Lecture 8

Virtual Machine, Part II

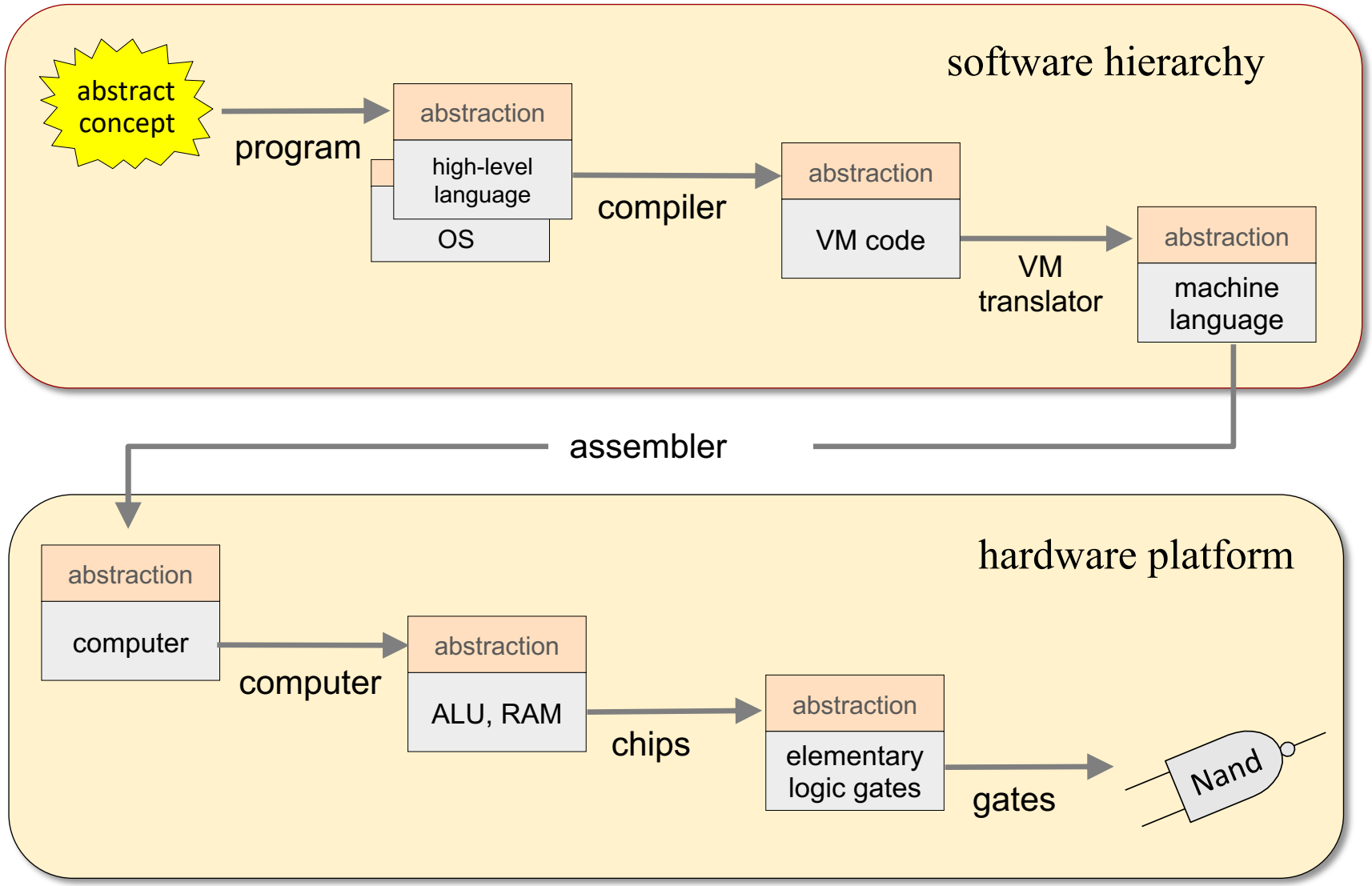
These slides support chapter 8 of the book

