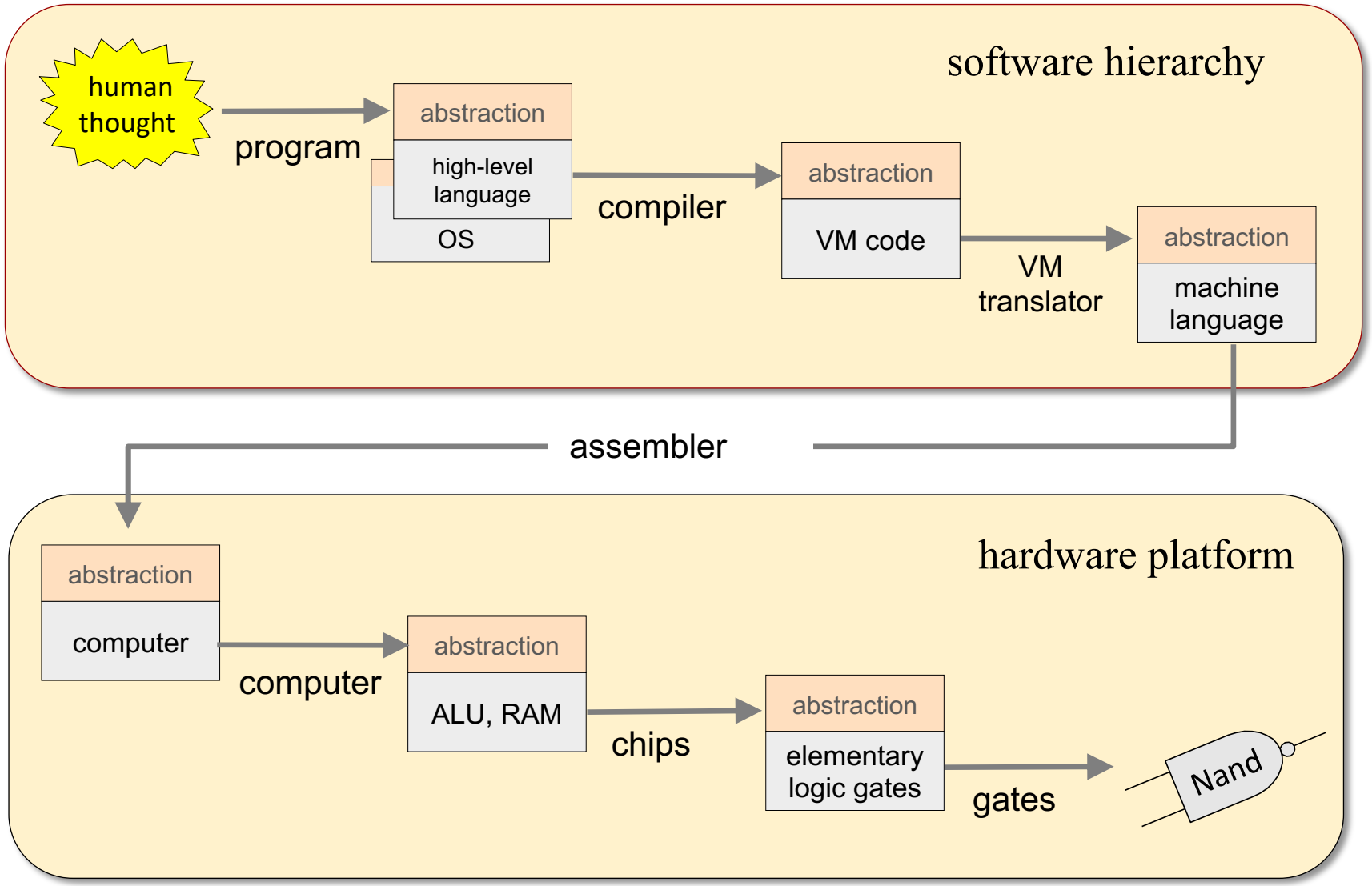
Chapter 8

Virtual Machine

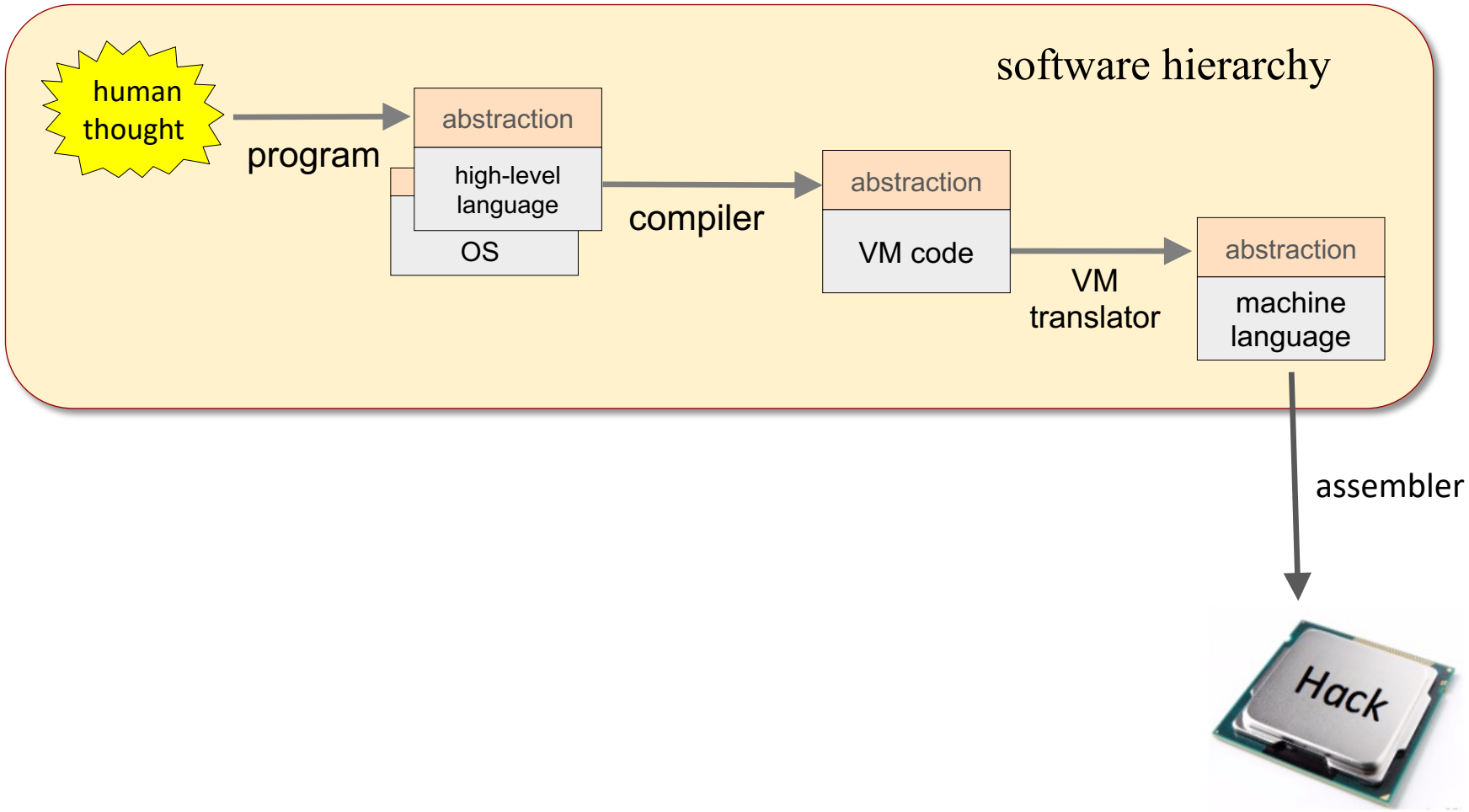
(Part II)

These slides support chapter 8 of the book
The Elements of Computing Systems
By Noam Nisan and Shimon Schocken
MIT Press, 2021

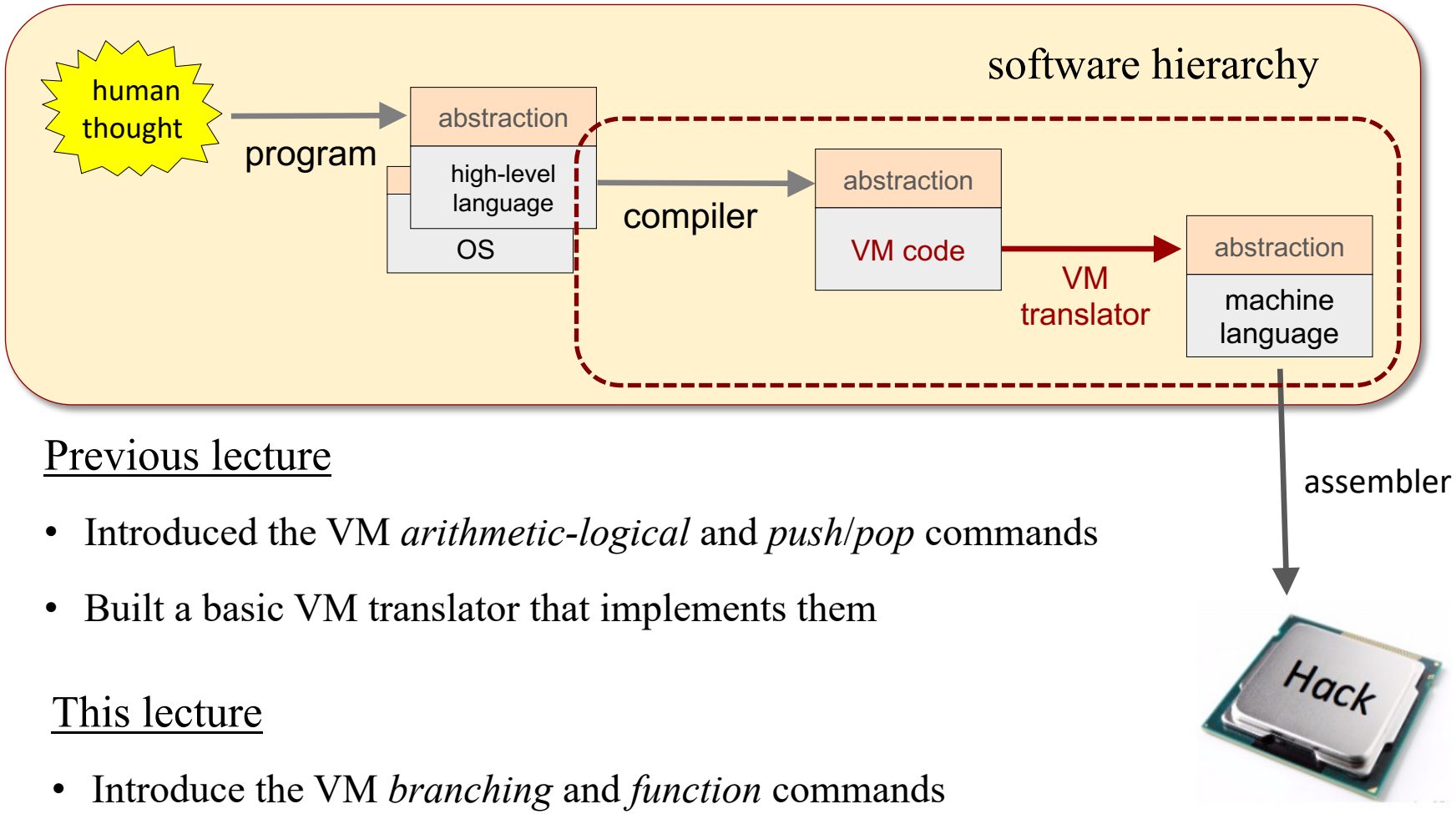
Nand to Tetris Roadmap



Nand to Tetris Roadmap: Part II



Nand to Tetris Roadmap: Part II



Previous lecture

- Introduced the VM *arithmetic-logical* and *push/pop* commands
- Built a basic VM translator that implements them

This lecture

- Introduce the VM *branching* and *function* commands
- Complete the VM translator.

The big picture: Compilation

High-level program

```
// Returns x * y
int mult(int x, int y) {
   int sum = 0;
   int n = 1;
   // sum = sum + x, y times
   while !(n > y) {
      sum += x;
      n++;
   }
   return sum;
}
```

(later in the course)

VM code

```
// Returns x * y
function mult 2
   push 0
   pop sum
   push 1
   pop n
label WHILE_LOOP
   push n
   push y
   gt
   if-goto END_LOOP
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto WHILE_LOOP
label END_LOOP
   push sum
   return
```

The VM Language

✓ **Arithmetic / Logical**
- `add`, `sub` , `neg`
- `eq` , `gt` , `lt`, `and`, `or` , `not`

✓ **Memory access**
- `push`
- `pop`

"imperatives"

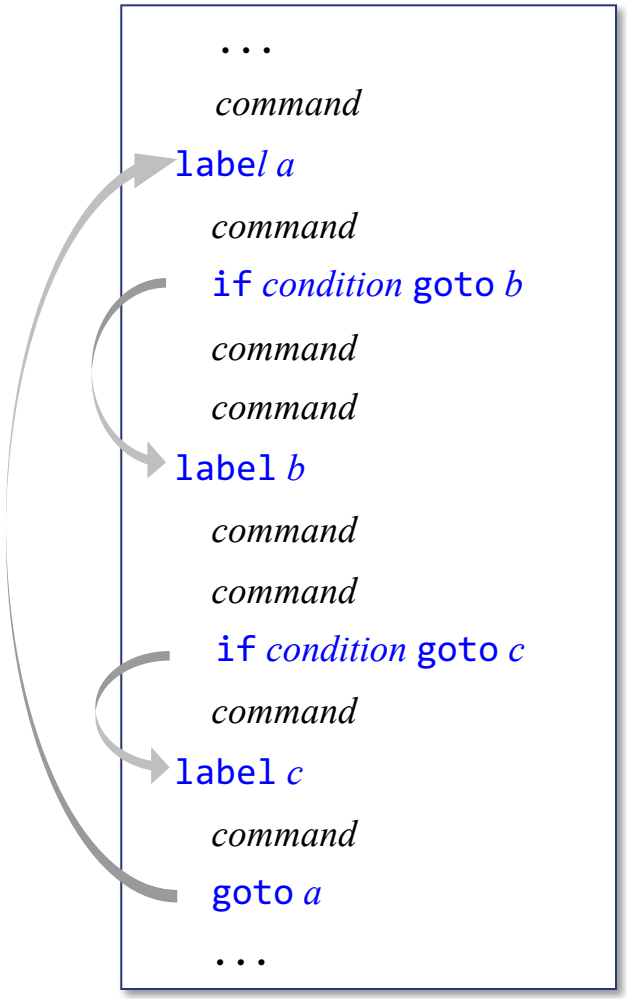
Branching
- `label`
- `goto`
- `if-goto`

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

"control"

Branching

Programs typically include branching logic (conditions, loops):



Branching

Unconditional: goto *label*

Conditional: if *condition* goto *label*

Implementation issues

How to realize…

- conditions?
- labels?
- branching operations?

Functions

Programs are typically divided into "*modules*":

- "Subroutines"
- "Procedures"
- "Methods"
- "functions"
- Etc.

} At the VM level, they all become *functions*

Functions

Function `foo`

```
// Some function
function foo
   ...
   // Computes 8 * 5 + 7
   push 8
   push 5
   call mult
   push 7
   add
   ...
   return
```

caller

Function `mult`

```
// Returns x * y
function mult
   push 0
   pop sum
   push 1
   pop n
label WHILE_LOOP
   push n
   push y
   gt
   if-goto END_LOOP
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto WHILE_LOOP
label END_LOOP
   push sum
   return
```

callee

Typical scenario

A function (like `foo`) calls a function (like `mult`) for its effect.

Implementation issues

How to pass arguments from the caller to the callee?

How to start executing the callee's code?

How to pass the return value from the callee to the caller?

How to resume the execution of the caller's code?

Take home lessons

Understanding program execution

- Branching
- Function call-and-return
- Recursion

Related issues

- Compilation
- Run-time system
- Memory management.

Branching: Abstraction

```
// Returns x * y
function mult
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto END
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto LOOP
label END
   push sum
   return
```

VM branching commands

```
label
goto
if-goto
```

Branching: Abstraction

```
// Returns x * y
function mult
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto END
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto LOOP
label END
   push sum
   return
```

VM branching commands

- `label`
- `goto`
- `if-goto`

Syntax: `label` *label*

Semantics

Marks the destination of `goto` commands.

Branching: Abstraction

```
// Returns x * y
function mult
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto END
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto LOOP
label END
   push sum
   return
```

VM branching commands

```
label
goto
if-goto
```

Syntax: `goto` *label*

Semantics

Jump to execute the command just after the *label*.

