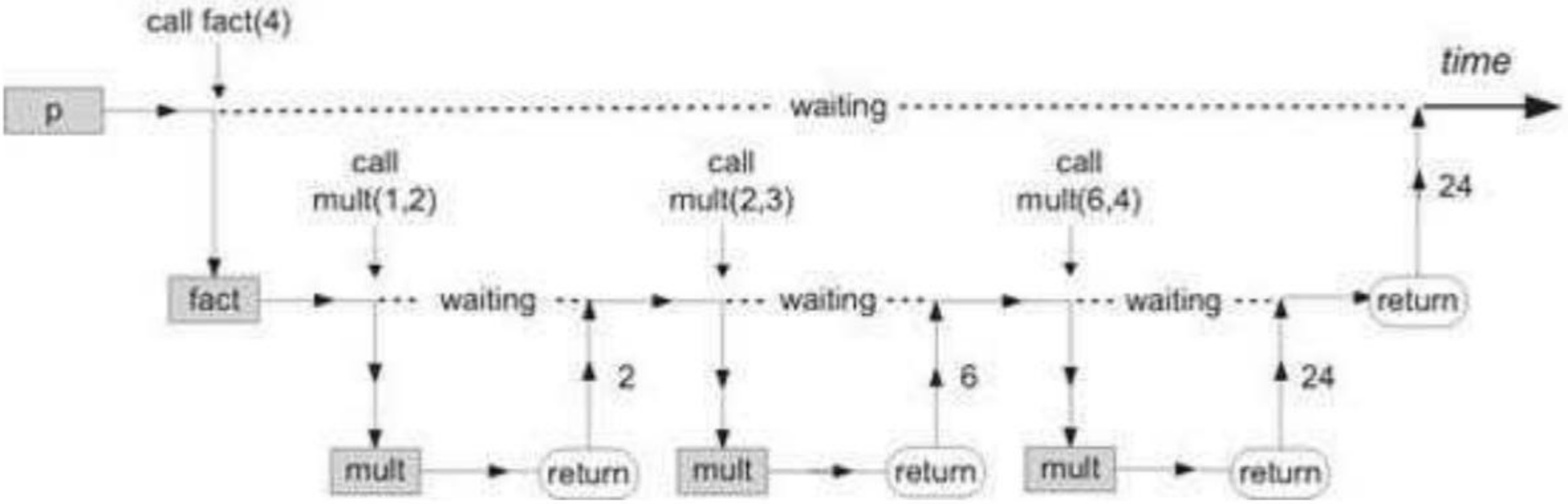
```
int mult(int x, int y) {
  int sum;
  sum = 0;
  for (int j = y; j != 0; j--) {
    sum += x;
  }
  return sum;
}
```

```
int factorial(int n) {
  int result = 1;
  for (int j = 1; j <= n; j++) {
    result = mult(result, j);
  }
  return result;
}

void p() {
   factorial(4);
}
```

Exercise: Draw the function stack for factorial(2)

Visualizing function calls



Exercise: Write VM Code for p and factorial

Example: Factorial

```
function p
  ...
  // Compute 4!
  push constant 4
  call fact 1 // 1 arg
```

```
function factorial 2 // 2 local variables
   push constant 1
   pop local 0  // result = 1
   push constant 1
   pop local 1   // j =1
label loop
   push constant 1
   push local 1
   add
   pop local 1  // j=j+1
   push local 1
   push argument 0
   gt
   if-goto end  // if j>n goto end
   push local 0
   push local 1
   call mut 2 // 2 arguments pushed
```

```
// function factorial 2  cont.
   pop local 0
   goto loop
label end
   push local 0
   return
```