The Elements of Computing Systems

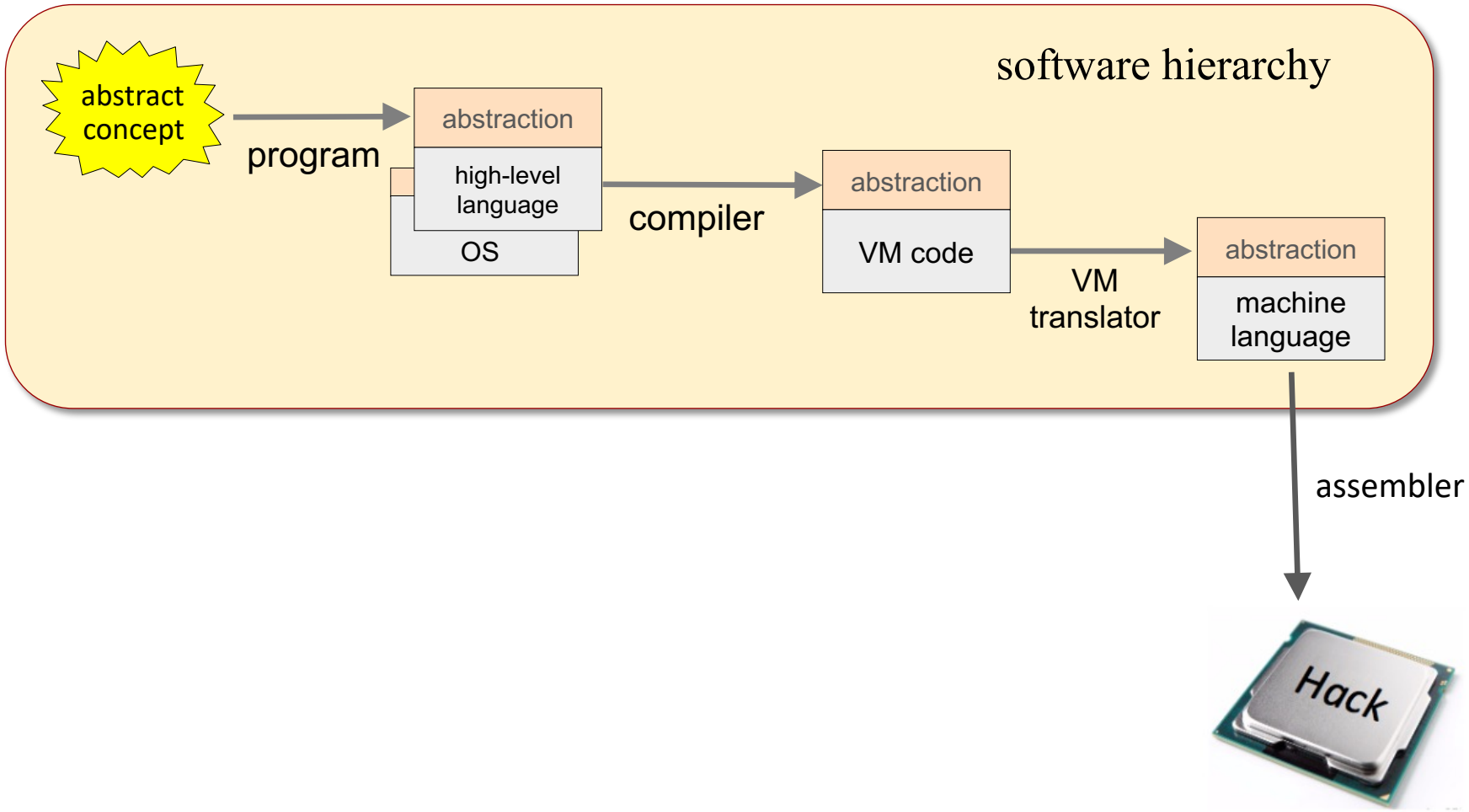
By Noam Nisan and Shimon Schocken

MIT Press, 2021

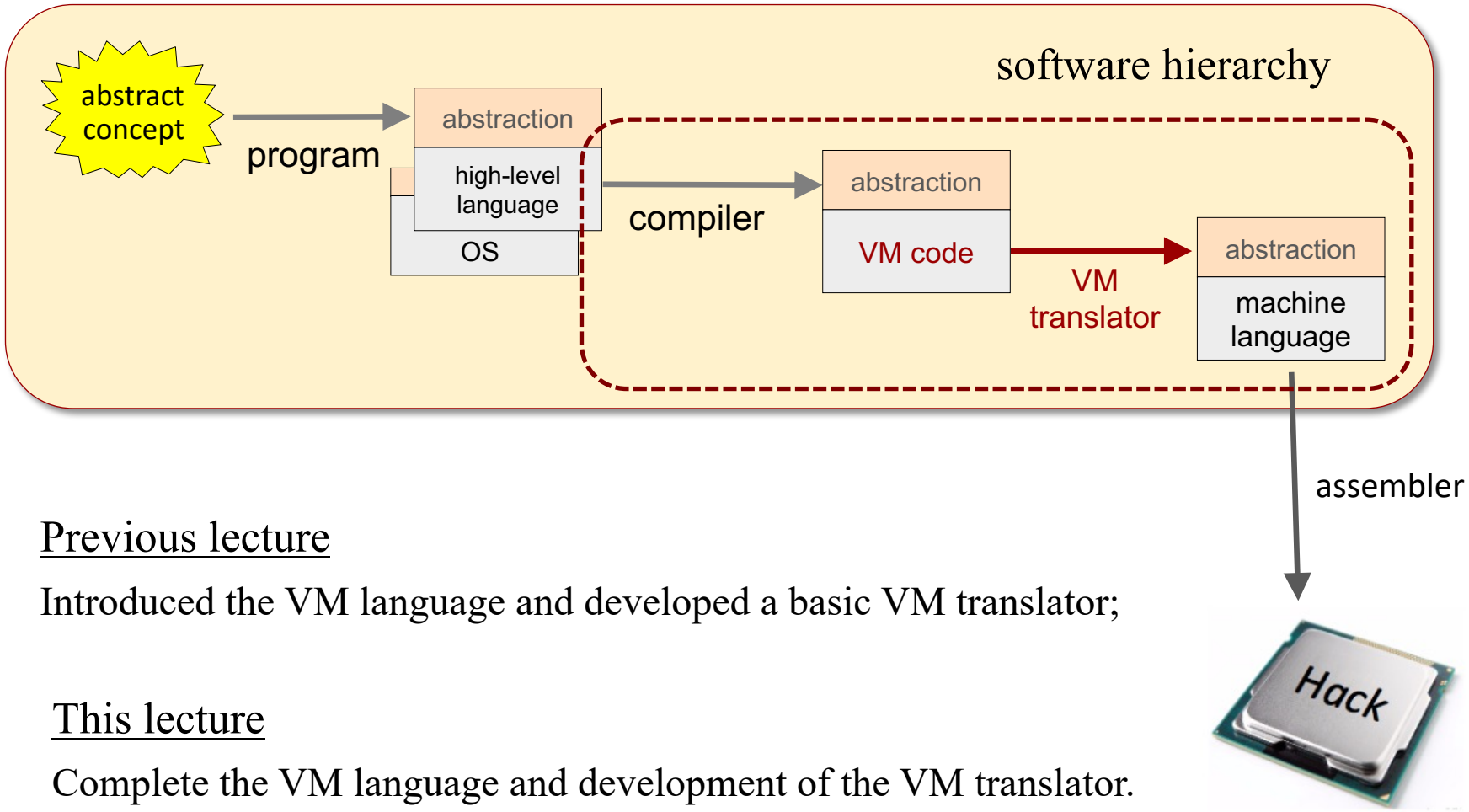
Nand to Tetris Roadmap



Nand to Tetris Roadmap: Part II



Nand to Tetris Roadmap: Part II



Previous lecture

Introduced the VM language and developed a basic VM translator;

This lecture

Complete the VM language and development of 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
   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
```

Compilation focus (lectures 10 – 11)

The compiler maps symbolic variables (`sum`, `n`, `x`,...) on virtual memory segment entries (`local 0`, `local 1`, `argument 0`, ...), and generates VM code that operates on these segments.

The big picture: Compilation

The VM Language

✓ **Arithmetic / Logical commands**

`add`, `sub` , `neg`

`eq` , `gt` , `lt`, `and`, `or` , `not`

✓ **Push / pop commands**

`push`

`pop`

Branching commands

`label`

`goto`

`if-goto`

Function commands

`function`

`call`

`return`

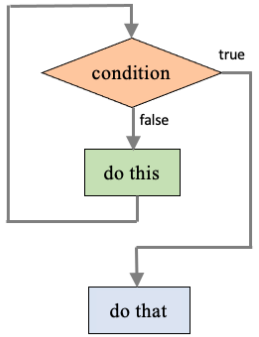
VM code

```
// Returns x * y
function mult
   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
```

Virtual machine focus (lectures 7–8)

Understanding and implementing the target VM code

Branching (in a nutshell)



VM code

```
// Returns x * y
function mult
    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
```

Programs typically include branching logic:

- Unconditional goto
- Conditional goto

Implementation challenges

How to realize:

Conditions

Goto destinations

Goto operations.

Functions (in a nutshell)

Function `bar`