Branching: Abstraction

```
// Returns x * y
function mult
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto END
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto LOOP
label END
   push sum
   return
```

VM branching commands

`label`

`goto`

`if-goto`

Syntax: `if-goto` *label*

Semantics

1. let *cond* = `pop`
2. if *cond* jump to execute the command just after the *label* else execute the next command.

Assumption

The code writer is expected to write code that pushes a logical expression onto the stack before the `if-goto` command. In this example, the highlighted code implements the abstraction: `if (n > y) goto END`

Note: In this lecture, "code writer" is typically a comiler.

Branching: Implementation

Abstraction (recap)

`label` *label* // label declaration

`goto` *label* // jump to execute the command just after the *label*

`if-goto` *label* // let *cond* = `pop`
// if *cond* jump to execute the command just after the *label*

Implementation

For each VM branching command, we have to generate machine language instructions that realize the command on a target platform. Example:

VM code

```
    ...
label LOOP
    ...
    goto LOOP
    ...
```

VM translator

Hack code

```
    ...
(LOOP)
    ...
    @LOOP
    0;JMP
    ...
```

The instruction set of any target platform (like Hack) features low-level means for implementing "labeling" and "goto" semantics;

Therefore, the translation of the VM branching commands is not difficult.

Lecture plan

✓ Branching

- Abstraction
- Implementation

Function call and return

- Abstraction
- Implementation

The Virtual Machine

- Program translation
- Standard mapping
- VM translator
- Project 8

Functions: Abstraction

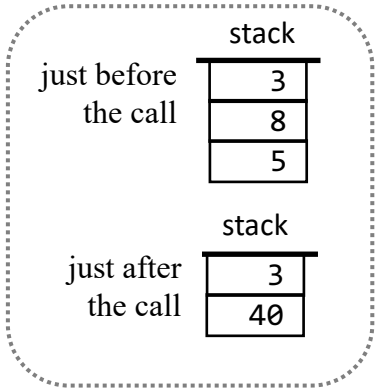
caller

```
// Computes 3 + 8 * 5
function foo
   push 3
   push 8
   push 5
   call mult
   add
   ...
```

callee

```
// Returns x * y
function mult
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto END
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto LOOP
label END
   push sum
   return
```

foo's view:



Typical scenario

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

Functions: Abstraction

caller

```
// Computes 3 + 8 * 5
function foo
  push 3
  push 8
  push 5
  call mult
  add
  ...
```

callee

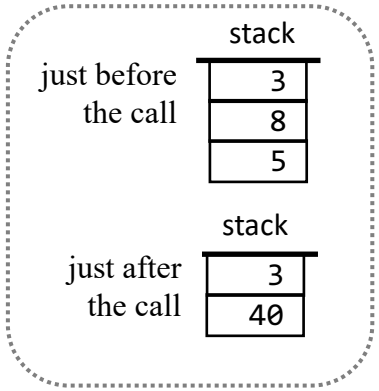
```
// Returns x * y
function mult
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto END
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto LOOP
label END
   push sum
   return
```

VM function commands

```
call
function
return
```

We will now switch from pseudo VM code to actual VM code

foo's view:



Functions: Abstraction

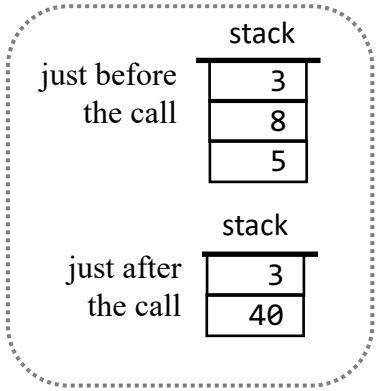
caller

```
// Computes 3 + 8 * 5
function foo 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  ...
```

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
```

foo's view:



VM function commands

```
call
function
return
```

Functions: Abstraction

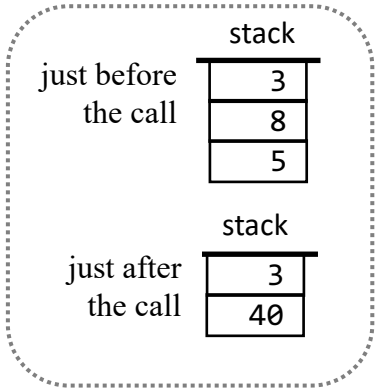
caller

```
// Computes 3 + 8 * 5
function foo 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  ...
```

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
```

foo's view:



VM function commands

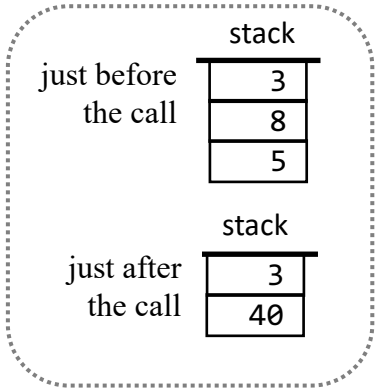
- `call`
- `function`
- `return`

Functions: Abstraction

caller

```
// Computes 3 + 8 * 5
function foo 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  ...
```

foo'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

Assumption

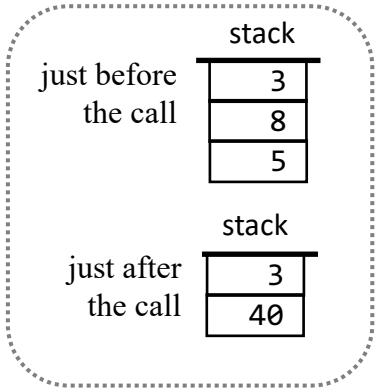
The code writer is expected to push *nArgs* arguments onto the stack before the `call` command.

Functions: Abstraction

caller

```
// Computes 3 + 8 * 5
function foo 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
```

foo'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

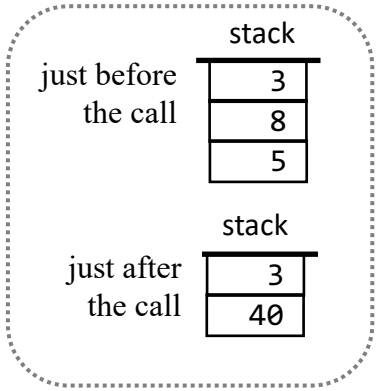
Here starts the declaration of a function that has name *functionName* and *nVars* local variables

Functions: Abstraction

caller

```
// Computes 3 + 8 * 5
function foo 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
```

foo'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`

Assumption

The code writer is expected to push a value onto the stack before the `return` command

Semantics

The *return value* will replace (in the stack) the argument values 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`.

Lecture plan

Branching

- Abstraction
- Implementation

Function call and return

✓ Abstraction

➡ Implementation

The Virtual Machine

- Program translation
- Standard mapping
- VM translator
- Project 8

Function call and return

caller

```
// Computes 3 + 8 * 5
function foo 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
```

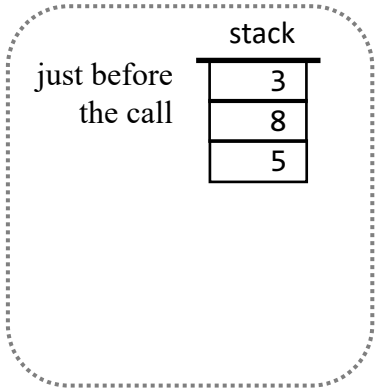
callee

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

Typical function-call-and-return scenario

Function `foo` (the *caller*) calls function `mult` (the *callee*) for its effect

`foo`'s view:



Function call and return

caller

```
// Computes 3 + 8 * 5
function foo 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
```

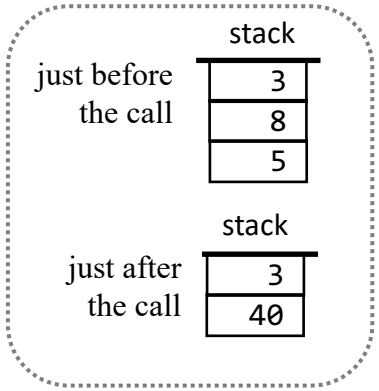
callee

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

Typical function-call-and-return scenario

Function `foo` (the *caller*) calls function `mult` (the *callee*) for its effect

`foo`'s view:



Magic!

Let's open the black box

Function call and return

caller

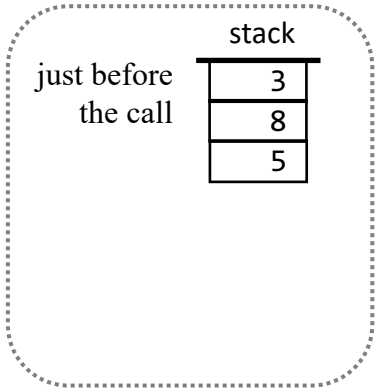
```
// Computes 3 + 8 * 5
function foo 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
```

callee

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

Adding line numbers, just for reference

foo's view:



The caller's execution is suspended

Typical function-call-and-return scenario

Function `foo` (the *caller*) calls function `mult` (the *callee*) for its effect

Function call and return

caller

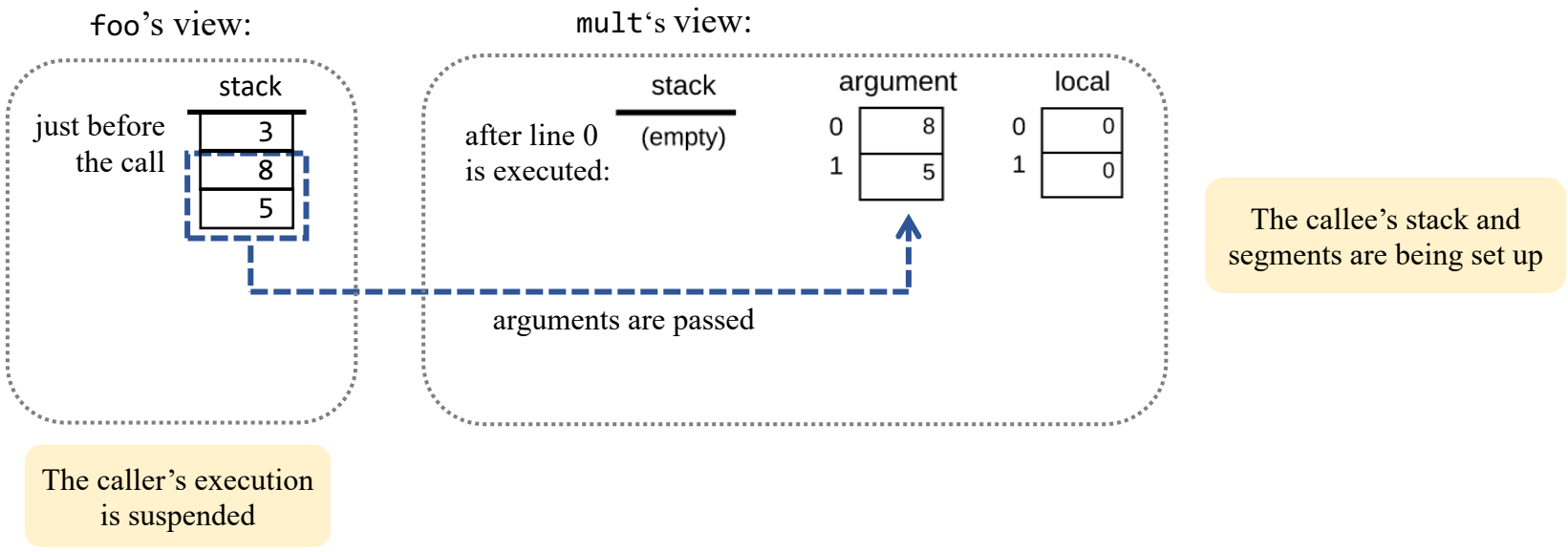
```
// Computes 3 + 8 * 5
function foo 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
```

callee

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

Typical function-call-and-return scenario

Function `foo` (the *caller*) calls function `mult` (the *callee*) for its effect



Function call and return

caller

```
// Computes 3 + 8 * 5
function foo 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  ...
```

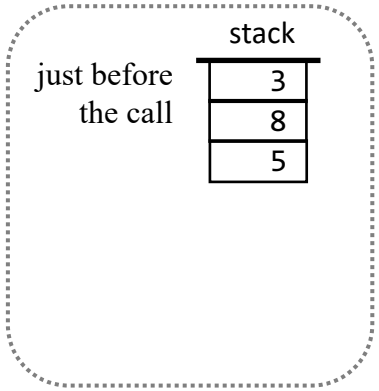
callee

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

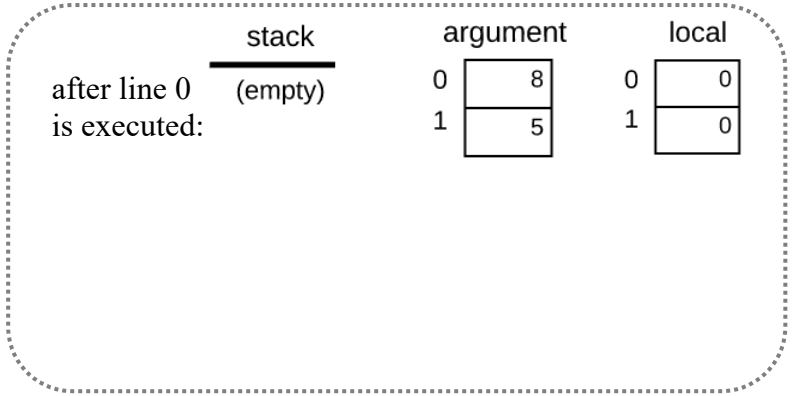
Typical function-call-and-return scenario

Function `foo` (the *caller*) calls function `mult` (the *callee*) for its effect

`foo`'s view:



`mult`'s view:



The callee's code is executed

Function call and return

caller

```
// Computes 3 + 8 * 5
function foo 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
```

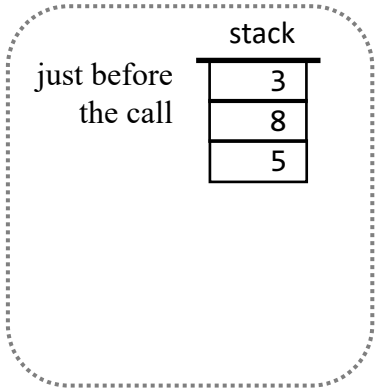
callee

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

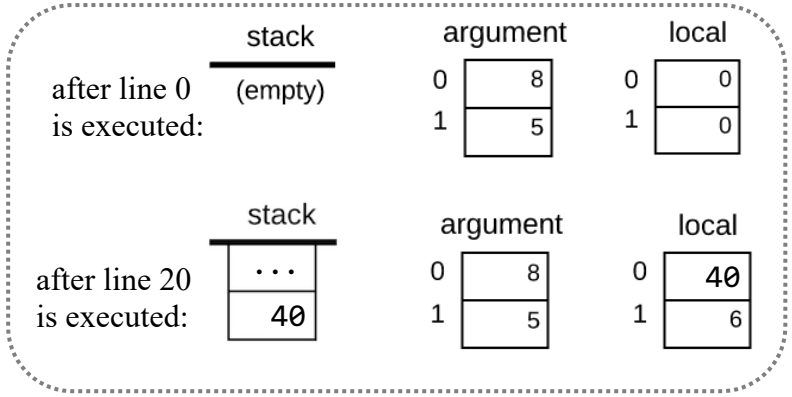
Typical function-call-and-return scenario

Function `foo` (the *caller*) calls function `mult` (the *callee*) for its effect

`foo`'s view:



`mult`'s view:



The callee's code is executed

Function call and return

caller

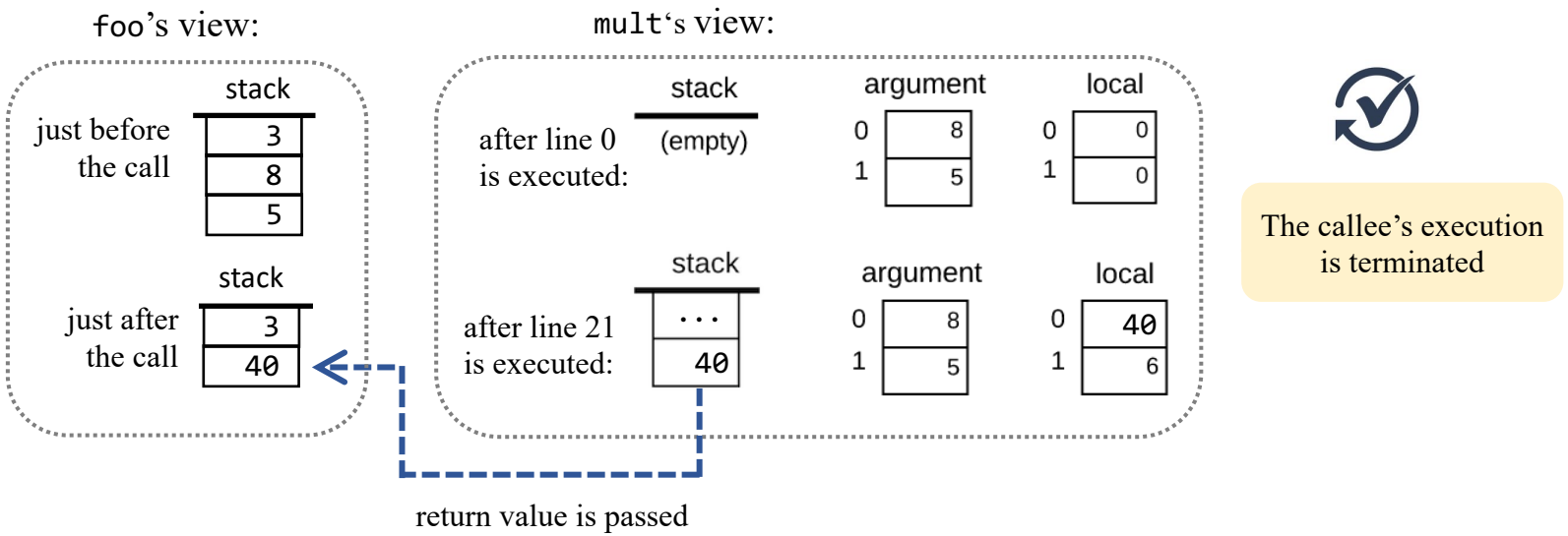
```
// Computes 3 + 8 * 5
function foo 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
```

callee

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

Typical function-call-and-return scenario

Function `foo` (the *caller*) calls function `mult` (the *callee*) for its effect



The callee's execution is terminated

Function call and return

caller

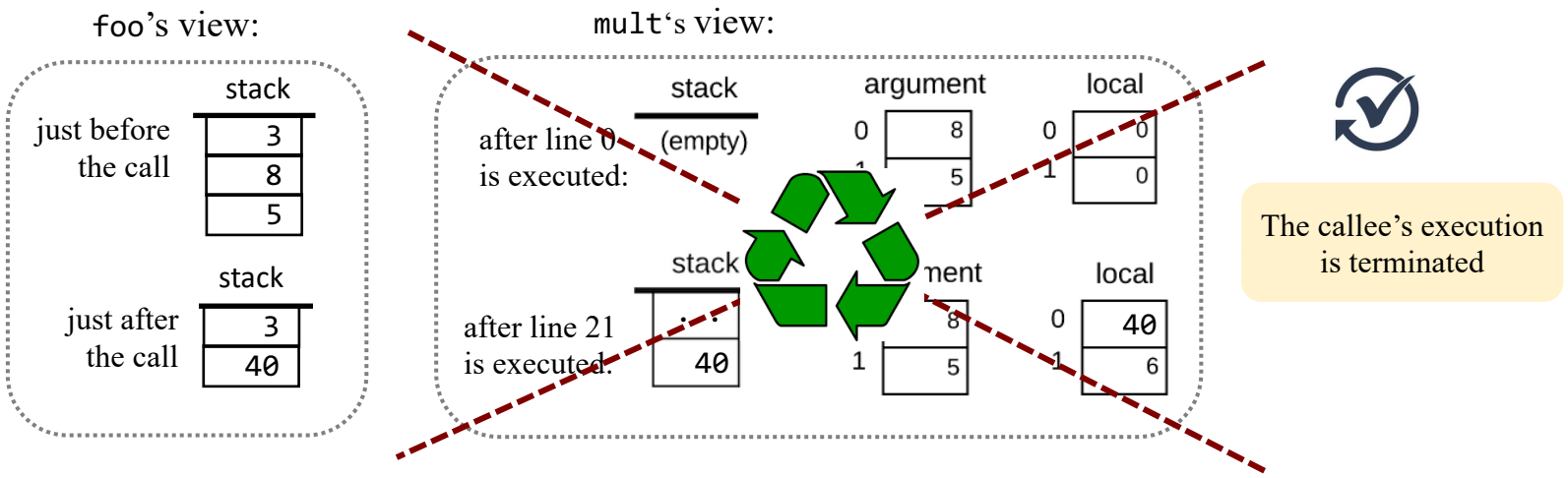
```
// Computes 3 + 8 * 5
function foo 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
```

callee

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

Typical function-call-and-return scenario

Function `foo` (the *caller*) calls function `mult` (the *callee*) for its effect



The callee's execution is terminated

Function call and return

caller

```
// Computes 3 + 8 * 5
function foo 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
```

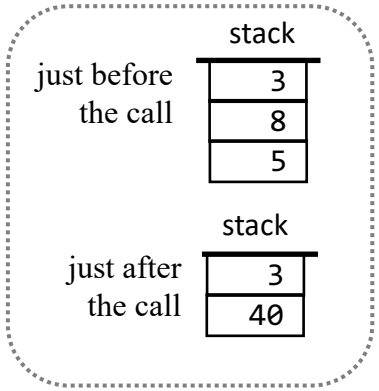
callee

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

Typical function-call-and-return scenario

Function `foo` (the *caller*) calls function `mult` (the *callee*) for its effect

`foo`'s view:



The caller's execution is resumed

Magic!

Function call and return

Recap

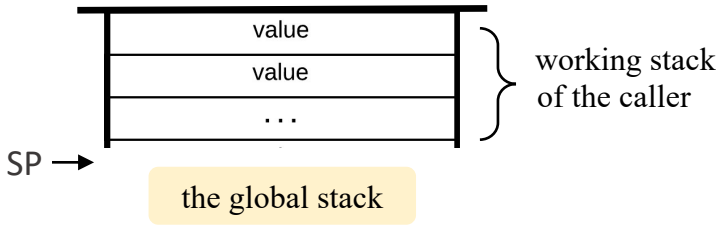
- A VM program consists of one or more VM functions
- The functions call each other, for their effect (perhaps recursively)
- Each function execution sees its own working stack and memory segments
- Arguments and return values are passed, somehow

Implementation

- We will build a run-time system that realizes this abstraction
- The run-time system will manage the working stacks and memory segments of all the running functions on a single, global stack, stored on the host RAM
- For every occurrence of `call`, `function`, and `return` command, the VM translator will generate machine language code that, when executed, will realize this run-time system
- And now for the gory details.

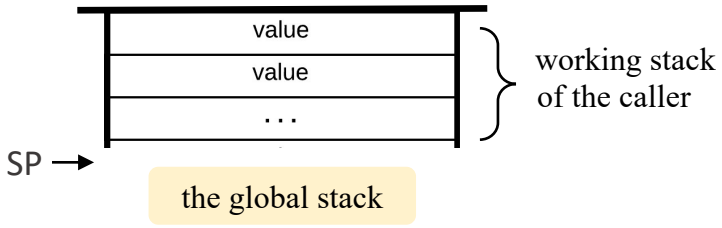
Implementing `call` / function / return

The caller is running,
doing various things



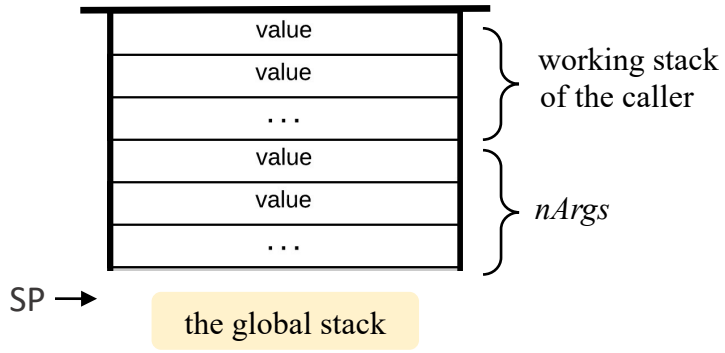
Implementing `call` / function / return

The caller prepares to call another function;
It pushes 0 or more arguments onto the stack



Implementing `call` / `function` / `return`

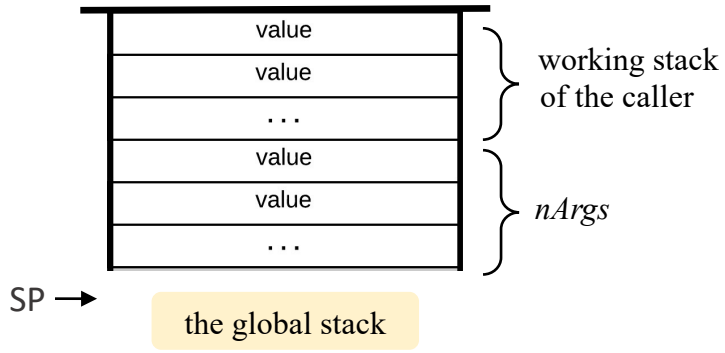
The caller prepares to call another function;
It pushes 0 or more arguments onto the stack



Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



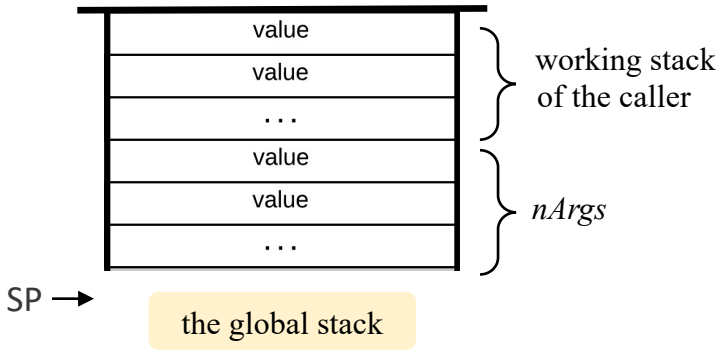
Handling `call` *functionName nArgs*

We have to:

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

We have to:

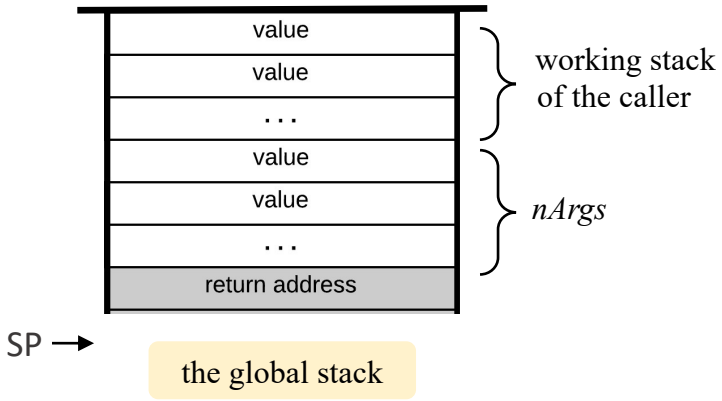
- Save the return address

The address to which control should return when the callee's execution is terminated

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

We have to:

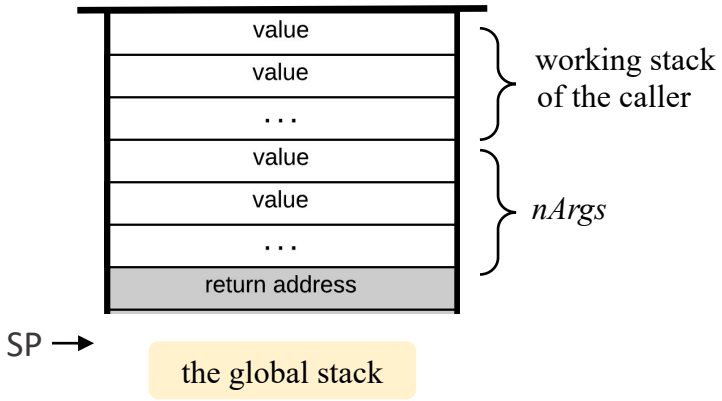
- Save the return address

The address to which control should return when the callee's execution is terminated

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

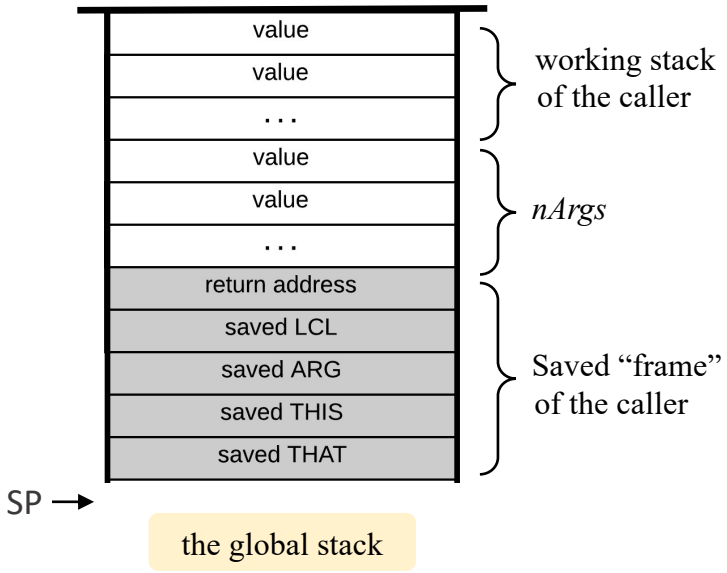
We have to:

- Save the return address
- Save the caller's segment pointers

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

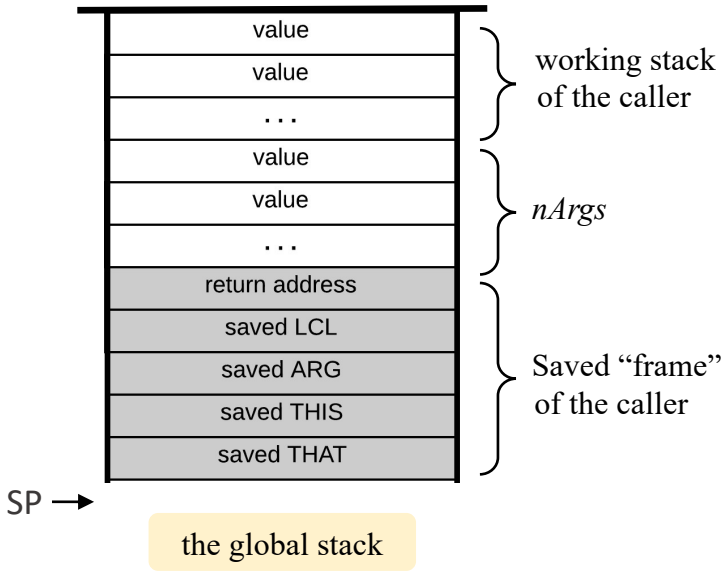
We have to:

- Save the return address
- Save the caller's segment pointers

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

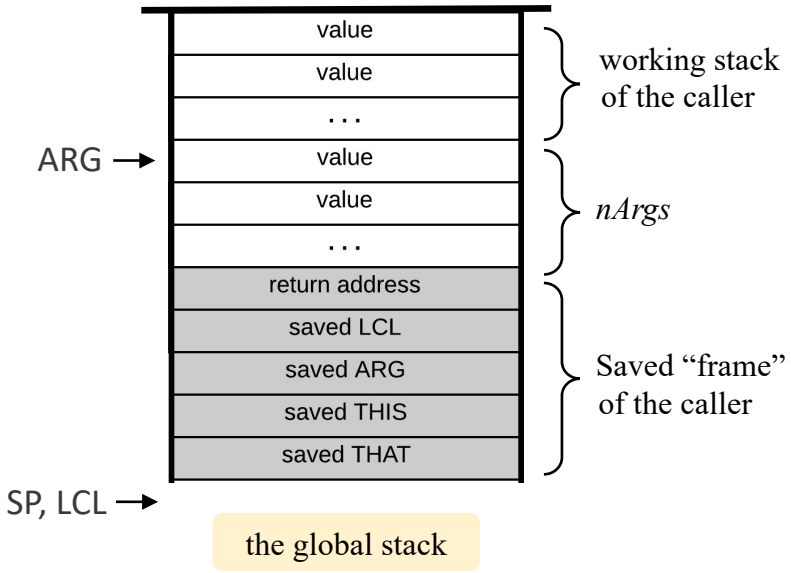
We have to:

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

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

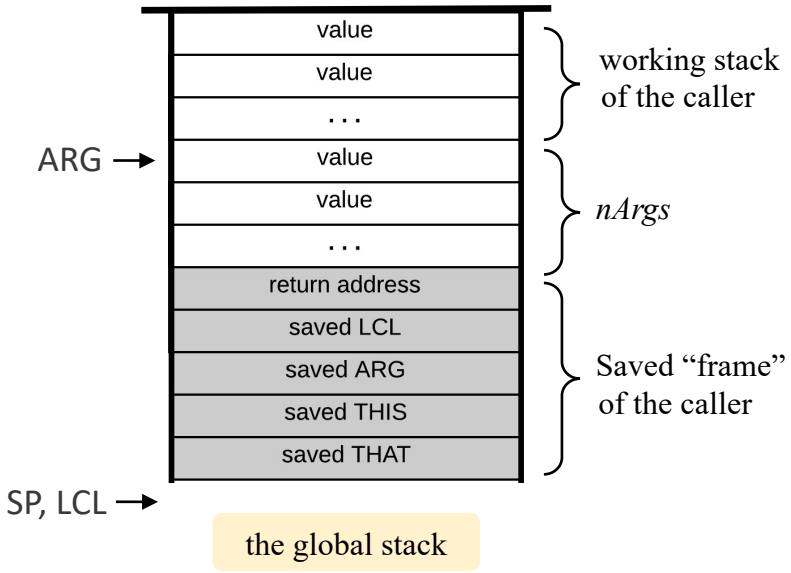
We have to:

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

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

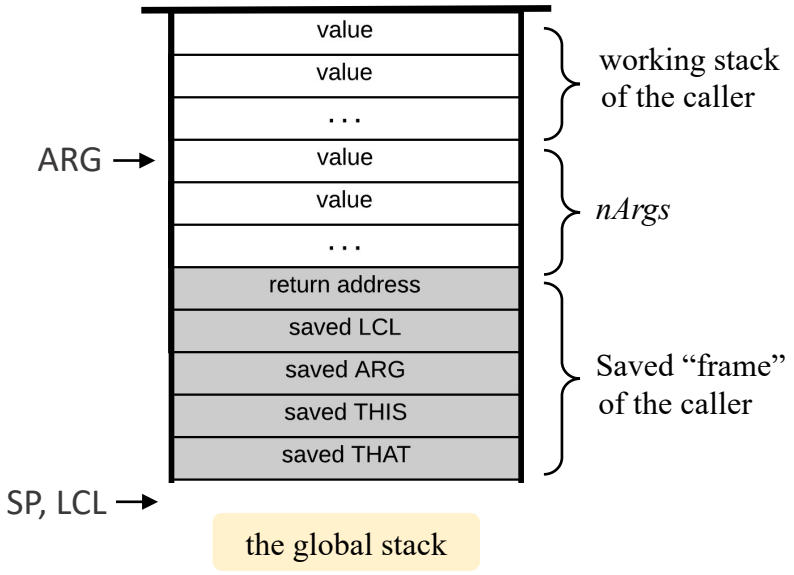
We have to:

- 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

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



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

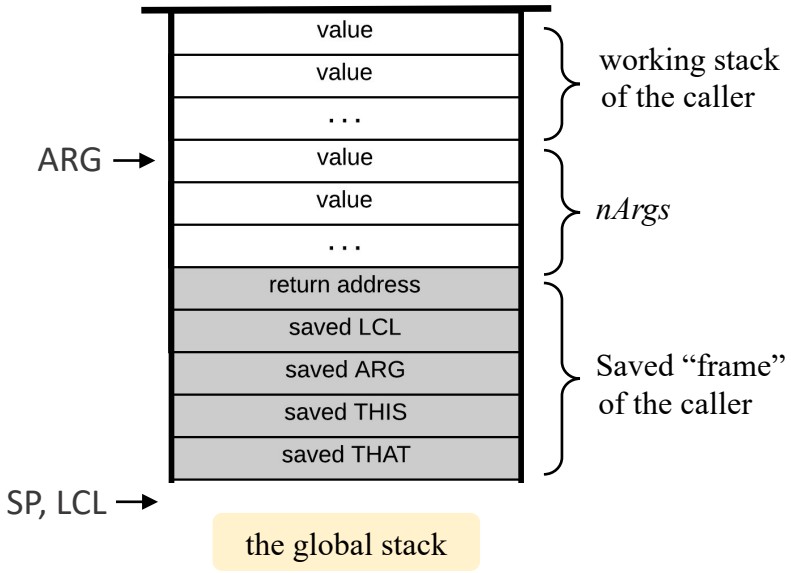
```
// call functionName nArgs
   push retAddrLabel     // Generates and pushes this label
   push LCL              // Saves the caller's LCL
   push ARG              // Saves the caller's ARG
   push THIS             // Saves the caller's THIS
   push THAT             // Saves the caller's THAT
   ARG = 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 should generate this code in assembly)

Implementing `call` / **`function`** / `return`

The callee is entered:

`function` *functionName nVars*



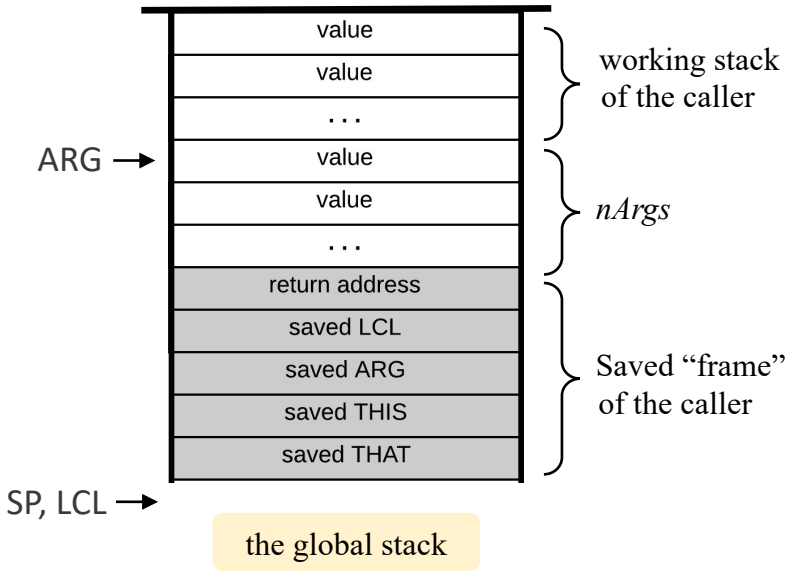
Handling `function` *functionName nVars*

We have to:

Implementing `call` / **`function`** / `return`

The callee is entered:

`function` *functionName nVars*



Handling `function` *functionName nVars*

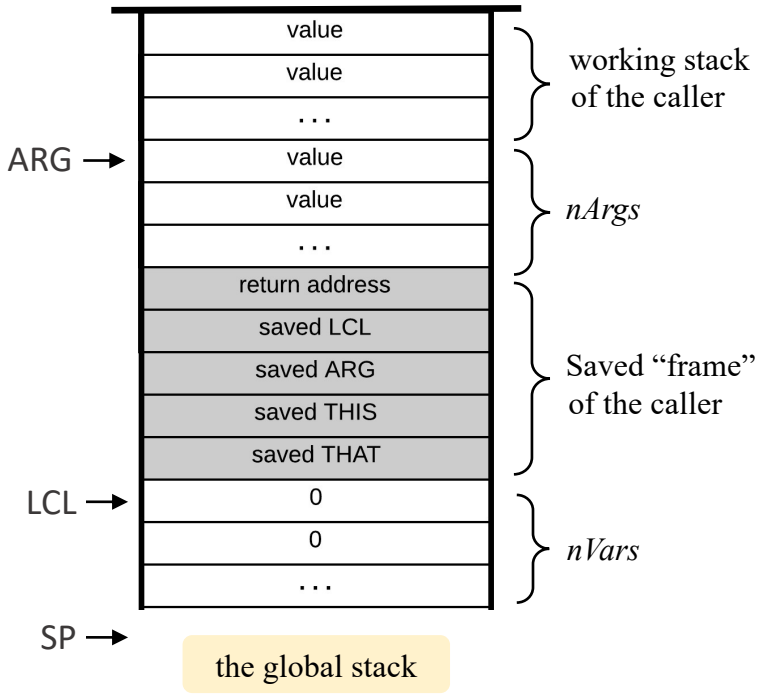
We have to:

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

Implementing `call` / **`function`** / `return`

The callee is entered:

`function` *functionName nVars*



Handling `function` *functionName nVars*

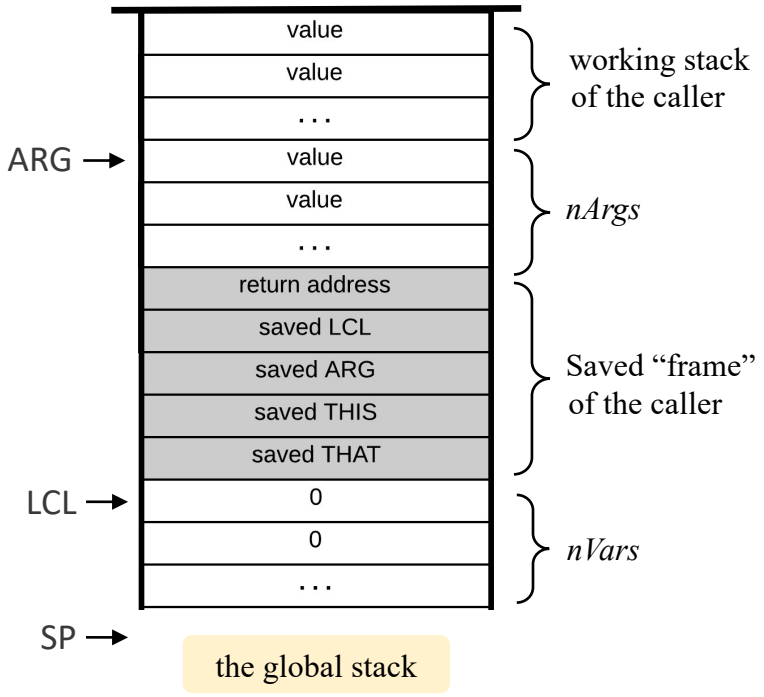
We have to:

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

Implementing `call` / **`function`** / `return`

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

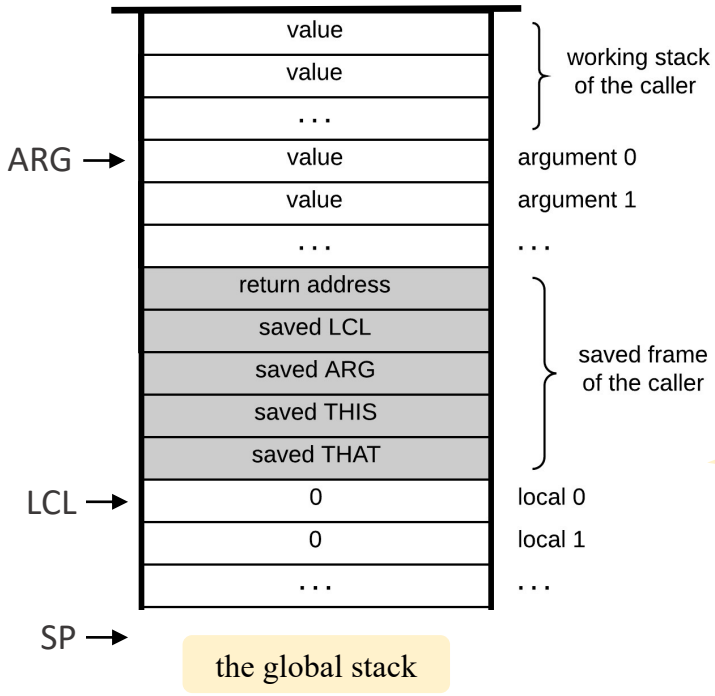
```
// function functionName nArgs
(functionName)          // function's entry point (injected label)
   // push nVars 0 values (initializes the local variables to 0)
   push 0
   ...
   push 0
```

(The VM translator should generate this code in assembly)

Implementing `call` / **`function`** / `return`

The callee is entered:

`function` *functionName nVars*

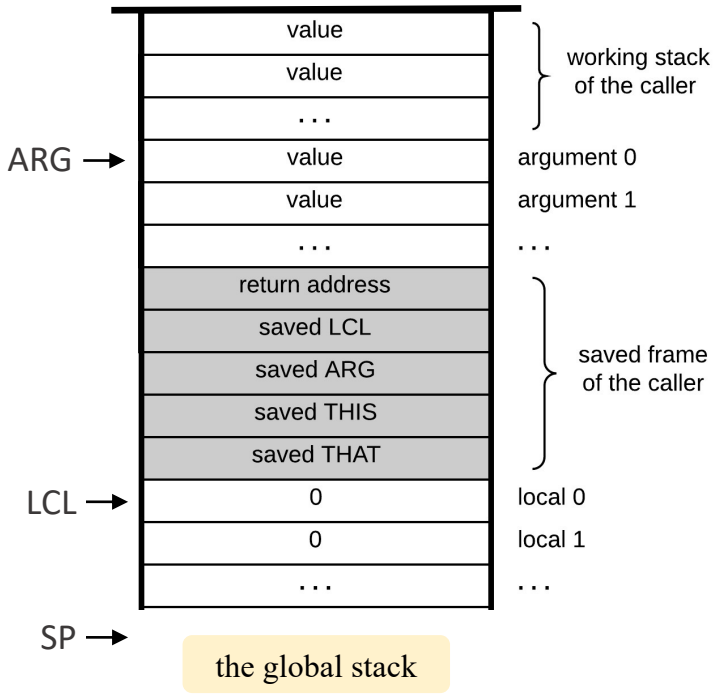


The callee is all set. It has:

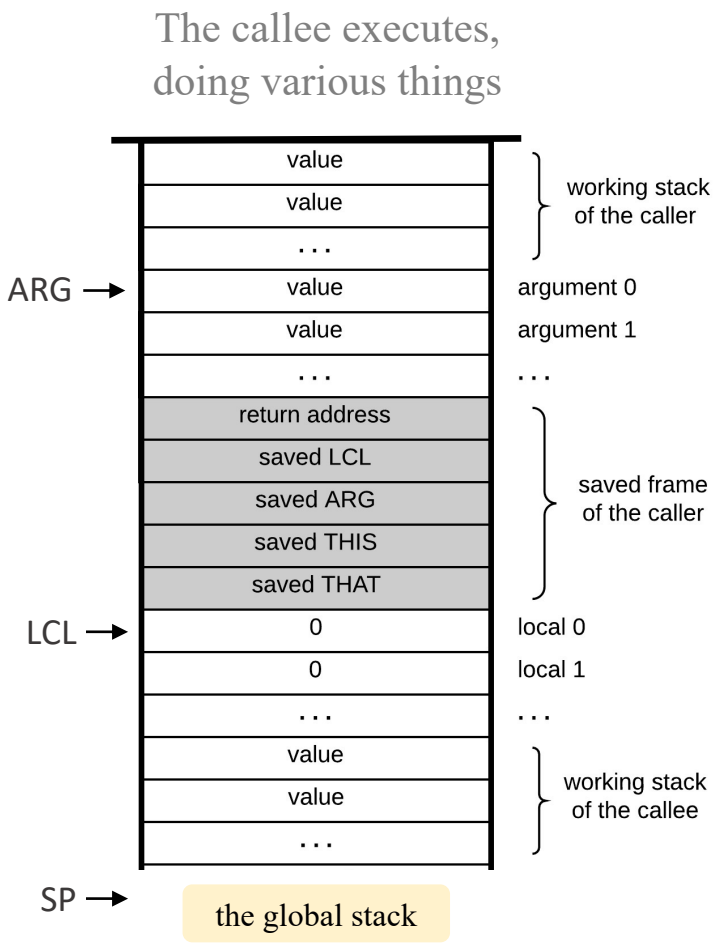
- Arguments (passed by the caller)
- Local variables (all set to 0)
- Working stack (empty)

Implementing `call / function / return`

The callee executes,
doing various things

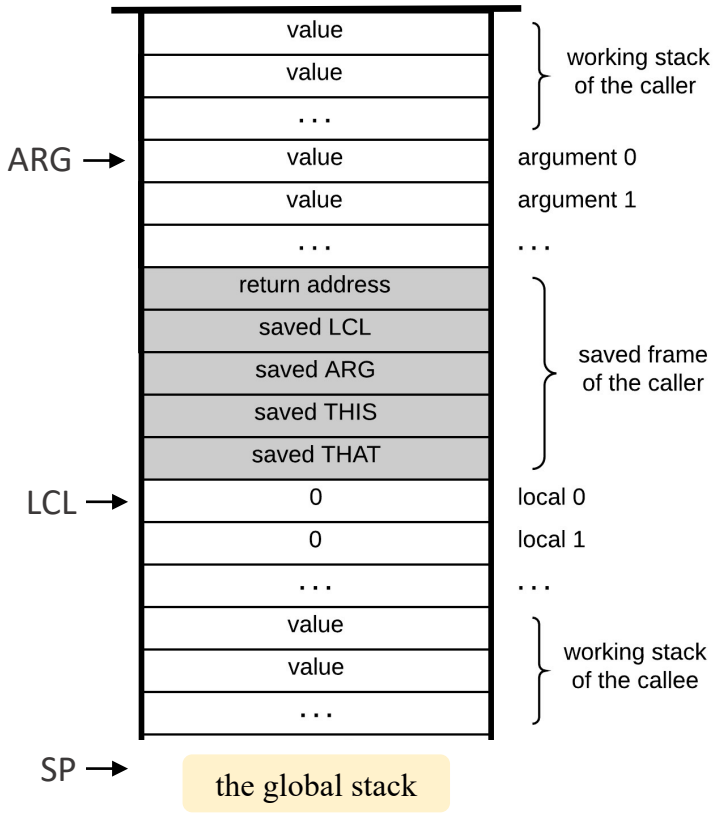


Implementing `call / function / return`



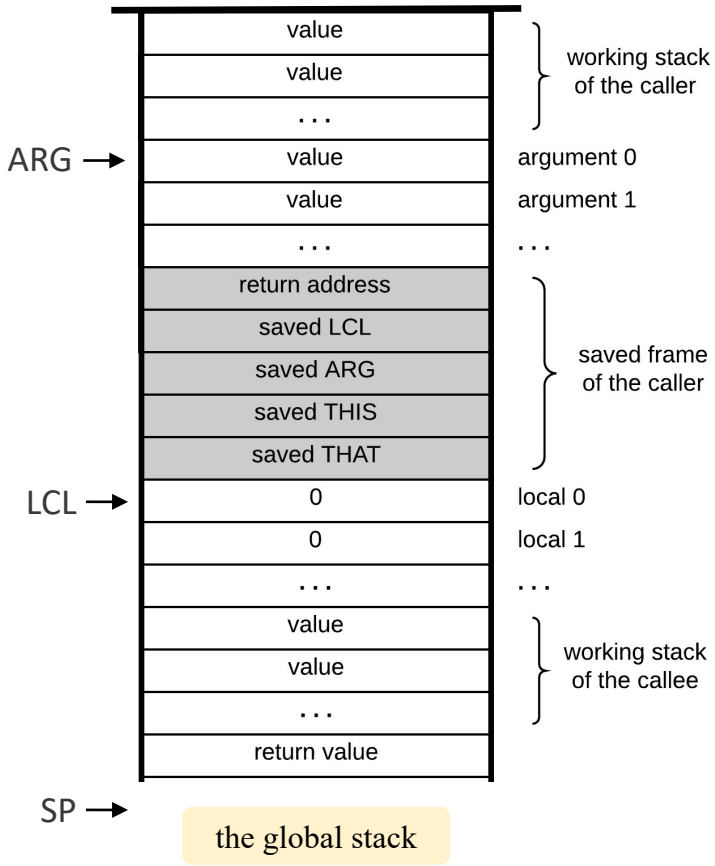
Implementing `call / function / return`

The callee prepares to return:
It pushes a *return value*



Implementing `call / function / return`

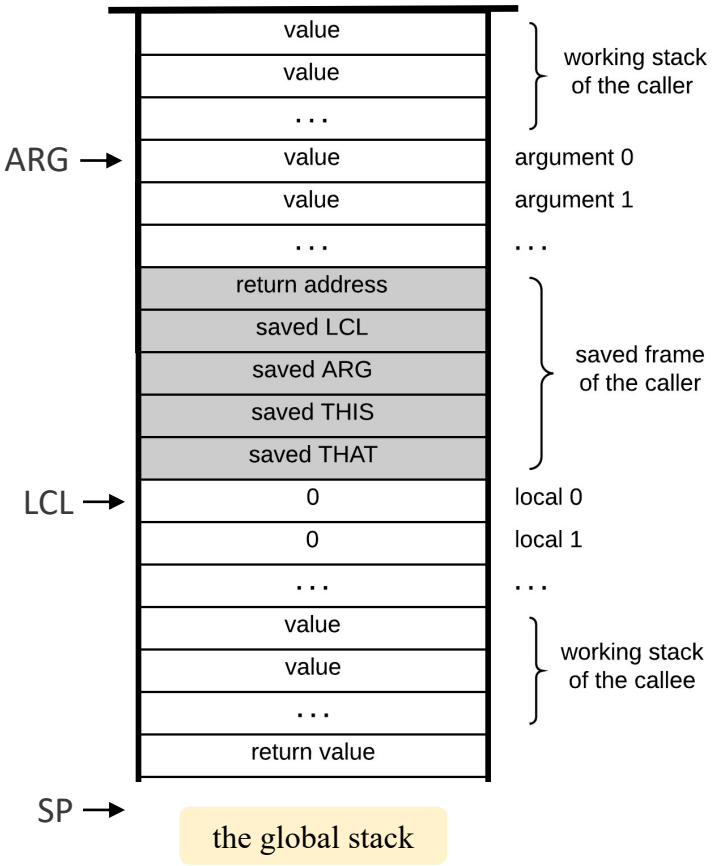
The callee prepares to return:
It pushes a *return value*



Implementing `call / function / return`

The callee says:

`return`



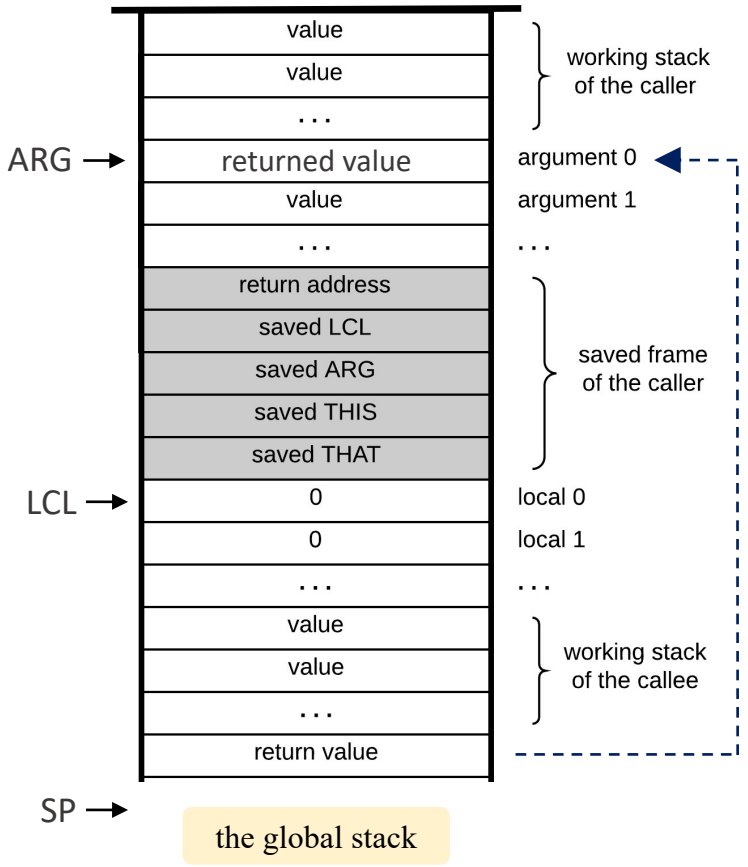
Handling `return`:

We have to:

Implementing `call / function / return`

The callee says:

`return`



Handling `return`:

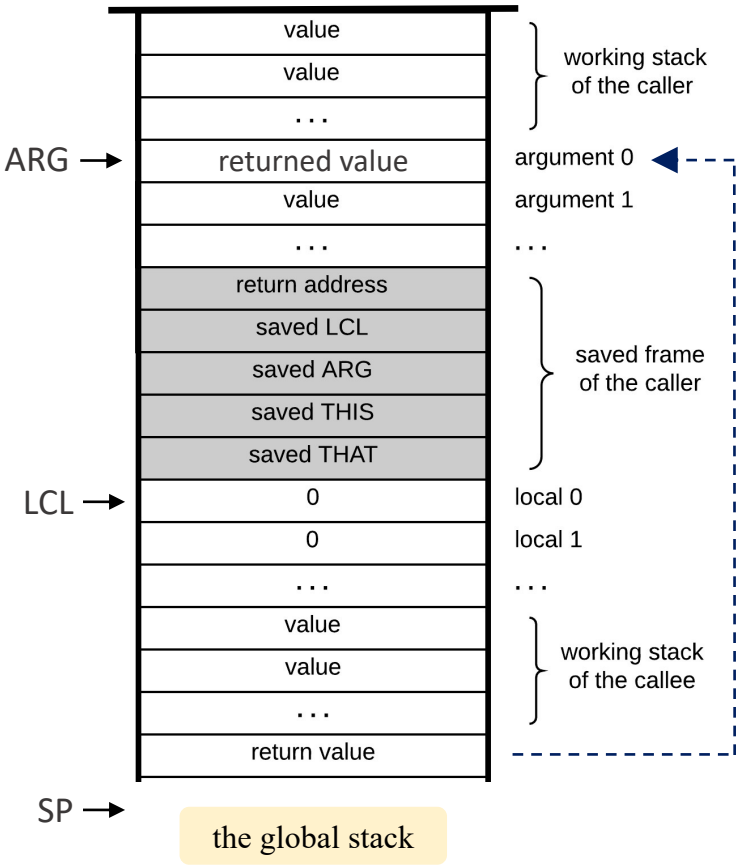
We have to:

1. Replace the arguments that the caller pushed with the value returned by the callee

Implementing `call / function / return`

The callee says:

`return`



Handling `return`:

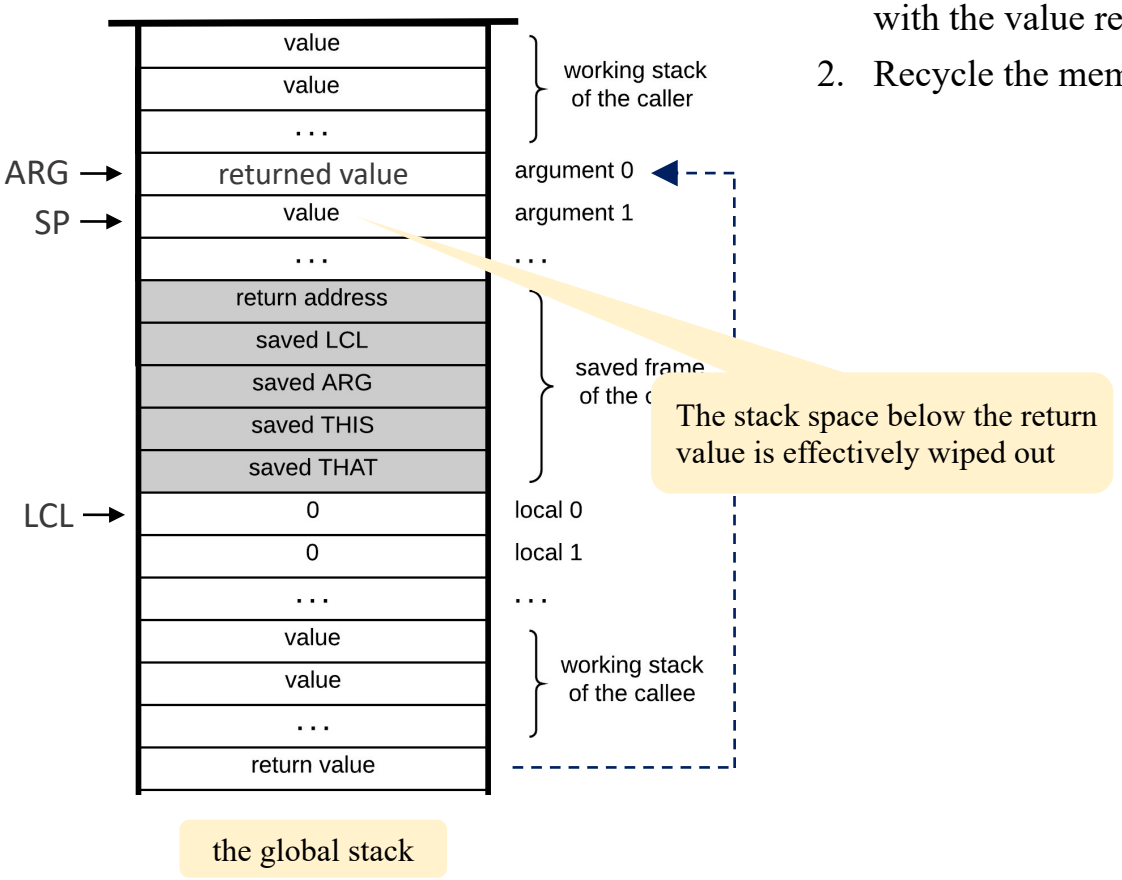
We have to:

1. Replace the arguments that the caller pushed with the value returned by the callee
2. Recycle the memory used by the callee

Implementing `call / function / return`

The callee says:

`return`



Handling `return`:

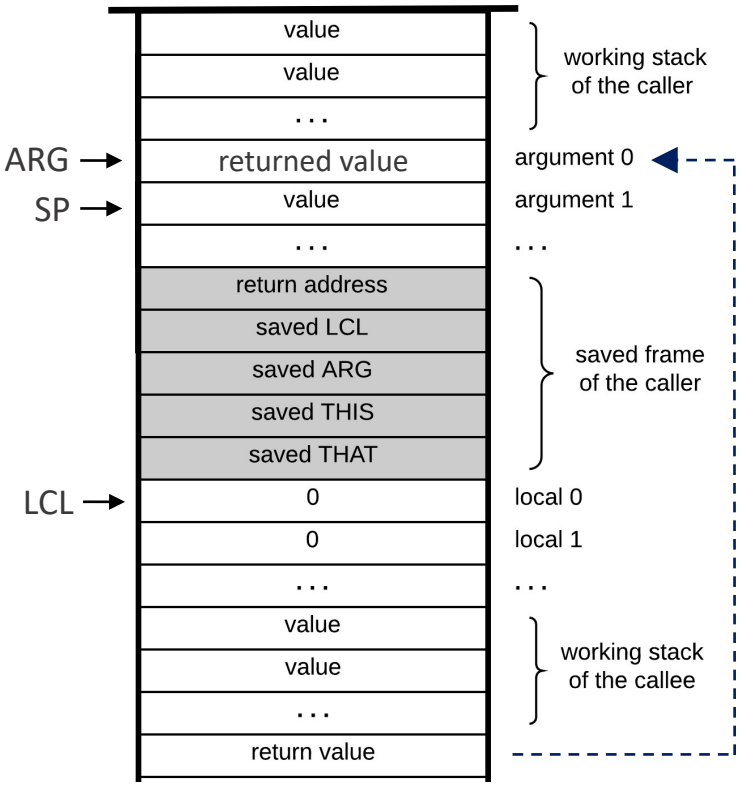
We have to:

1. Replace the arguments that the caller pushed with the value returned by the callee
2. Recycle the memory used by the callee

Implementing `call / function / return`

The callee says:

`return`



the global stack

Handling `return`:

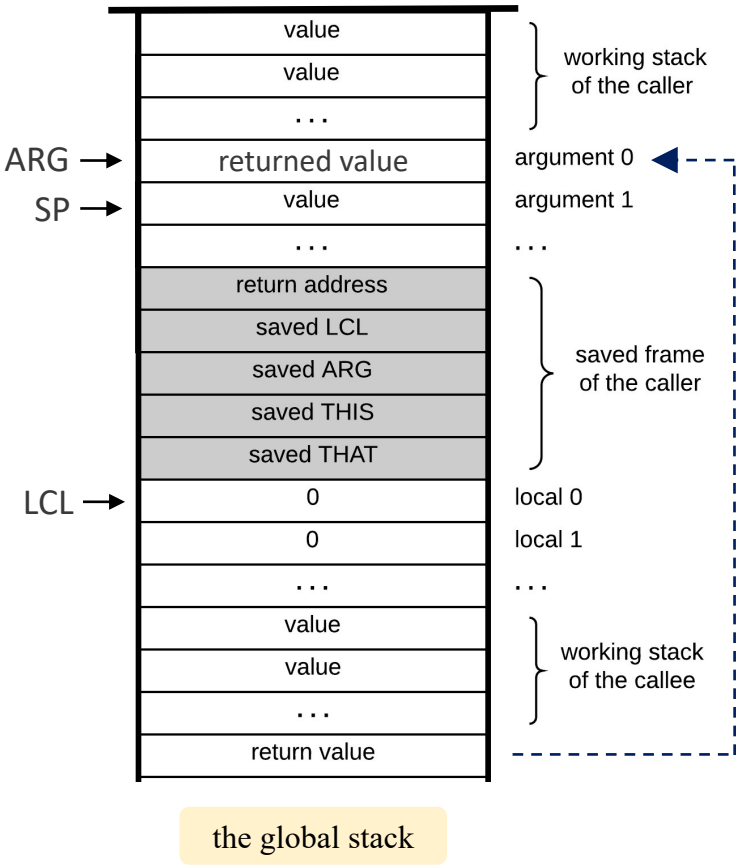
We have to:

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

Implementing `call / function / return`

The callee says:

`return`



the global stack

Handling `return`:

We have to:

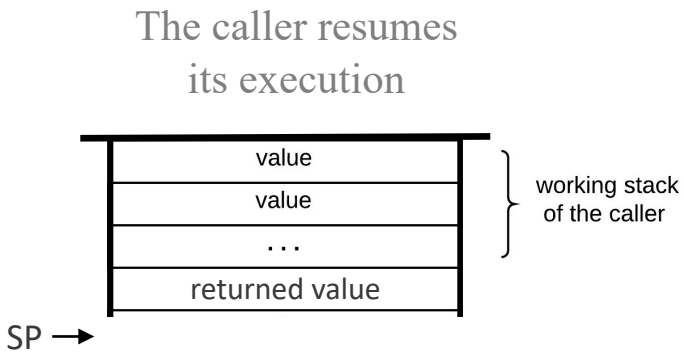
1. Replace the arguments that the caller pushed with the value returned by the callee
2. Recycle the memory used by the callee
3. Reinstate the caller's segment pointers
4. Jump to the return address

Generated code