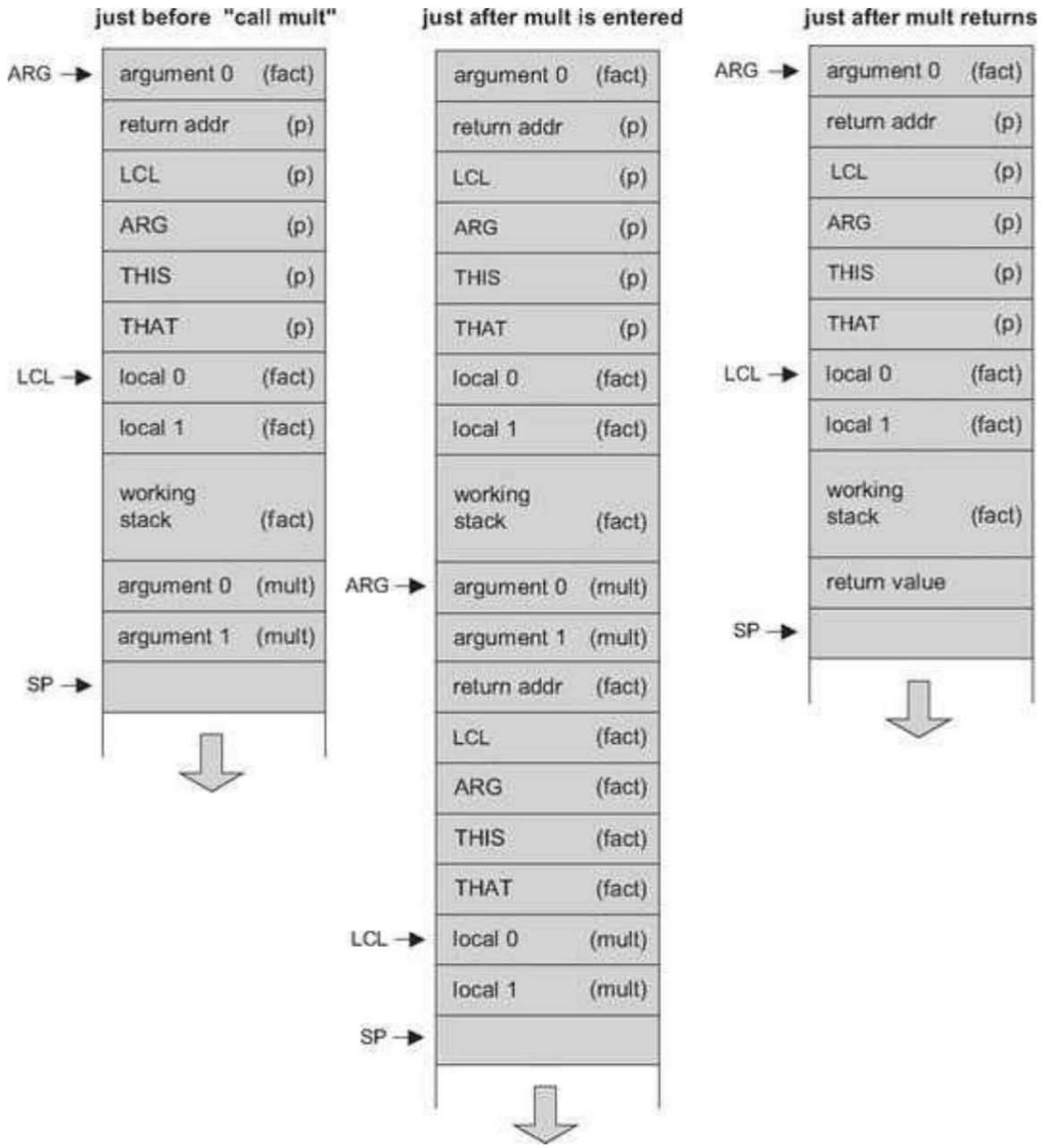
```
   function mult 2
     ...
     // Returns result
     push local 0
     return
```

Example: Factorial

Each function pushes its state to the stack before executing the function.