```
// Some function
function bar
   ...
   // Computes 8 * 5 + 7
   push constant 8
   push constant 5
   call mult 2
   push constant 7
   add
   ...
   return
```

caller

Function `mult`

```
// Returns x * y
function mult
   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
```

callee

Programs typically consist of one ore more:

- "methods"
- "functions"
- "subroutines"
- "procedures"

etc.

Different high-level languages have different terminologies;

In the VM language terminology, all these code units are called *functions*

The functions call each other, as shown in the example.

Functions (in a nutshell)

Function `bar`

```
// Some function
function bar
   ...
   // Computes 8 * 5 + 7
   push constant 8
   push constant 5
   call mult 2
   push constant 7
   add
   ...
   return
```

caller

Function `mult`

```
// Returns x * y
function mult
   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
```

callee

Implementation issues

How to pass arguments from the caller to the callee?

How to switch from executing the caller to executing the callee?

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

How to terminate the callee, and switch back to the caller?

Take home lessons

Understanding program execution

- Branching
- Function call-and-return
- Recursion

Behind the scene

- Compilation
- Virtual machine
- Run-time
- Stack processing.

Branching: Abstraction

```
// Returns x * y
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

label

goto

if-goto

Branching: Abstraction

```
// Returns x * y
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

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

Syntax: `label` *label*

Semantics

Marks the destination of `goto` commands.

Branching: Abstraction

```
// Returns x * y
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

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

Syntax: `goto` *label*

Semantics

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

Branching: Abstraction

```
// Returns x * y
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

- `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.

Branching: Abstraction

```
// Returns x * y
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

- `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.

Convention: The code writer (typically, a *compiler*) must write code that pushes a boolean expression onto the stack before the `if-goto` command;

In this example, the highlighted code implements the abstraction: `if (local 1 > argument 1) goto END_LOOP`

Branching: Implementation

Abstraction (recap)

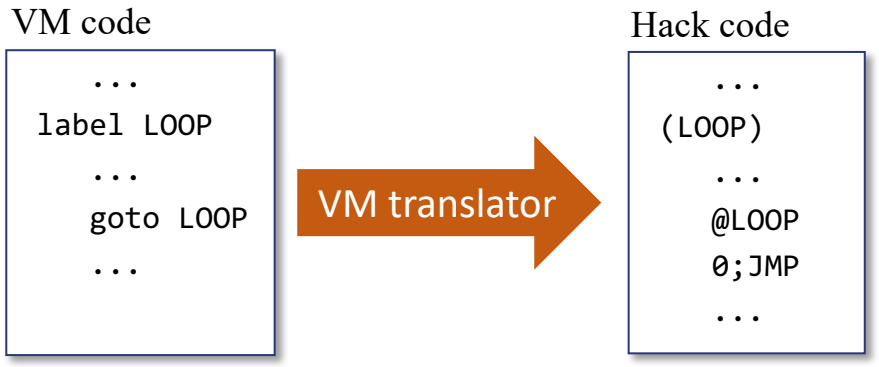
`label` *label* // label declaration

`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 generate machine language instructions that realize the command on the target platform. Example:



The instruction set of every computer features low-level "labeling" and "goto" primitives;

Therefore, the translation of the VM branching commands to the machine language of the target platform is not difficult.

Lecture plan

✓ Branching

- Abstraction
- Implementation

Function call and return

- Abstraction
- Implementation

VM translator

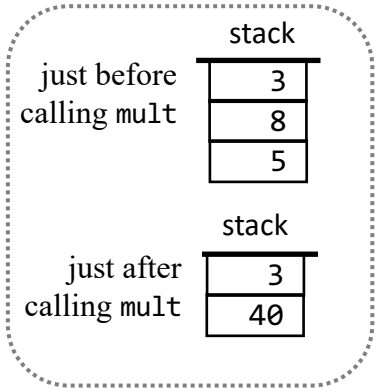
- Bootstrap
- Conventions
- Architecture
- Project 8

Functions: Abstraction

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

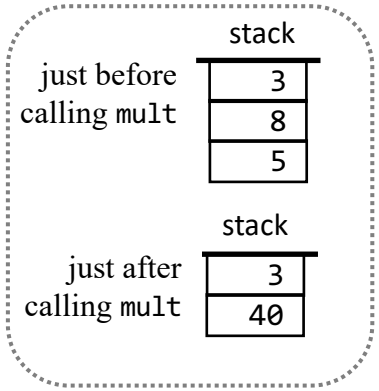
A function (the *caller*) calls a function (the *callee*) for its effect

Functions: Abstraction

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

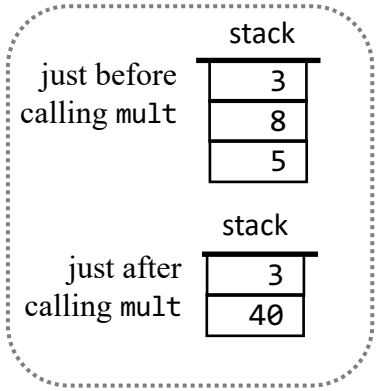
```
call
function
return
```

Functions: Abstraction

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

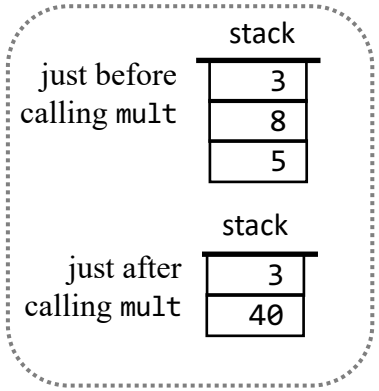
Convention: The caller must push *nArgs* arguments onto the stack before the `call` command.

Functions: Abstraction

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

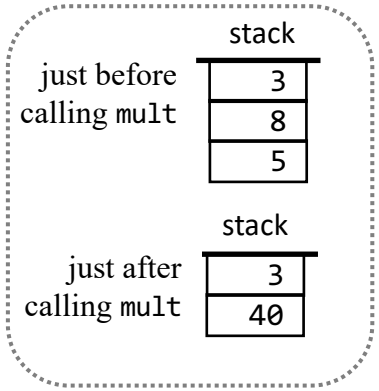
Note: In this example the caller passes 2 arguments, and the function has 2 local variables. This is just a coincidence; *nArgs* had nothing to do with *nVars*.

Functions: Abstraction

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 push a value onto the stack before 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`.

Lecture plan

Branching

- Abstraction
- Implementation

Function call and return

✓ Abstraction

➡ Implementation

VM translator

- Bootstrap
- Conventions
- Architecture
- Project 8

Function call and return

caller

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

Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect

VM files and VM functions

The full name of a VM function is *fileName*.*functionName*

In this example, the caller and the callee happen to be in the same VM file, `Foo.vm`

In general, they can be in different VM files.