```
// endFrame and retAddr are temporary variables.
// The pointer notation *addr denotes 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 should generate this code in assembly)

Implementing `call / function / return`

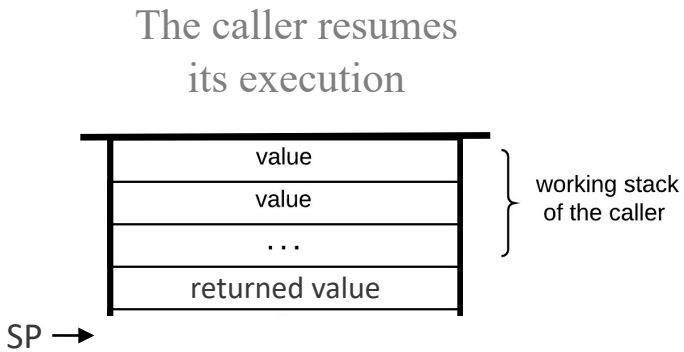


The caller's world is exactly the same as before the call, except that the arguments that it pushed before the call were replaced by the value returned by the callee.

Any sufficiently advanced technology is indistinguishable from magic.

– Arthur C. Clarke, 1962

Implementing `call / function / return`



What if the calling chain consists of multiple calls?

`foo` calls `bar`, then `bar` calls `baz`, etc.

And what about recursion?

Solution

Follows exactly the same scheme, once for every call-and-return scenario;

The global stack will grow and shrink as needed: *Last in, first out*

Now that is wisdom: In every instance of your labor, hitch your wagon to a star, and see your chore done by the gods themselves. – Ralph Waldo Emerson, 1870

Lecture plan

✓ Abstraction

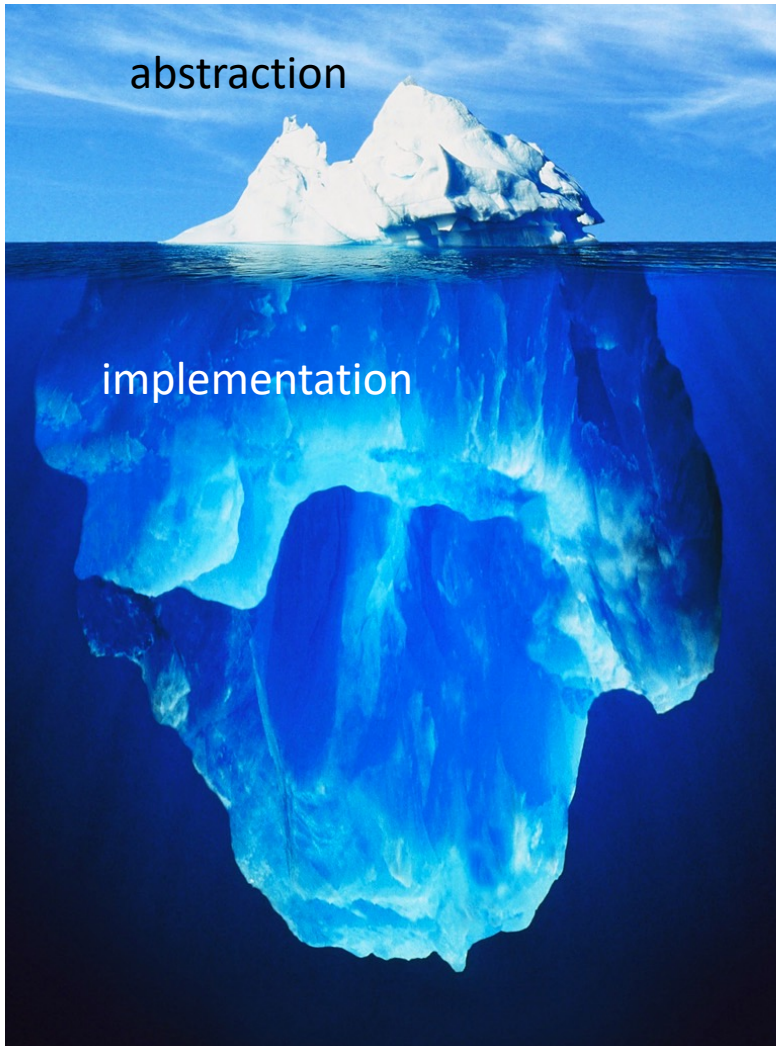
- VM branching commands
- VM function commands

✓ Implementation (conceptual)

- VM branching commands
- VM function commands

➡ Implementation (nuts and bolts)

- Run-time system
- Standard mapping
- VM translator
- Project 8



VM translator in action

VM code

```
function Foo.main 4
    ...
    // Computes 8 * 5 + 7
    push constant 8
    push constant 5
    call Foo.mult 2
    push constant 7
    add
    ...
// Returns arg 0 * arg 1
function Foo.mult 2
    push constant 0
    pop local 0
    push constant 1
    pop local 1
    ...
    // Returns the result
    push local 0
    return
```

VM translator

Generated code (in assembly)

VM translator in action

VM code

```
function Foo.main 4
    ...
    // Computes 8 * 5 + 7
    push constant 8
    push constant 5
    call Foo.mult 2
    push constant 7
    add
    ...
// Returns arg 0 * arg 1
function Foo.mult 2
    push constant 0
    pop local 0
    push constant 1
    pop local 1
    ...
    // Returns the result
    push local 0
    return
```

VM translator

Generated code (in assembly)

```
(Foo.main)  // function's entry point label, injected by the VM translator
    // assembly code that initializes the function's local variables
```

VM translator in action

VM code

```
function Foo.main 4
    ...
    // Computes 8 * 5 + 7
    push constant 8
    push constant 5
    call Foo.mult 2
    push constant 7
    add
    ...
// Returns arg 0 * arg 1
function Foo.mult 2
    push constant 0
    pop local 0
    push constant 1
    pop local 1
    ...
    // Returns the result
    push local 0
    return
```

VM translator

Generated code (in assembly)

```
(Foo.main)  // function's entry point label, injected by the VM translator
    // assembly code that initializes the function's local variables
```

VM translator in action

VM code

```
function Foo.main 4
    ...
    // Computes 8 * 5 + 7
    push constant 8
    push constant 5
    call Foo.mult 2
    push constant 7
    add
    ...
// Returns arg 0 * arg 1
function Foo.mult 2
    push constant 0
    pop local 0
    push constant 1
    pop local 1
    ...
    // Returns the result
    push local 0
    return
```

VM translator

Generated code (in assembly)

```
(Foo.main)  // function's entry point label, injected by the VM translator
    // assembly code that initializes the function's local variables
    ...
    // assembly code that handles  push constant 8
```

VM translator in action

VM code

```
function Foo.main 4
    ...
    // Computes 8 * 5 + 7
    push constant 8
    push constant 5
    call Foo.mult 2
    push constant 7
    add
    ...
// Returns arg 0 * arg 1
function Foo.mult 2
    push constant 0
    pop local 0
    push constant 1
    pop local 1
    ...
    // Returns the result
    push local 0
    return
```

VM translator

Generated code (in assembly)

```
(Foo.main)  // function's entry point label, injected by the VM translator
    // assembly code that initializes the function's local variables
    ...
    // assembly code that handles  push constant 8
```

VM translator in action

VM code

```
function Foo.main 4
    ...
    // Computes 8 * 5 + 7
    push constant 8
    push constant 5
    call Foo.mult 2
    push constant 7
    add
    ...
// Returns arg 0 * arg 1
function Foo.mult 2
    push constant 0
    pop local 0
    push constant 1
    pop local 1
    ...
    // Returns the result
    push local 0
    return
```

VM translator

Generated code (in assembly)

```
(Foo.main)  // function's entry point label, injected by the VM translator
    // assembly code that initializes the function's local variables
    ...
    // assembly code that handles  push constant 8
    // assembly code that handles  push constant 5
```

VM translator in action

VM code

```
function Foo.main 4
   ...
   // Computes 8 * 5 + 7
   push constant 8
   push constant 5
   call Foo.mult 2
   push constant 7
   add
   ...
// Returns arg 0 * arg 1
function Foo.mult 2
   push constant 0
   pop local 0
   push constant 1
   pop local 1
   ...
   // Returns the result
   push local 0
   return
```

VM translator

Generated code (in assembly)

```
(Foo.main)  // function's entry point label, injected by the VM translator
   // assembly code that initializes the function's local variables
   ...
   // assembly code that handles push constant 8
   // assembly code that handles push constant 5
   // assembly code that handles call Foo.mult 2: creates and saves a return
   // address label (Foo$ret.1), saves the segment pointers, and then generates:
   goto Foo.mult  // (in assembly)
(Foo$ret.1)          // return address label, injected by the VM translator
   // assembly code that handles push constant 7
   ...
(Foo.mult)   // function's entry point label, injected by the VM translator
   // assembly code that initializes the function's local variables.
```

VM translator in action

VM code

```
function Foo.main 4
   ...
   // Computes 8 * 5 + 7
   push constant 8
   push constant 5
   call Foo.mult 2
   push constant 7
   add
   ...
// Returns arg 0 * arg 1
function Foo.mult 2
   push constant 0
   pop local 0
   push constant 1
   pop local 1
   ...
   // Returns the result
   push local 0
   return
```

VM translator

Generated code (in assembly)

```
(Foo.main)  // function's entry point label, injected by the VM translator
   // assembly code that initializes the function's local variables
   ...
   // assembly code that handles push constant 8
   // assembly code that handles push constant 5
   // assembly code that handles call Foo.mult 2: creates and saves a return
   // address label (Foo$ret.1), saves the segment pointers, and then generates:
   goto Foo.mult  // (in assembly)
(Foo$ret.1)          // return address label, injected by the VM translator
   // assembly code that handles push constant 7
   ...
(Foo.mult)   // function's entry point label, injected by the VM translator
   // assembly code that initializes the function's local variables.
   ...
   // Assembly code that handles push local 0
```

VM translator in action

VM code

```
function Foo.main 4
   ...
   // Computes 8 * 5 + 7
   push constant 8
   push constant 5
   call Foo.mult 2
   push constant 7
   add
   ...
// Returns arg 0 * arg 1
function Foo.mult 2
   push constant 0
   pop local 0
   push constant 1
   pop local 1
   ...
   // Returns the result
   push local 0
   return
```

VM translator

Generated code (in assembly)

```
(Foo.main)  // function's entry point label, injected by the VM translator
   // assembly code that initializes the function's local variables
   ...
   // assembly code that handles push constant 8
   // assembly code that handles push constant 5
   // assembly code that handles call Foo.mult 2: creates and saves a return
   // address label (Foo$ret.1), saves the segment pointers, and then generates:
   goto Foo.mult  // (in assembly)
(Foo$ret.1)        // return address label, injected by the VM translator
   // assembly code that handles push constant 7
   ...

(Foo.mult)   // function's entry point label, injected by the VM translator
   // assembly code that initializes the function's local variables.
   ...
   // Assembly code that handles push local 0
   // Assembly code that handles return: gets the return address (which happens
   // to be Foo$ret.1), copies the return value, reinstates the segment pointers
   // of Foo.main, and then generates:
   goto Foo$ret.1     // (in assembly)
```

VM translator in action

The translation creates a run-time system

When executed, the generated code realizes the program semantics, as well as the function call-and-return overhead (passing arguments and return values, saving and reinstating function frames).

Notice: At the generated code level, functions no longer exist: The generated code is one long sequence of assembly instructions.

Generated code (in assembly)

```
(Foo.main)  // function's entry point label, injected by the VM translator
    // assembly code that initializes the function's local variables
    ...
    // assembly code that handles push constant 8
    // assembly code that handles push constant 5
    // assembly code that handles call Foo.mult 2: creates and saves a return
    // address label (Foo$ret.1), saves the segment pointers, and then generates:
    goto Foo.mult  // (in assembly)
(Foo$ret.1)          // return address label, injected by the VM translator
    // assembly code that handles push constant 7
    ...

(Foo.mult)    // function's entry point label, injected by the VM translator
    // assembly code that initializes the function's local variables.
    ...
    // Assembly code that handles push local 0
    // Assembly code that handles return: gets the return address (which happens
    // to be Foo$ret.1), copies the return value, reinstates the segment pointers
    // of Foo.main, and then generates:
    goto Foo$ret.1      // (in assembly)
```

Lecture plan

✓ Abstraction

- VM branching commands
- VM function commands

✓ Implementation (conceptual)

- VM branching commands
- VM function commands

Implementation (nuts and bolts)

- Run-time system
- Standard mapping
- VM translator
- Project 8

The big picture: Compilation

`myProg` folder

`Main.jack`

```
class Main
  function main {...}
  method bar {...}
}
```

`Foo.jack`

```
class Foo
  constructor new {...}
  method bar {...}
}
```

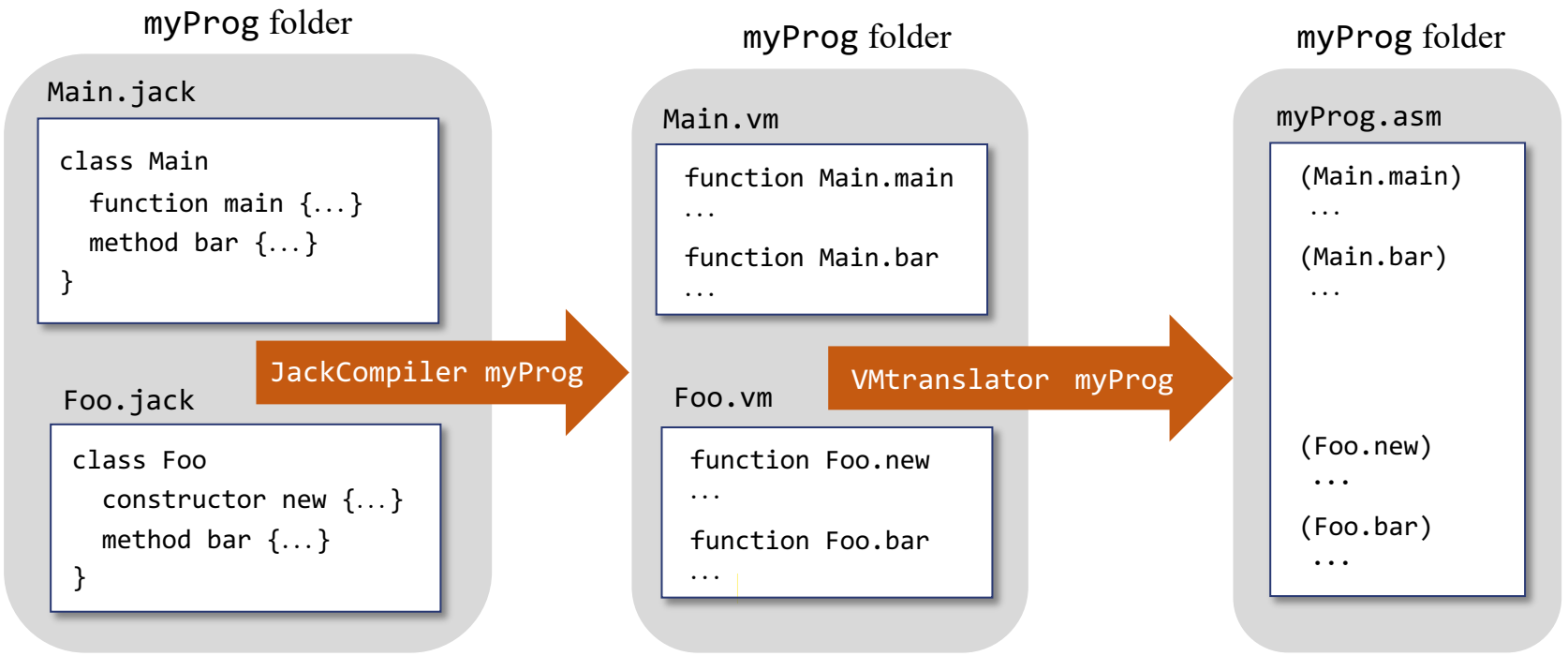
Each *Jack class* is a set of *constructors*, *methods*, and *functions* (static methods)

Jack program:

One or more class files, stored in the same folder;

The program name is taken to be the folder's name.