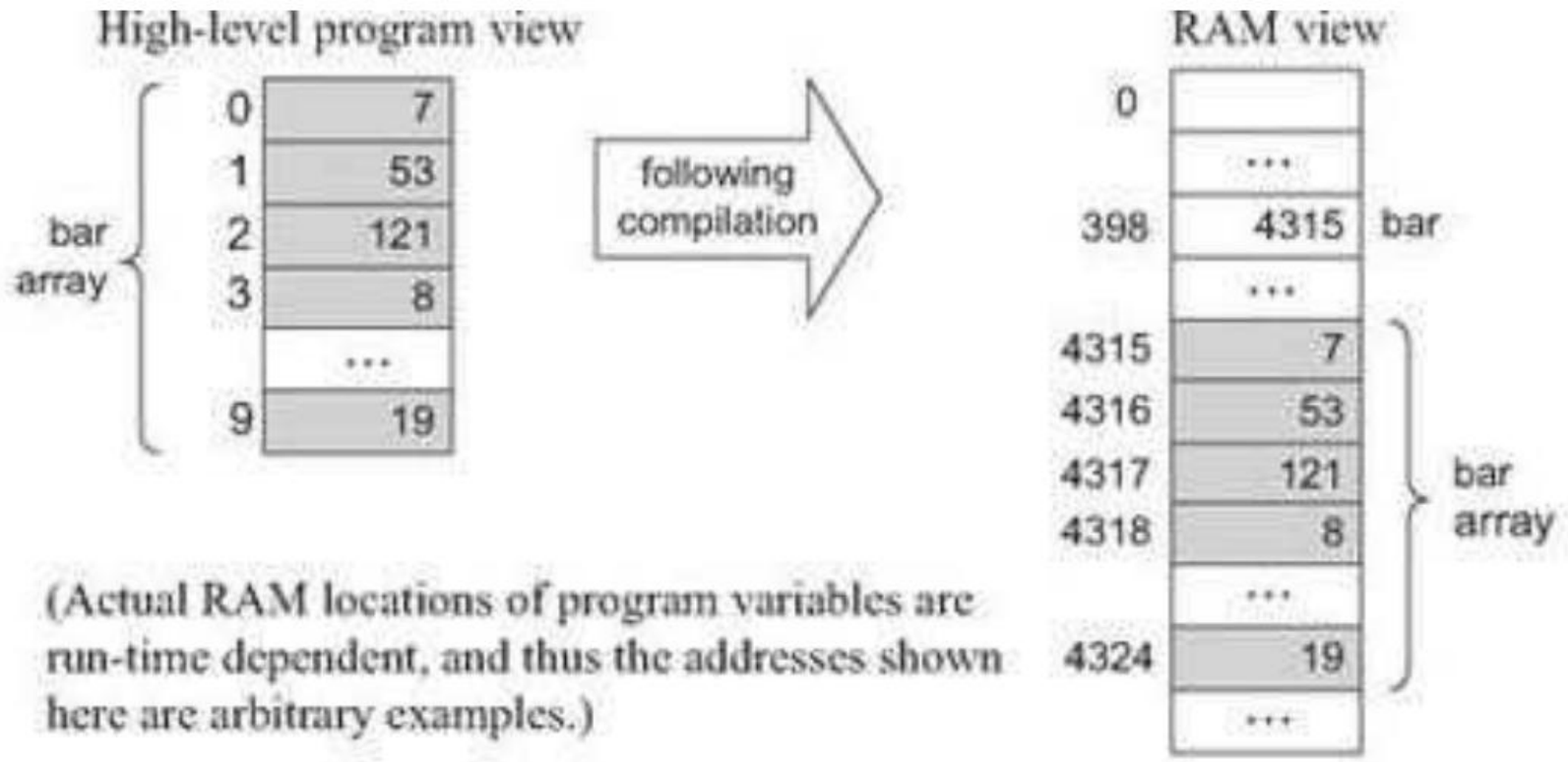
This state is restored when we return from the function.