Function call and return

caller

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

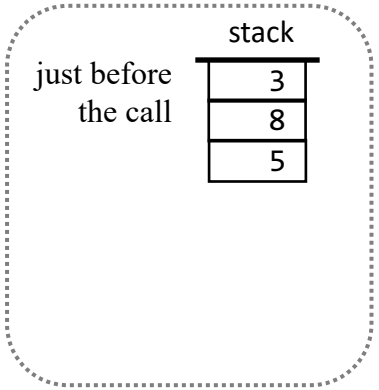
callee

```
// 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
```

Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect

`Foo.bar`'s view



Function call and return

caller

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

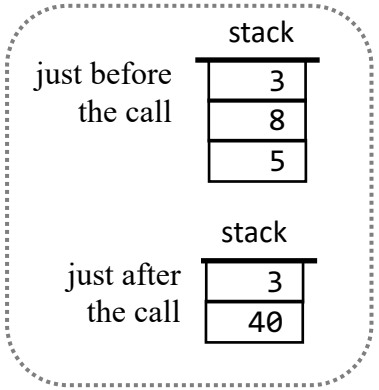
callee

```
// 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
```

Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect

`Foo.bar`'s view



Magic!

Let's open the black box

Function call and return

caller

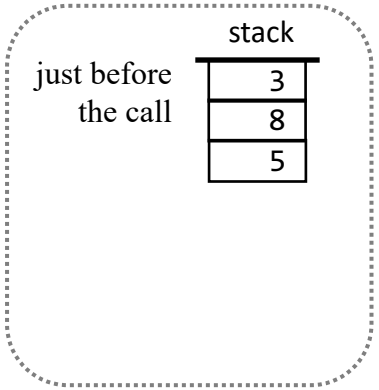
```
function Foo.bar 4
   ...
   // Computes 3 + 8 * 5
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
   return
```

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
```

Adding line numbers, just for reference

Foo.bar's view



Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect

Function call and return

caller

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

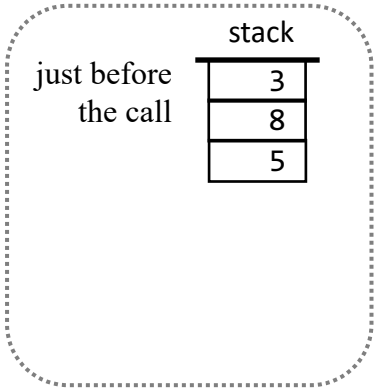
callee

```
   // 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
```

Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect

`Foo.bar`'s view



The caller's execution is put on hold

Function call and return

caller

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

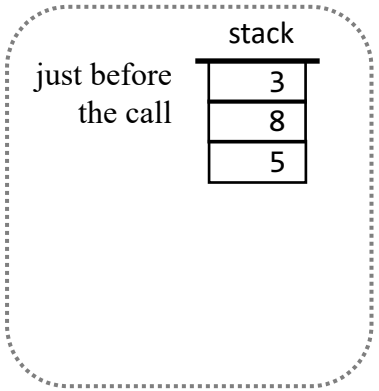
callee

```
   // 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
```

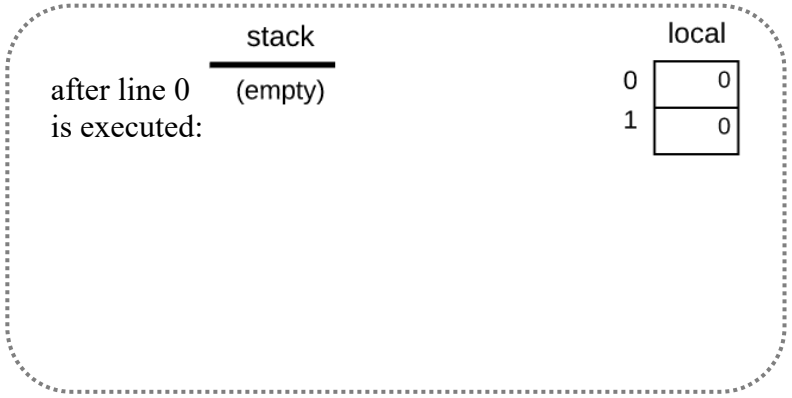
Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect

`Foo.bar`'s view



`Foo.mult`'s view



The caller's execution is put on hold

Function call and return

caller

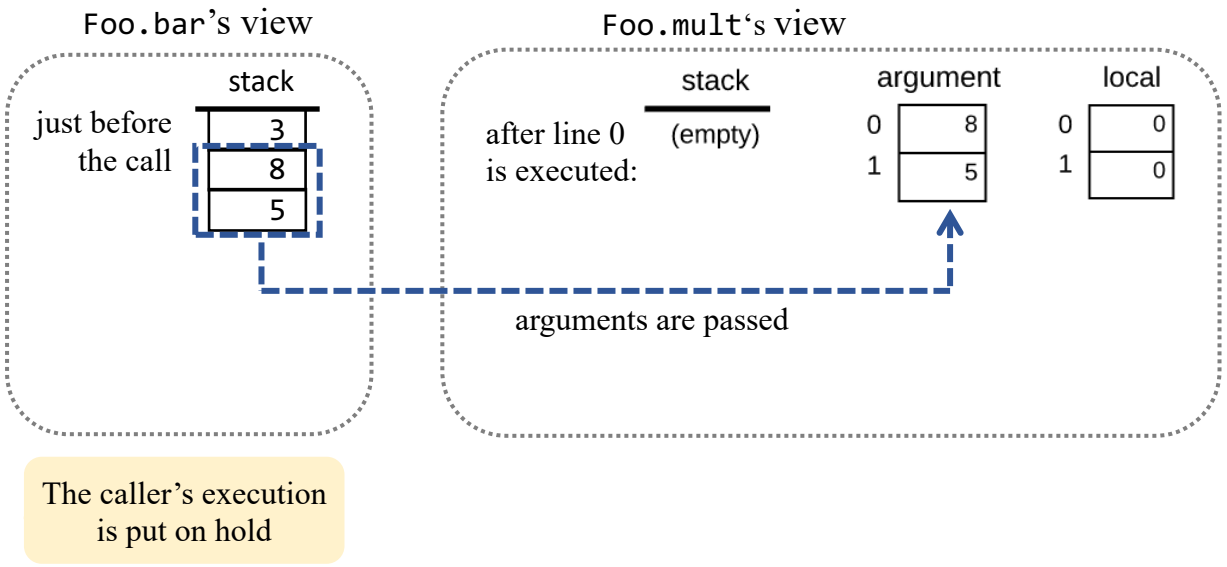
```
function Foo.bar 4
   ...
   // Computes 3 + 8 * 5
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
   return
```

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
```

Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect



The caller's execution is put on hold

Function call and return

caller

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

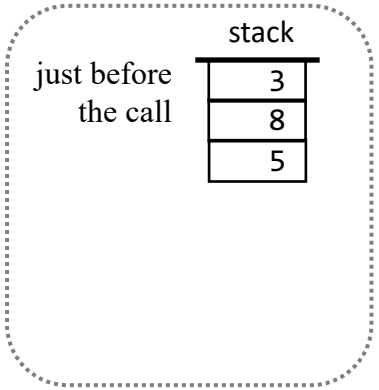
callee

```
   // 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
```

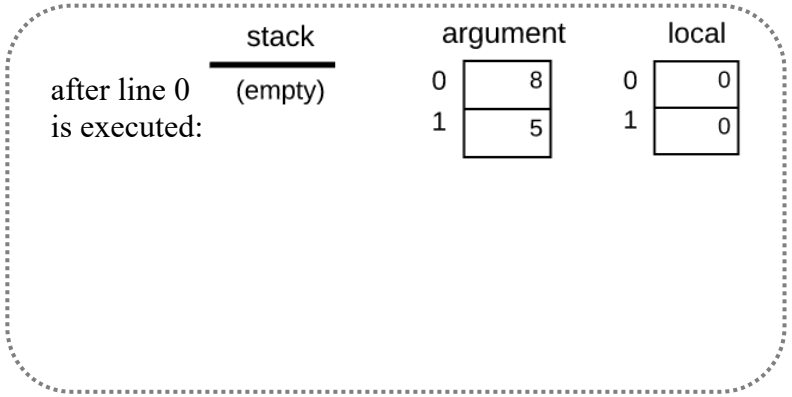
Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect

`Foo.bar`'s view



`Foo.mult`'s view



The callee's code is executed

Function call and return

caller

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

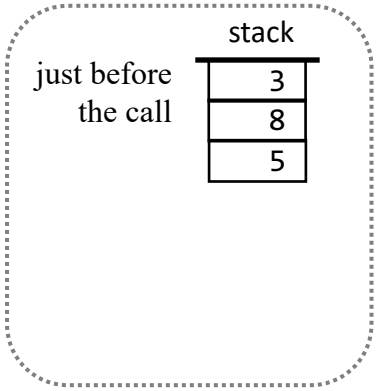
callee

```
   // 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
```

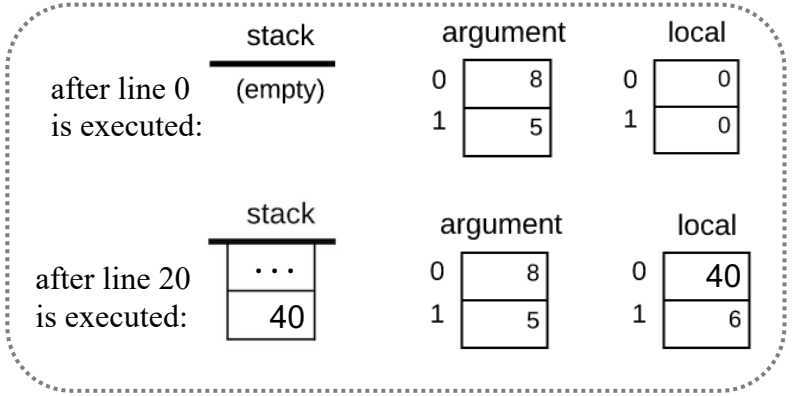
Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect

`Foo.bar`'s view



`Foo.mult`'s view



The callee's code is executed

Function call and return

caller

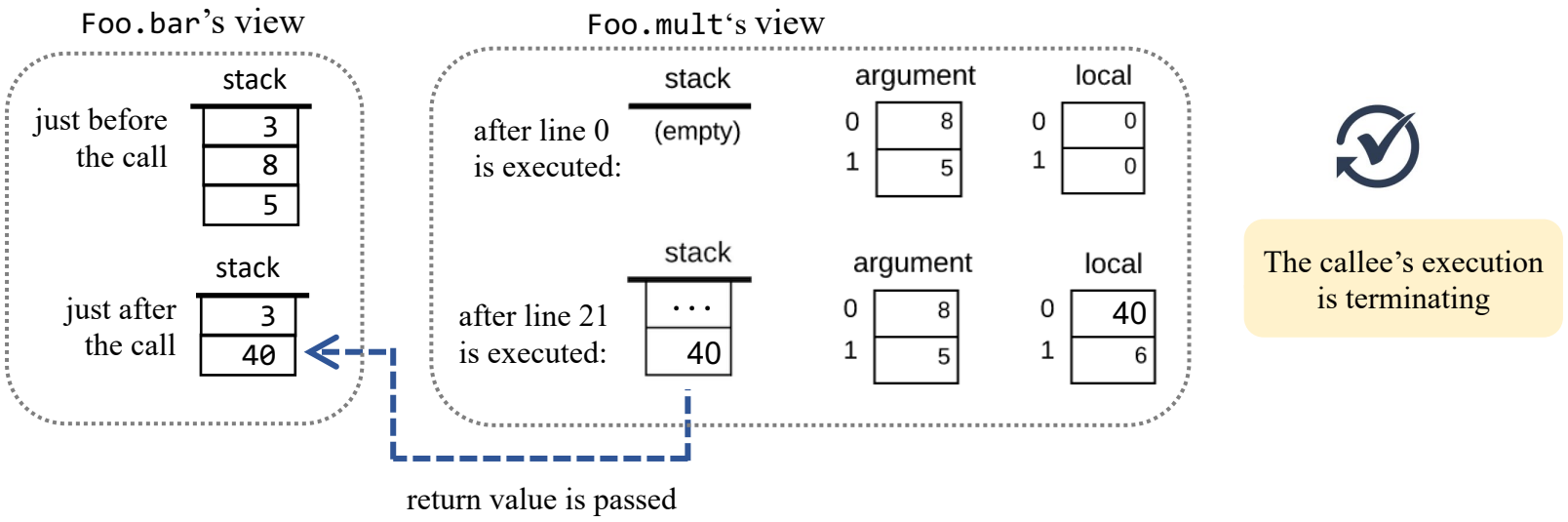
```
function Foo.bar 4
   ...
   // Computes 3 + 8 * 5
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
   return
```