The big picture: Compilation



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

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

All the VM functions are translated into a single assembly file

Program conventions

One class file in any Jack program is expected to be named `Main.Jack`

The `Main.jack` file is expected to have at least one function, named `main`

Bootstrap code

Run-time conventions (of Jack programs on the Hack platform)

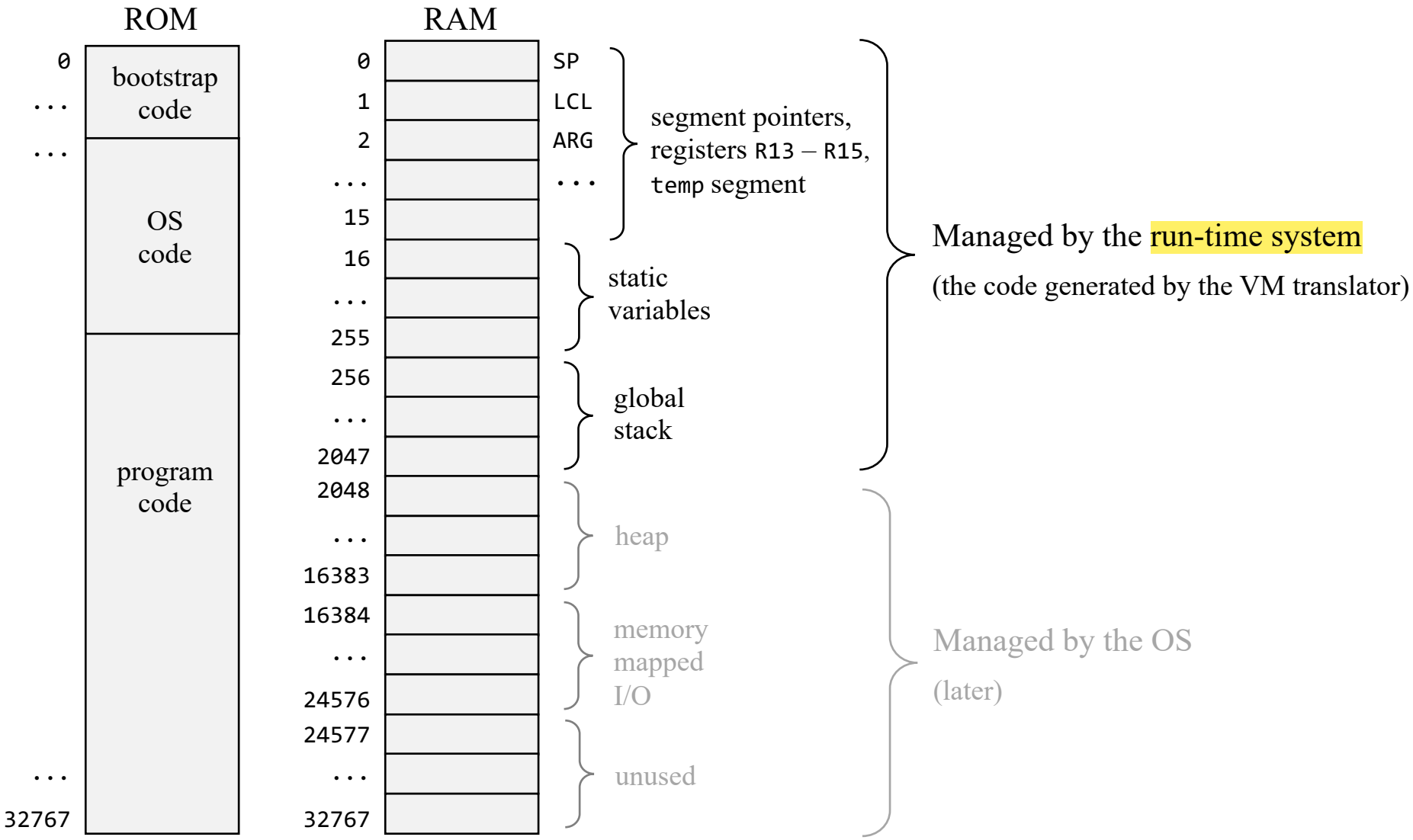
- The *stack* is stored in the RAM, starting at address 256
- In addition to the program functions, there should be an OS function named `Sys.init`
- `Sys.init` initializes the operating system, calls `Main.main`, and enters an infinite loop.

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

(The VM translator should generate this code in assembly)

This code should be stored in the Hack ROM, starting at address 0.

Standard mapping of the VM on the Hack platform



Standard mapping of the VM on the Hack platform

Symbol	*Usage*
SP	This predefined symbol points to the memory address within the host RAM just following the address containing the topmost stack value.
LCL, ARG, THIS, THAT	These predefined symbols point, respectively, to the base RAM addresses of the virtual segments `local`, `argument`, `this`, and `that` of the currently running VM function.
`Xxx.`*i* symbols (represent static variables)	Each reference to `static` i appearing in file `Xxx.vm` is translated to the assembly symbol `Xxx.`*i*. In the subsequent assembly process, the Hack assembler will allocate these symbolic variables to the RAM, starting at address 16.

Implemented in project 7

Standard mapping of the VM on the Hack platform

Symbol	*Usage*
SP	This predefined symbol points to the memory address within the host RAM just following the address containing the topmost stack value.
LCL, ARG, THIS, THAT	These predefined symbols point, respectively, to the base RAM addresses of the virtual segments `local`, `argument`, `this`, and `that` of the currently running VM function.
`Xxx.`*i* symbols (represent static variables)	Each reference to `static` i appearing in file `Xxx.vm` is translated to the assembly symbol `Xxx.`*i*. In the subsequent assembly process, the Hack assembler will allocate these symbolic variables to the RAM, starting at address 16.
functionName `$`*label* (destinations of `goto` commands)	Let `foo` be a function within the file `Xxx.vm`. The handling of each `label bar` command within `foo` generates, and injects into the assembly code stream, the symbol `Xxx.foo$bar`. When translating `goto bar` and `if-goto bar` commands (within `foo`) into assembly, the label `Xxx.foo$bar` must be used instead of `bar`.
functionName (function entry point symbols)	The handling of each `function foo` command within the file `Xxx.vm` generates, and injects into the assembly code stream, a symbol `Xxx.foo` that labels the entry-point to the function's code. In the subsequent assembly process, the assembler translates this symbol into the physical address where the function code starts.
functionName `$ret.`*i* (return address symbols)	Let `foo` be a function within the file `Xxx.vm`. The handling of each `call` command within `foo`'s code generates, and injects into the assembly code stream, a symbol `Xxx.foo$ret.`*i*, where *i* is a running integer (one such symbol is generated for each `call` command within `foo`). This symbol is used to mark the return address within the caller's code. In the subsequent assembly process, the assembler translates this symbol into the physical memory address of the command immediately following the `call` command.
R13 - R15	These predefined symbols can be used for any purpose. For example, if the VM translator generates assembly code that needs to use some low-level variables for temporary storage, R13 - R15 can come handy.

Complete specification of all the special symbols that the VM translator must use, and generate.

Read carefully, for project 8.

Lecture plan

✓ Abstraction

- VM branching commands
- VM function commands

✓ Implementation (conceptual)

- VM branching commands
- VM function commands

Implementation (nuts and bolts)

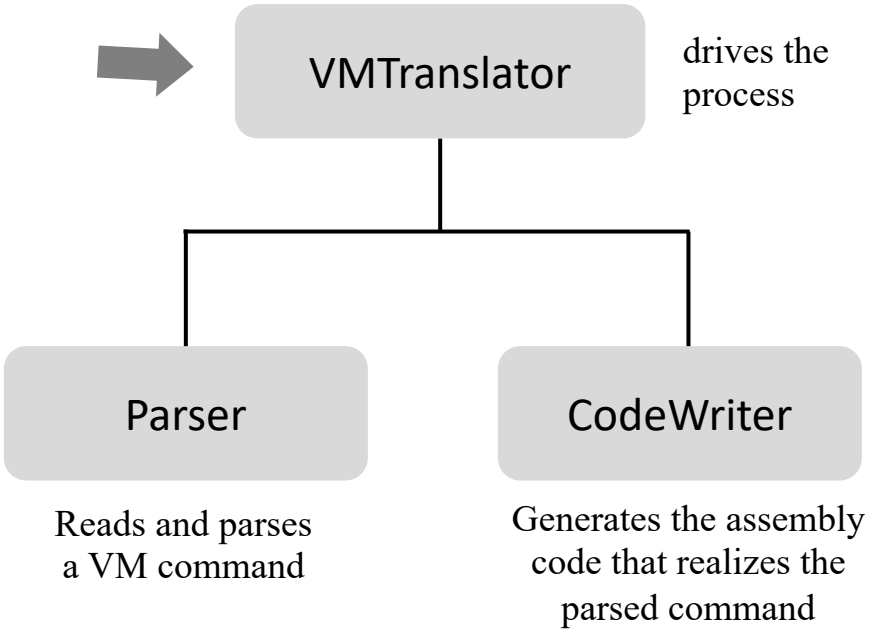
✓ Run-time system

✓ Standard mapping

➡ VM translator

- Project 8

VM translator: Proposed design



Each module extends the corresponding module developed in project 7,

Adding the implementation of the *branching* and *function* commands.

VMTranslator

Usage: `% VMTranslator` *input*

Where *input* is either a single *fileName*`.vm`, or a *folderName* containing one or more `.vm` files;

Output: A single assembly file named *input*`.asm`

Process

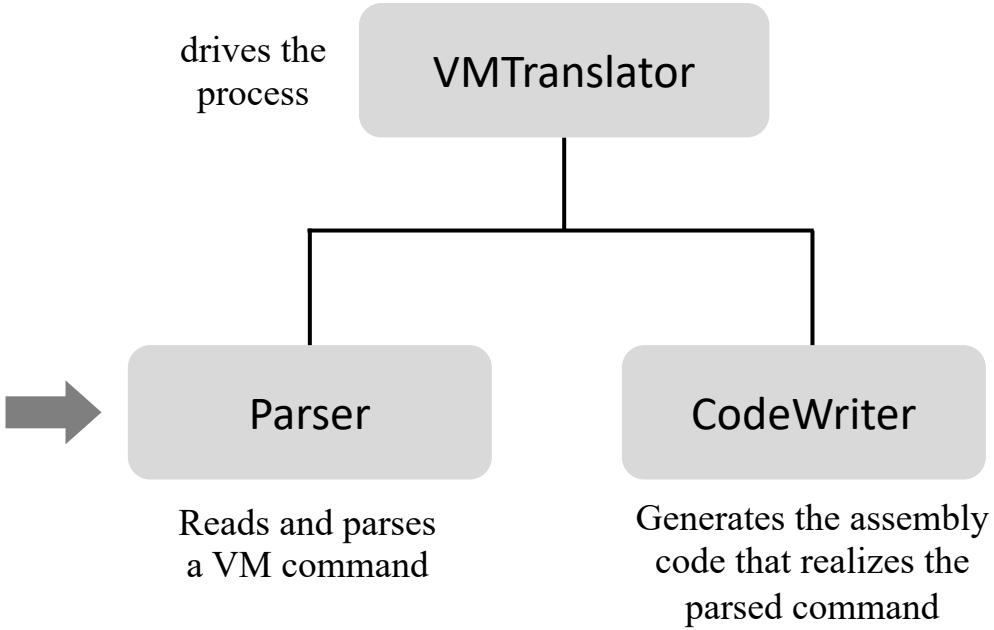
- Constructs a `CodeWriter`
- If the input is a `.vm` file:

  Constructs a `Parser` to handle the input file;

  For each VM command in the input file:
  uses the `Parser` to parse the command,
  uses the `CodeWriter` to generate assembly code from it
- If the input is a folder:

  Handles every `.vm` file in the folder in the manner described above.

VM translator: Proposed design



Each module extends the corresponding module developed in project 7,

Adding the implementation of the *branching* and *function* commands.

Parser

The same `Parser` developed in project 7:

- Handles the parsing of a single `.vm` file
- Reads a VM command, parses the command into its lexical components, and provides convenient access to these components
- Skips white space and comments.

Parser

Routine	*Arguments*	*Returns*	*Function*
constructor	input file / stream	—	Opens the input file/stream, and gets ready to parse it.
`hasMoreLines`	—	boolean	Are there more lines in the input?
`advance`	—	—	Reads the next command from the input and makes it the *current command*. This method should be called only if `hasMoreLines` is true. Initially there is no current command.
`commandType`	—	`C_ARITHMETIC`, `C_PUSH`, `C_POP`, `C_LABEL`, `C_GOTO`, `C_IF`, `C_FUNCTION`, `C_RETURN`, `C_CALL` (constant)	Returns a constant representing the type of the current command. If the current command is an arithmetic-logical command, returns `C_ARITHMETIC`.
`arg1`	—	string	Returns the first argument of the current command. In the case of `C_ARITHMETIC`, the command itself (`add`, `sub`, etc.) is returned. Should not be called if the current command is `C_RETURN`.
`arg2`	—	int	Returns the second argument of the current command. Should be called only if the current command is `C_PUSH`, `C_POP`, `C_FUNCTION`, or `C_CALL`.

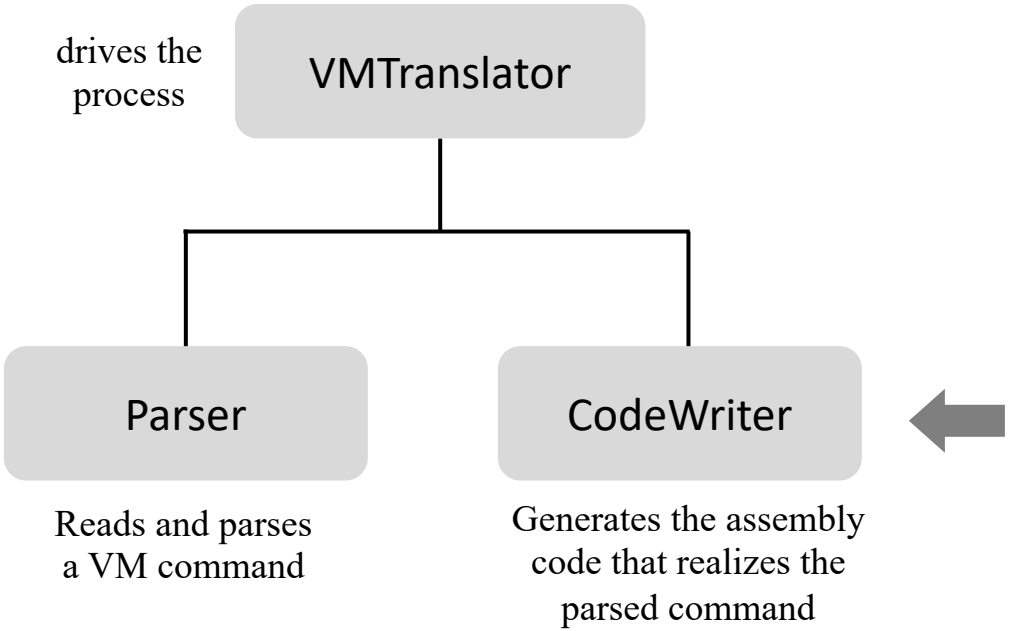
Same API as in project 7;

If your project 7 `Parser` did not handle the parsing of the VM commands:

`goto`, `if-goto`, `label`, `call`, `function`, and `return`,

add this parsing functionality now.