The return address works the same as labels used for loops or if statements (it's an address in ROM)



Example: Arrays

```
// High-level Code
int* bar = {7, 53, 121, 8, 1, 2, 3, 4, -5, 19};
bar[2] = 19; // same as *(bar + 2) = 19
```

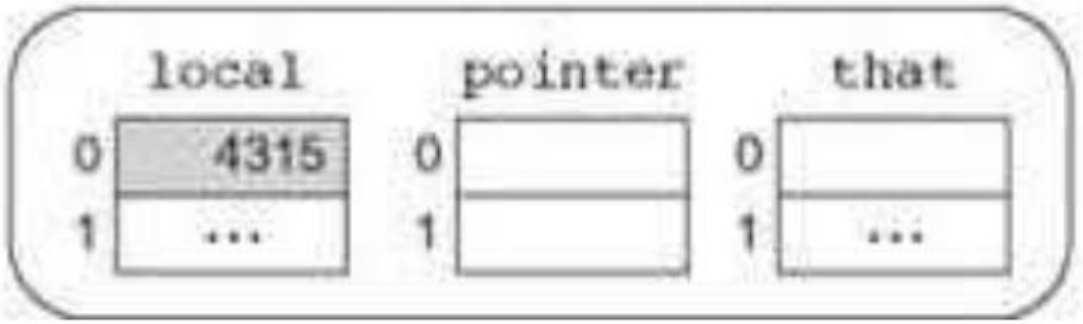


Example: Arrays

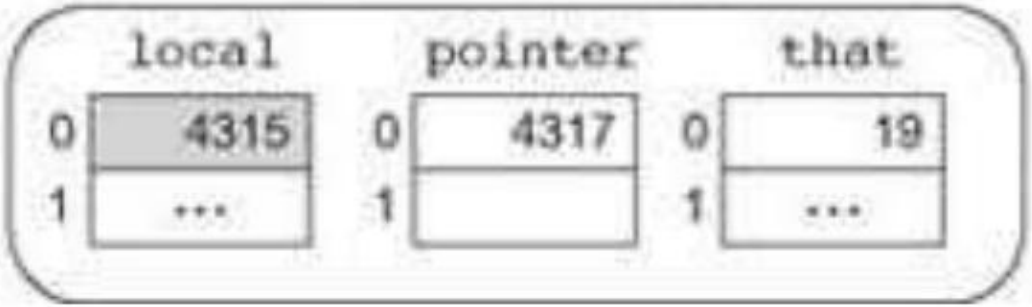
```
// High-level Code
int* bar = {7, 53, 121, 8, 1, 2, 3, 4, -5, 19};
bar[2] = 19; // same as *(bar + 2) = 19

// VM Code
push local 0
push constant 2
add
pop pointer 1 // Sets base to (bar+2)
push constant 19
pop that 0 // *(bar+2) = 19
```

Virtual memory segments just before the bar[2]=19 operation:



Virtual memory segments just after the bar[2]=19 operation:



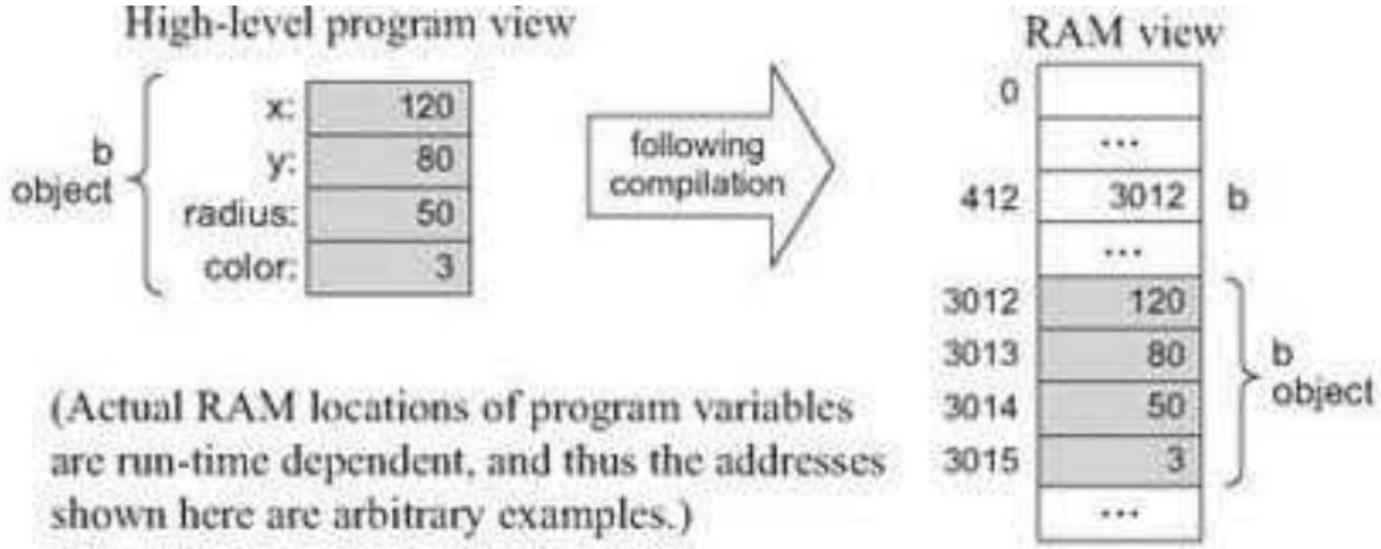
(that 0 is now aligned with RAM[4317])

Example: Objects

```
// High-level code

class Ball {
  int x;
  int y;
  int radius;
  int color;
....

  void setRadius(int r) {
    radius = r; // same as self.radius = r
  }
}
```