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
```

Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect



The callee's execution is terminating

Function call and return

caller

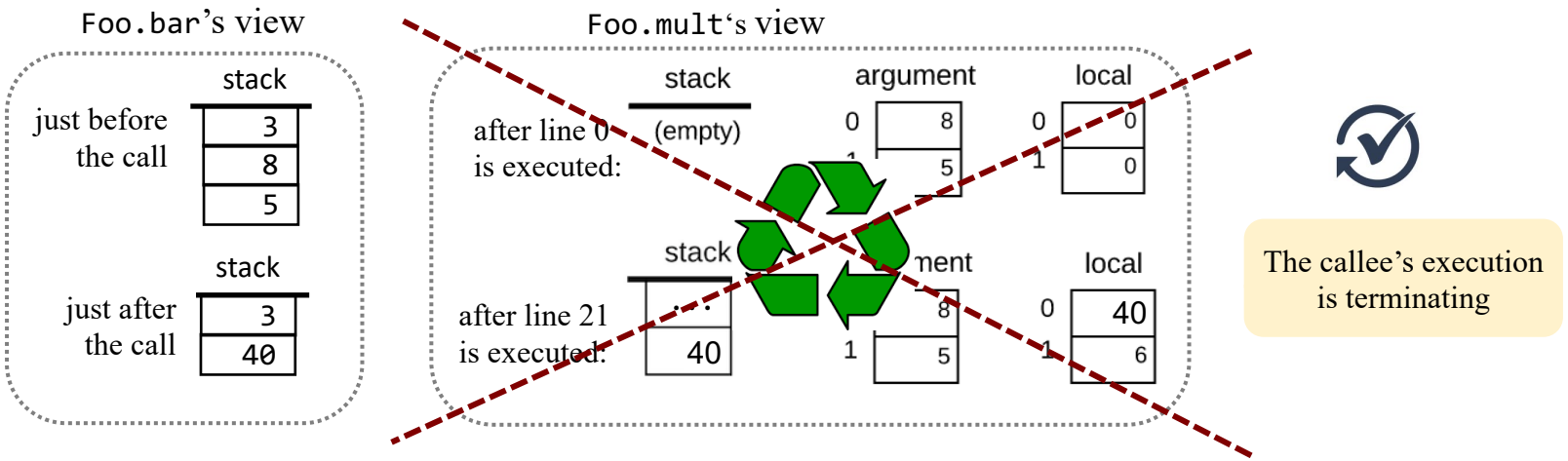
```
function Foo.bar 4
   ...
   // Computes 3 + 8 * 5
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   ...
   return
```

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
```

Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect



Function call and return

caller

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

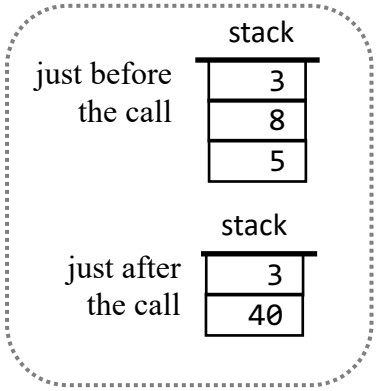
callee

```
   // 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
```

Typical function-call-and-return scenario

Function `Foo.bar` (the *caller*) calls function `Foo.mult` (the *callee*) for its effect

`Foo.bar`'s view



The caller's execution is resumed;

The next command to be executed:

`add`

Function call and return

Abstraction (recap):

A VM program consists of one or more VM functions;

The functions call each other, for their effect (including recursively);

Each function execution sees its own working stack and memory segments;

Arguments and return values are passed, somehow.

Implementation

We'll describe the handling of the function-call-and-return commands in two stages:

- The translation process
- The global stack

Translation

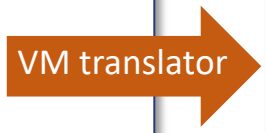
VM code

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:

```
// Computes 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
```



Generated code (pseudo assembly)

Conventions

Each VM function starts with a `function` command, and ends with a `return` command;

Each VM function returns a value (void functions return a value which is ignored by the caller). The return value is the top value in the callee's stack.

Translation

VM code

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:

```
// Computes 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 (pseudo assembly)

```
// function Foo.bar 4
(Foo.bar)  // injected function's entry point label
    // assembly code that initializes the function's 4 local variables
```

Translation

VM code

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:

```
// Computes 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 (pseudo assembly)

```
// function Foo.bar 4
(Foo.bar)  // injected function's entry point label
    // assembly code that initializes the function's 4 local variables
    // assembly code that handles  ..., push constant 3, ..., push constant 5
```

Blue: Generated by the basic VM translator (project 7);

Black: Generated by the completed VM translator (project 8).

Translation

VM code

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:

```
// Computes 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 (pseudo assembly)

```
// function Foo.bar 4
(Foo.bar)  // injected function's entry point label
    // assembly code that initializes the function's 4 local variables
    // assembly code that handles ..., push constant 3, ..., push constant 5
// call Foo.mult 2
    // Creates a return address label (Foo.bar$ret.1); Saves the return
    // address and the caller's segment pointers, and then generates:
    goto Foo.mult      // injected branching to the called function
(Foo.bar$ret.1)        // injected return address label
```

Blue: Generated by the basic VM translator (project 7);

Black: Generated by the completed VM translator (project 8).

Translation

VM code

caller:

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

VM translator

callee:

```
// Computes 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
```

Generated code (pseudo assembly)

```
// function Foo.bar 4
(Foo.bar)   // injected function's entry point label
    // assembly code that initializes the function's 4 local variables
    // assembly code that handles ..., push constant 3, ..., push constant 5
// call Foo.mult 2
    // Creates a return address label (Foo.bar$ret.1); Saves the return
    // address and the caller's segment pointers, and then generates:
    goto Foo.mult      // injected branching to the called function
(Foo.bar$ret.1)        // injected return address label
    // assembly code that handles add, ...
```

Blue: Generated by the basic VM translator (project 7);

Black: Generated by the completed VM translator (project 8).

Translation

VM code

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:

```
// Computes 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 (pseudo assembly)

```
// function Foo.bar 4
(Foo.bar)  // injected function's entry point label
    // assembly code that initializes the function's 4 local variables
    // assembly code that handles ..., push constant 3, ..., push constant 5
// call Foo.mult 2
    // Creates a return address label (Foo.bar$ret.1); Saves the return
    // address and the caller's segment pointers, and then generates:
    goto Foo.mult      // injected branching to the called function
(Foo.bar$ret.1)        // injected return address label
    // assembly code that handles add, ...

// function Foo.mult 2
(Foo.mult)  // injected function's entry point label
    // assembly code that initializes the function's 2 local variables
```

Translation

VM code

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:

```
// Computes 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 (pseudo assembly)

```
// function Foo.bar 4
(Foo.bar)  // injected function's entry point label
    // assembly code that initializes the function's 4 local variables
    // assembly code that handles  ..., push constant 3, ..., push constant 5
// call Foo.mult 2
    // Creates a return address label (Foo.bar$ret.1); Saves the return
    // address and the caller's segment pointers, and then generates:
    goto Foo.mult      // injected branching to the called function
(Foo.bar$ret.1)        // injected return address label
    // assembly code that handles  add, ...