VM translator: Proposed design



Each module extends the corresponding module developed in project 7,

Adding the implementation of the *branching* and *function* commands.

CodeWriter

Routine	*Arguments*	*Returns*	*Function*
constructor	output file / stream	—	Opens an output file / stream and gets ready to write into it. Writes the assembly instructions that effect the bootstrap code that starts the program's execution. This code must be placed at the beginning of the generated output file / stream.
`setFileName`	`fileName` (string)	—	Informs that the translation of a new VM file has started (called by the `VMTranslator`).
`writeArithmetic` (developed in project 7)	command (string)	—	Writes to the output file the assembly code that implements the given arithmetic-logical command.
`WritePushPop` (developed in project 7)	command (`C_PUSH` or `C_POP`), segment (string), `index` (int)	—	Writes to the output file the assembly code that implements the given `push` or `pop` command.

(API continues in the next slide)

CodeWriter

Routine	*Arguments*	*Returns*	*Function*
`writeLabel`	`label` (string)	—	Writes assembly code that effects the `label` command.
`writeGoto`	`label` (string)	—	Writes assembly code that effects the `goto` command.
`writeIf`	`label` (string)	—	Writes assembly code that effects the `if-goto` command.
`writeFunction`	`functionName` (string) `nVars` (int)	—	Writes assembly code that effects the `function` command.
`writeCall`	`functionName` (string) `nArgs` (int)	—	Writes assembly code that effects the `call` command.
`writeReturn`	—	—	Writes assembly code that effects the `return` command.
`close` (developed in project 7)	—	—	Closes the output file.

The generated assembly code must follow the guidelines and symbols described in the Standard Mapping of the VM on the Hack Platform.

Lecture plan

Abstraction

- VM branching commands
- VM function commands

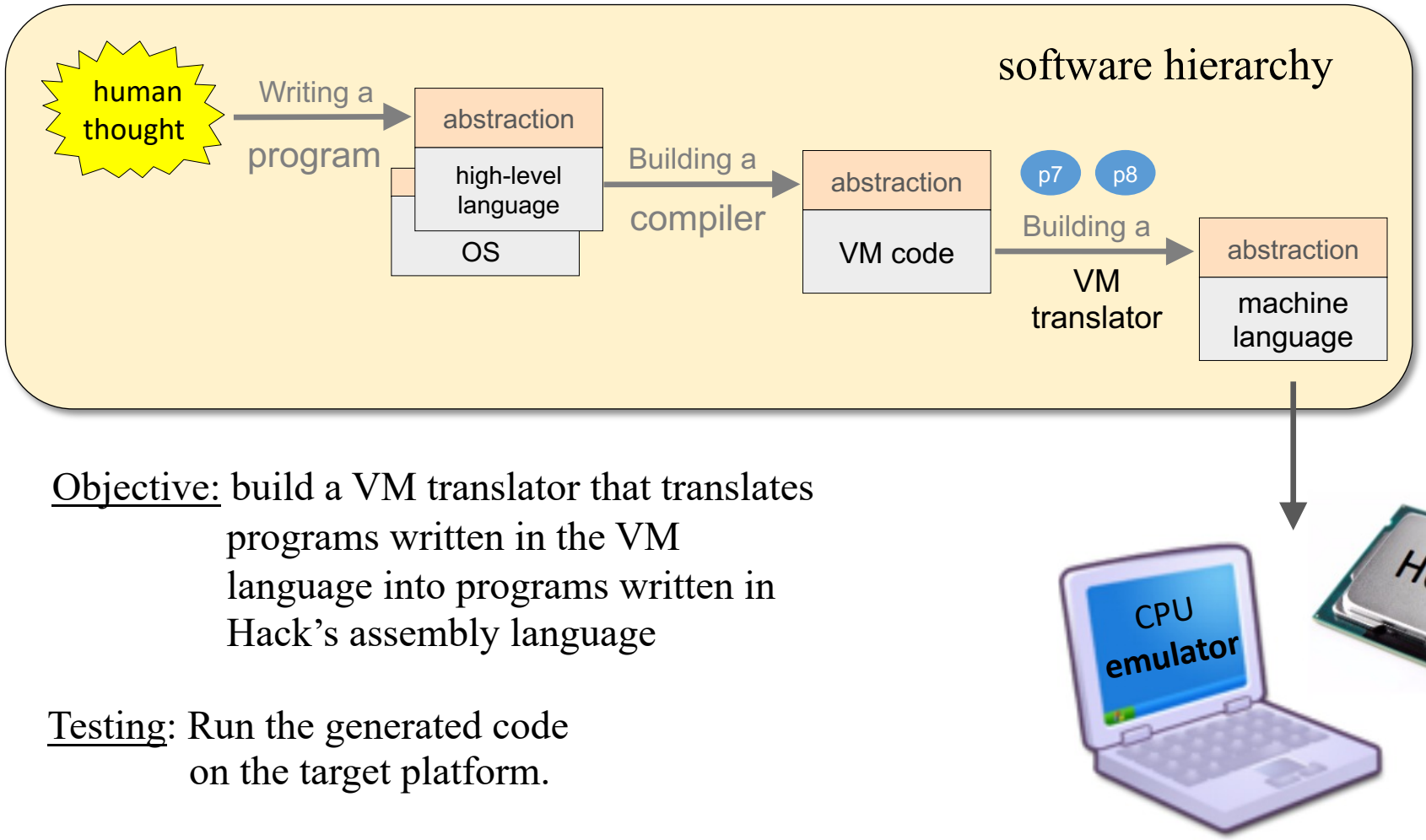
Implementation (conceptual)

- VM branching commands
- VM function commands

Implementation (nuts and bolts)

- Run-time system
- Standard mapping
- VM translator
- Project 8

The Big Picture



Objective: build a VM translator that translates programs written in the VM language into programs written in Hack's assembly language

Testing: Run the generated code on the target platform.

Test programs

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Test programs: BasicLoop

ProgramFlow:

- BasicLoop
  - **BasicLoop.vm**
  - BasicLoopVME.tst
  - BasicLoop.tst
  - BasicLoop.cmp
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

BasicLoop.vm

```
// Computes the sum 1 + 2 + ... + argument[0],
// and pushes the  result onto the stack.
push constant 0
   pop local 0
label LOOP_START
   push argument 0
   push local 0
   add
   pop local 0
   push argument 0
   push constant 1
   sub
   pop argument 0
   push argument 0
if-goto LOOP_START
   push local 0
```

Tests the handling of the VM commands:

`label`

`if-goto`

Test programs: FibonacciSeries

ProgramFlow:

- BasicLoop
- FibonacciSeries
  - **FibSeries.vm**
  - FibSeriesVME.tst
  - FibSeries.tst
  - FibSeries.cmp

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Test programs: FibonacciSeries

ProgramFlow:
- BasicLoop
- FibonacciSeries
  - **FibSeries.vm**
  - FibSeriesVME.tst
  - FibSeries.tst
  - FibSeries.cmp

FunctionCalls:
- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

FibSeries.vm

```
// Computes the first argument[0] elements of the Fibonacci series.
// Puts the elements in the RAM, starting at the address given in argument[1].
push argument 1
   pop pointer 1
   push constant 0
   pop that 0
   push constant 1
   pop that 1
   ...
label MAIN_LOOP_START
   push argument 0
   if-goto COMPUTE_ELEMENT
   goto END_PROGRAM
label COMPUTE_ELEMENT
   push that 0
   push that 1
   add
   ...
   goto MAIN_LOOP_START
label END_PROGRAM
```

A more elaborate test of handling the VM commands:

`Label`

`goto`

`if-goto`

Test programs: SimpleFunction

ProgramFlow:
- BasicLoop
- FibonacciSeries

FunctionCalls:
- SimpleFunction
  - **SimpleFunction.vm**
  - SimpleFunctionVME.tst
  - SimpleFunction.tst
  - SimpleFunction.cmp
- NestedCall
- FibonacciElement
- StaticsTest

SimpleFunction.vm

```
// Performs a simple (and meaningless) calculation involving local
// and argument values, and returns the result.

function SimpleFunction.test 2
   push local 0
   push local 1
   add
   not
   push argument 0
   add
   push argument 1
   sub
   return
```

Tests the handling of the VM commands
- `function`
- `return`

Basic test, involving no caller

Test programs: FibonacciElement

ProgramFlow:
- BasicLoop
- FibonacciSeries

FunctionCalls:
- SimpleFunction
- NestedCall
- FibonacciElement
  - **Main.vm**
  - **Sys.vm**
  - FibElementVME.tst
  - FibElement.tst
  - FibElement.cmp
- StaticsTest

Usage: % VMTranslator FibonacciElement (translates a *folder*)

Generates a single output file: FibonacciElement.asm

- Tests that the VM translator can handle more than one VM file
- Tests the handling of function, return, call
- Tests that the VM translator initializes the memory segments
- Tests that the the bootstrap code initializes the stack and calls Sys.init

Test programs: FibonacciElement

ProgramFlow:
- BasicLoop
- FibonacciSeries

FunctionCalls:
- SimpleFunction
- NestedCall
- FibonacciElement
  - **Main.vm**
  - **Sys.vm**
  - FibElementVME.tst
  - FibElement.tst
  - FibElement.cmp
- StaticsTest

Main.vm

```
// Main.fibonacci: computes the n'th element of the Fibonacci series,
// recursively. The n value is supplied by the caller, and stored in
// argument 0.
function Main.fibonacci 0
   push argument 0
   push constant 2
   lt
   if-goto IF_TRUE
   goto IF_FALSE
label IF_TRUE
   push argument 0
   return
label IF_FALSE
   push argument 0
   push constant 2
   sub
   call Main.fibonacci 1
   push argument 0
   push constant 1
   sub
   call Main.fibonacci 1
   add
   return
```

Test programs: FibonacciElement

- ProgramFlow:
  - BasicLoop
  - FibonacciSeries
- FunctionCalls:
  - SimpleFunction
  - NestedCall
  - FibonacciElement
    - **Main.vm**
    - **Sys.vm**
    - FibElementVME.tst
    - FibElement.tst
    - FibElement.cmp
  - StaticsTest

Main.vm

```
// Main.fibonacci: computes the n'th element of the Fibonacci series,
// recursively. The n value is supplied by the caller, and stored in
// argument 0.
function Main.fibonacci 0
   push argument 0
   push constant 2
   lt
   if-goto IF_TRUE
   goto IF_FALSE
label IF_TRUE
   push argument 0
   return
label IF_FALSE
   push argument 0
   push constant 2
   sub
   call Main.fibonacci 1
   push argument 0
   push constant 1
   sub
   call Main.fibonacci 1
   add
   return
```

Normally, `Sys.init` is used to call `Main.main`.

In Project 8 we use `Sys.init` to call test functions, directly.

Sys.vm

```
// Sys.init: pushes n onto the stack,
// and calls Main.fibonacii to compute
// the n'th Fibonacci element.

// (Called by the bootstrap code generated
// by the VM translator ).
function Sys.init 0
   push constant 4
   call Main.fibonacci 1
label WHILE
   goto WHILE
```

Test programs: NestedCall

ProgramFlow:
- BasicLoop
- FibonacciSeries

FunctionCalls:
- SimpleFunction
- NestedCall
  - **Sys.vm**
  - NestedCallVME.tst
  - NestedCall.tst
  - NestedCall.cmp
  - NestedCall.html
  - NestedCallStack.html
- FibonacciElement
- StaticsTest

- Closes testing gaps between SimpleFunction and FibonacciElement
- Recommended when SimpleFunction tests successfully and FibonacciElement fails or crashes
- Self-documented

Test programs: StaticsTest

ProgramFlow:
- BasicLoop
- FibonacciSeries

FunctionCalls:
- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Tests the handling of static variables in a program consisting of more than one VM file

Class1.vm
Class2.vm
Sys.vm

StaticsTestVME.tst
StaticsTest.tst
StaticsTest.cmp

Usage: `% VMTranslator StaticTest` (translates a *folder*)

Generates a single output file: `StaticTest.asm`

Test programs: StaticsTest

ProgramFlow:
- BasicLoop
- FibonacciSeries

FunctionCalls:
- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest
  - **Class1.vm**
  - **Class2.vm**
  - **Sys.vm**
  - StaticsTestVME.tst
  - StaticsTest.cmp

Class1.vm

```
// Stores two supplied arguments in
// static 0 and static 1
function Class1.set 0
    push a
    pop st
    push a
    pop st
    push c
    return

// Returns (s
function
    push s
    push s
    sub
    return
```

Class2.vm

```
// Stores two supplied arguments in
 static 0 and static 1
function Class2.set 0
    push
    pop s
    push
    pop s
    push
    retur

// Returns
functior
    push
    push
    sub
    retur
```

Sys.vm

```
function Sys.init 0
   // Calls Class1.set with 6 and 8
    push constant 6
    push constant 8
    call Class1.set 2
    pop temp 0      // dumps the return value
   // Calls Class2.set with 23 and 15
    push constant 23
    push constant 15
    call Class2.set 2
    pop temp 0      // dumps the return value
    // Checks the two resulting static segments
    call Class1.get 0
    call Class2.get 0
label WHILE
    goto WHILE
```

Test programs

ProgramFlow:
- BasicLoop
  - **BasicLoop.vm**
  - BasicLoopVME.tst
  - BasicLoop.tst
  - BasicLoop.cmp
- FibonacciSeries

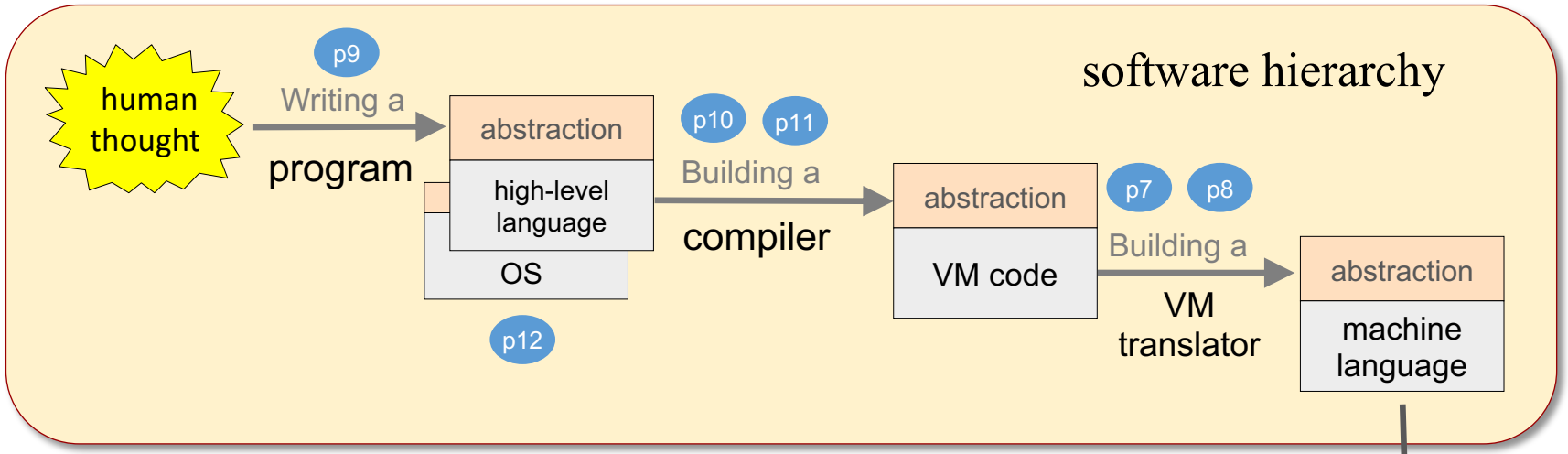
FunctionCalls:
- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Testing routine for every test program `Xxx`:

0. Recommended: Load and run `XxxVME.tst` on the VM emulator; This script loads the `Xxx.vm` test program into the VM emulator, allowing you to experiment with its code
1. Use your VM translator to translate `Xxx.vm` generating a file named `Xxx.asm`
2. Load and run `Xxx.tst` on the CPU emulator; This script loads `Xxx.asm` into the emulator, executes it, and compares the output to `Xxx.cmp`

Note: All these files are supplied, except for `Xxx.asm`

Recap / Next



Work plan

- ✓ Projects 7,8: VM translator (compiler's backend)
- Projects 10,11: Compiler (compiler's frontend)
- ➡ Project 9: High-level language
- Project 12: Operating system.