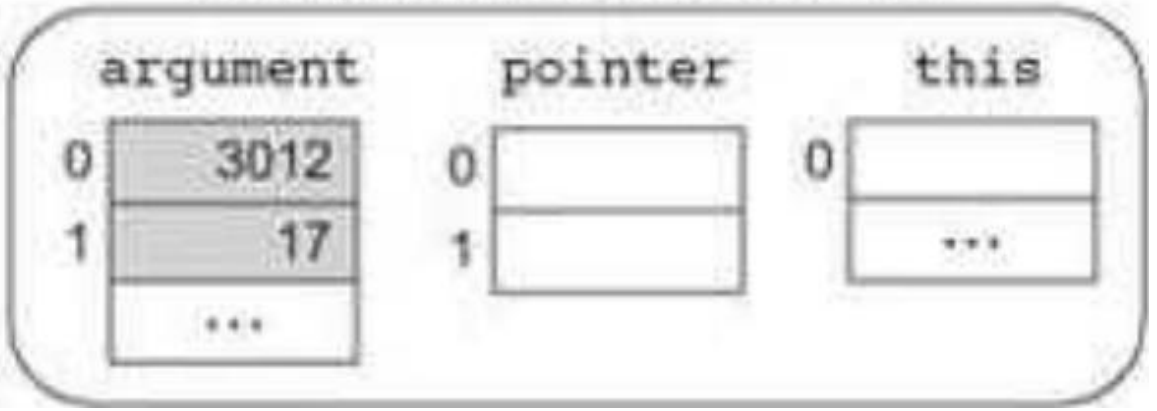


Example: Objects

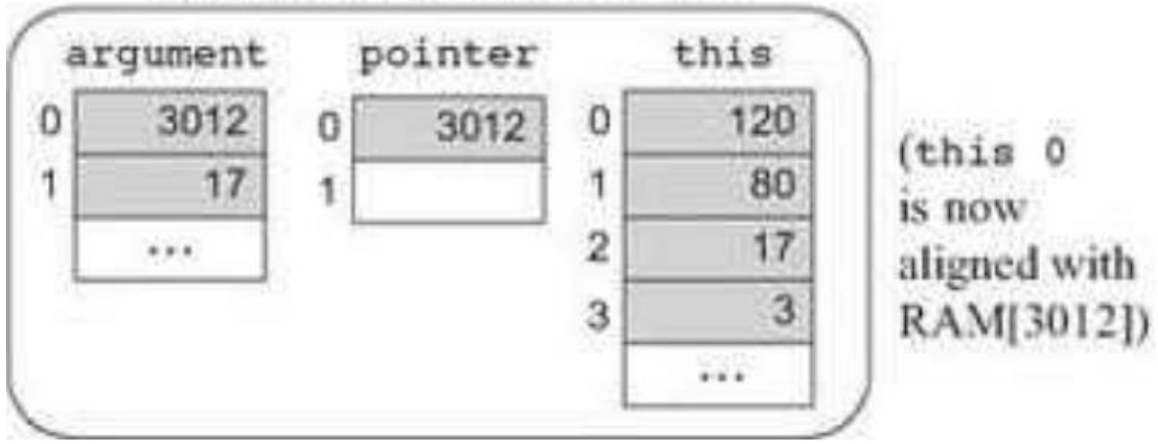
```
// VM code inside setRadius
// Assume that the Ball objects and r are
// passed to the function as arguments 0
// and 1 respectively.
// The following code implements
//    self.radius = r

push argument 0 // Get base address of self
pop pointer 0       // Save base address in ptr
push argument 1 // Get r's value
pop this 2              // Set 3rd value of self
```

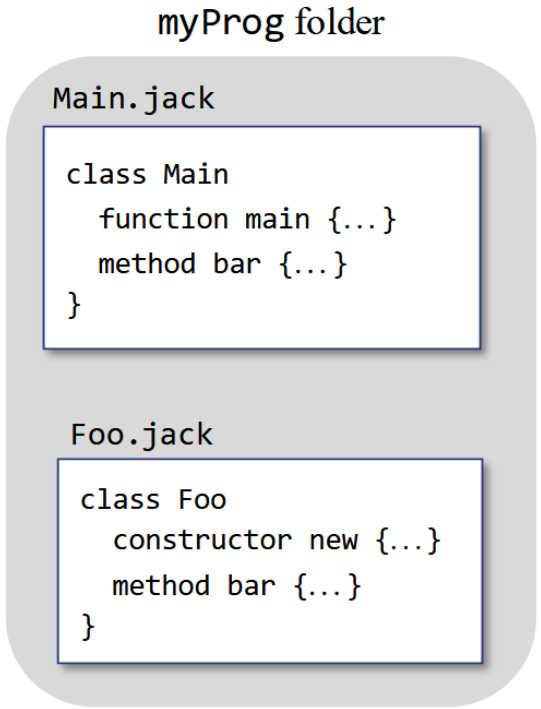
Virtual memory segments just before the operation `b.radius=17`:



Virtual memory segments just after the operation `b.radius=17`:



The Big Picture: Compilation



High level language conventions

(much more about it in lecture 9):

Jack program: a set of one or more class files, all in the same folder;

Jack class: a set of one or more *methods*, *functions* (static methods), and *constructors*;

There must be at least one class file named `Main.jack`,
and this file must contain at least one method named `main`;

Program's entry point: `Main.main()`

OS conventions

(much more about it in lecture 12):