// function Foo.mult 2
(Foo.mult)  // injected function's entry point label
    // assembly code that initializes the function's 2 local variables
    // assembly code that handles push constant 0 , ..., push local 0
```

Translation

VM code

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:

```
// Computes 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 (pseudo assembly)

```
// function Foo.bar 4
(Foo.bar)   // injected function's entry point label
    // assembly code that initializes the function's 4 local variables
    // assembly code that handles  ..., push constant 3, ..., push constant 5
// call Foo.mult 2
    // Creates a return address label (Foo.bar$ret.1); Saves the return
    // address and the caller's segment pointers, and then generates:
    goto Foo.mult      // injected branching to the called function
(Foo.bar$ret.1)        // injected return address label
    // assembly code that handles  add, ...

// function Foo.mult 2
(Foo.mult)    // injected function's entry point label
    // assembly code that initializes the function's 2 local variables
    // assembly code that handles push constant 0 , ..., push local 0
// return
    // Retrieves the saved return address (which, in this example, happens to be
    // Foo.bar$ret.1), replaces the arguments pushed by the caller with the
    //  return value (stack's top element), reinstates the segment pointers of the
    //  caller, and then generates:
    goto Foo.bar$ret.1  // injected branching back to the calling site.
```

Translation

VM code

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:

```
// Computes 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 (pseudo assembly)

```
// function Foo.bar 4
(Foo.bar)   // injected function's entry point label
    // assembly code that initializes the function's 4 local variables
    // assembly code that handles  ..., push constant 3, ..., push constant 5
// call Foo.mult 2
    // Creates a return address label (Foo.bar$ret.1); Saves the return
    // address and the caller's segment pointers, and then generates:
    goto Foo.mult      // injected branching to the called function
(Foo.bar$ret.1)        // injected return address label
    // assembly code that handles  add, ...