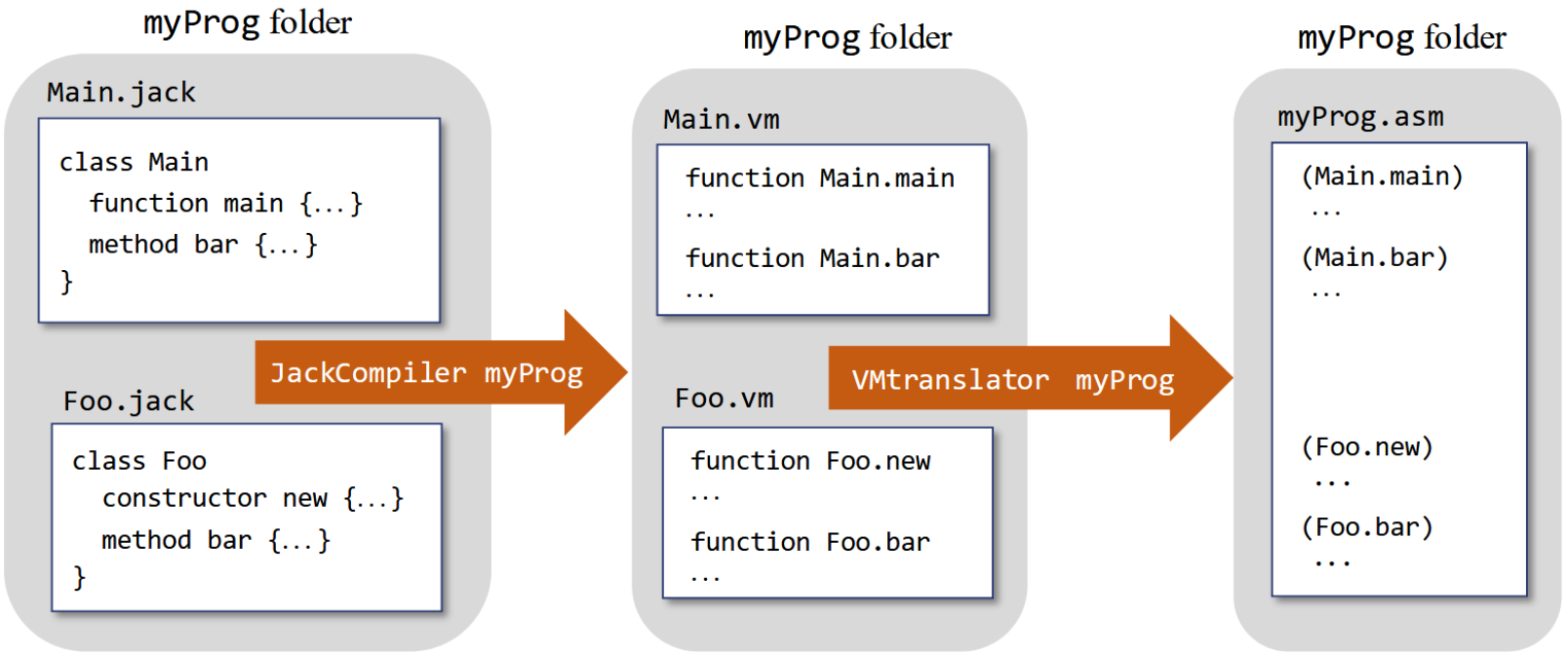
The OS is written in Jack (just like Unix is written in C);

One OS class, `Sys`, contains a method named `init`

When the computer boots, it executes `Sys.init`

`Sys.init` calls `Main.main`.

The big picture: Compilation steps



Each *Jack class* is a set of *methods*, *functions*, and *constructors*

Every *method*, *function*, and *constructor* is translated into a *VM function*

All the VM functions from all the compiled class files are translated into a single assembly file

(the notion of multiple VM files melts away)

Program's entry point: `Main.main()`

Bootstrap code, e.g. executing Main.main

Run-time conventions

The compiled code base includes the program:
A set of VM functions (in any order), one of which is `Main.main`

The compiled code base also includes the operating system:
Also a set of VM functions, one of which is `Sys.init`

`Sys.init` calls `Main.main`, and enters an infinite loop

The stack is stored in the RAM, starting at address 256

To make this happen

The assembly code generated by the VM translator should start with the following code:

```
// Bootstrap code
SP = 256
call Sys.init    // (no arguments)
```

(The VM translator must generate this pseudocode in assembly).

Standard VM mapping for the hack platform