// function Foo.mult 2
(Foo.mult)    // injected function's entry point label
    // assembly code that initializes the function's 2 local variables
    // assembly code that handles push constant 0 , ..., push local 0
// return
    // Retrieves the saved return address (which, in this example, happens to be
    // Foo.bar$ret.1), replaces the arguments pushed by the caller with the
    //  return value (stack's top element), reinstates the segment pointers of the
    //  caller, and then generates:
    goto Foo.bar$ret.1  // injected branching back to the calling site.
```

Notes

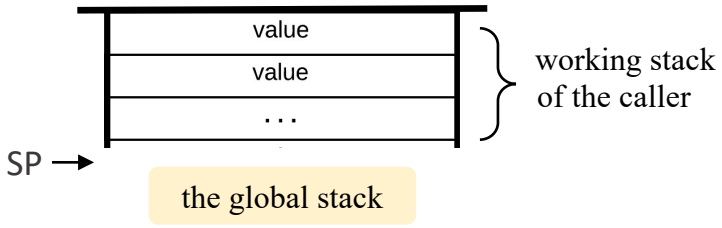
This pseudocode must be generated in the target platform's assembly language;

When the resulting assembly code will execute, it will realize the semantics implied by the VM code.

Translation (detailed): `call / function / return`

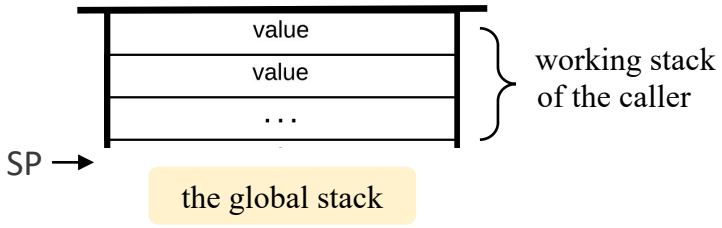
Translation (detailed): `call` / `function` / `return`

The caller is running,
doing various things



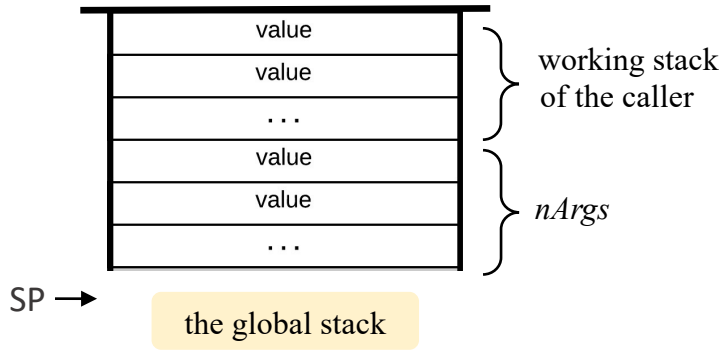
Translation (detailed): `call` / `function` / `return`

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



Translation (detailed): `call` / `function` / `return`

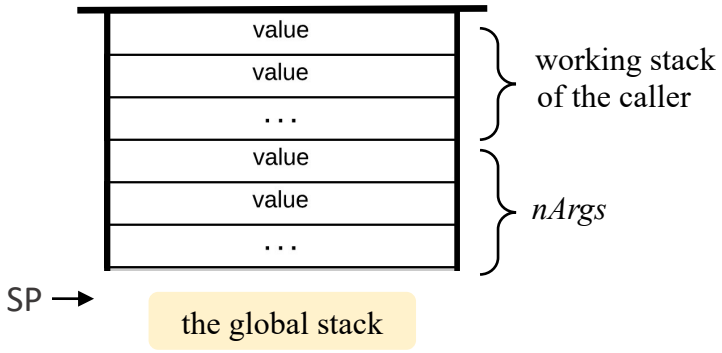
The caller prepares to call another function;
It pushes 0 or more arguments onto the stack



Translation (detailed): `call` / `function` / `return`

The caller says:

`call` *functionName nArgs*



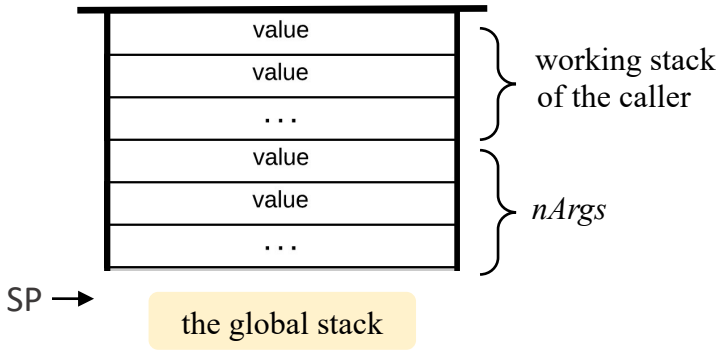
Handling `call` *functionName nArgs*

We have to:

Translation (detailed): `call` / `function` / `return`

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

We have to:

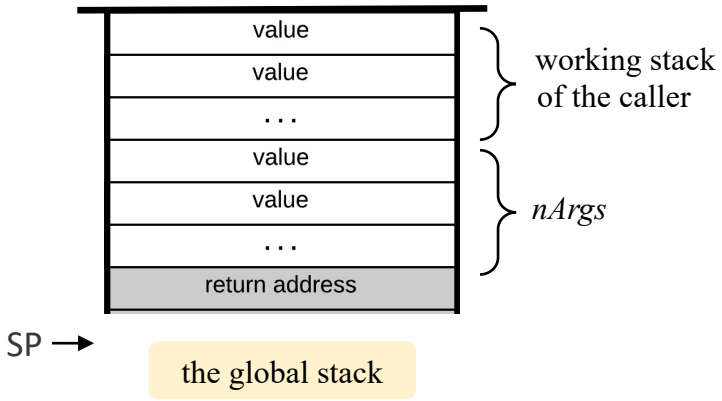
- Save the return address

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

Translation (detailed): `call` / `function` / `return`

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

We have to:

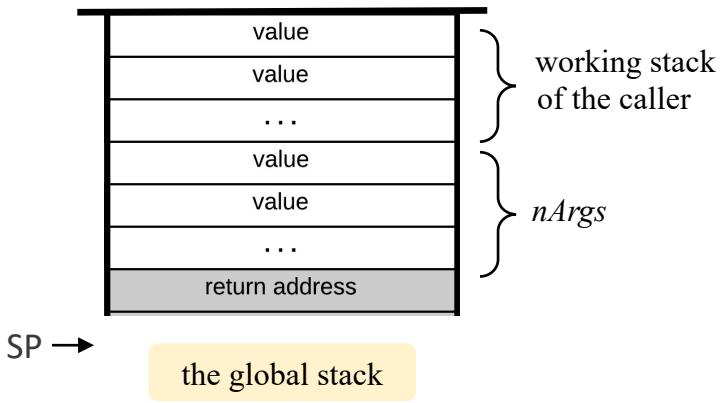
- Save the return address

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

Translation (detailed): `call` / `function` / `return`

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

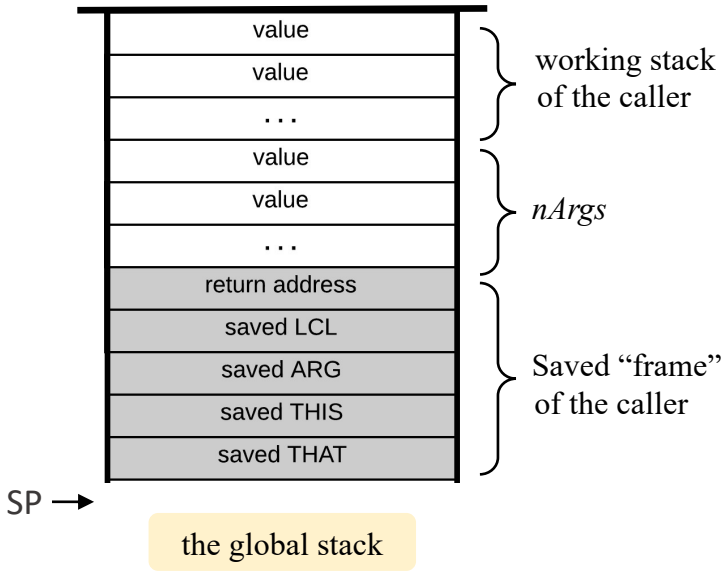
We have to:

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

Translation (detailed): `call` / `function` / `return`

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

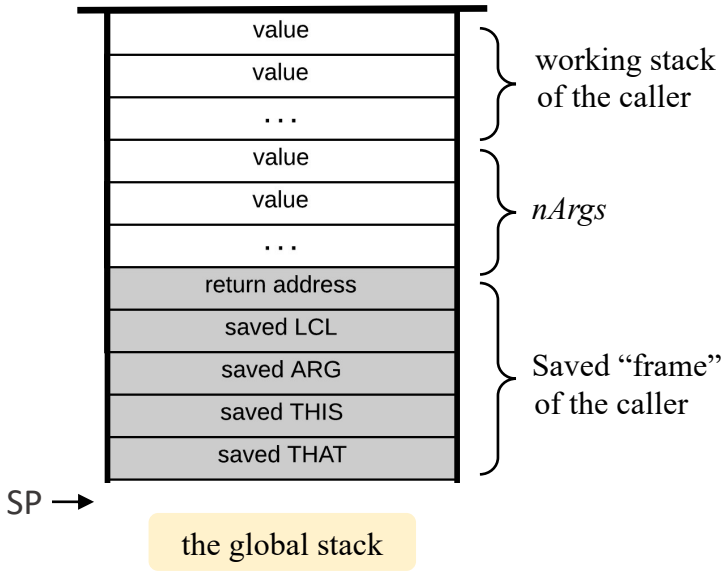
We have to:

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

Translation (detailed): `call` / `function` / `return`

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

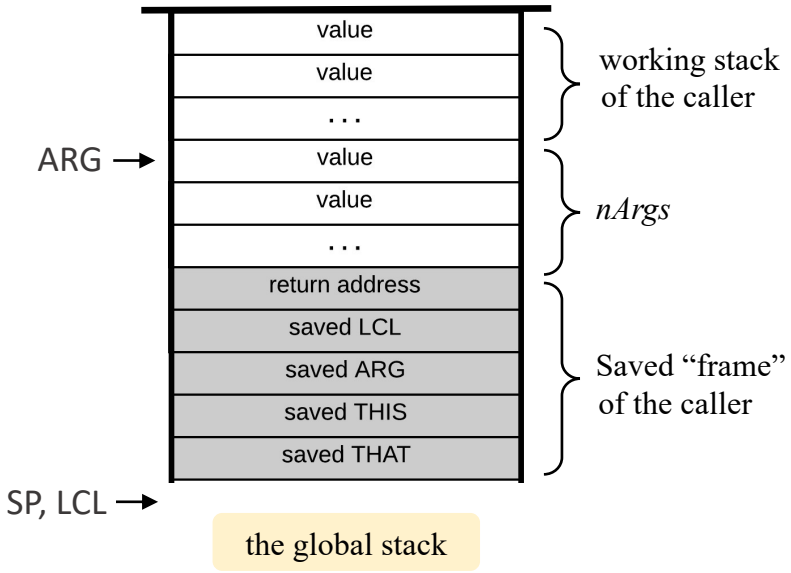
We have to:

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

Translation (detailed): `call` / `function` / `return`

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

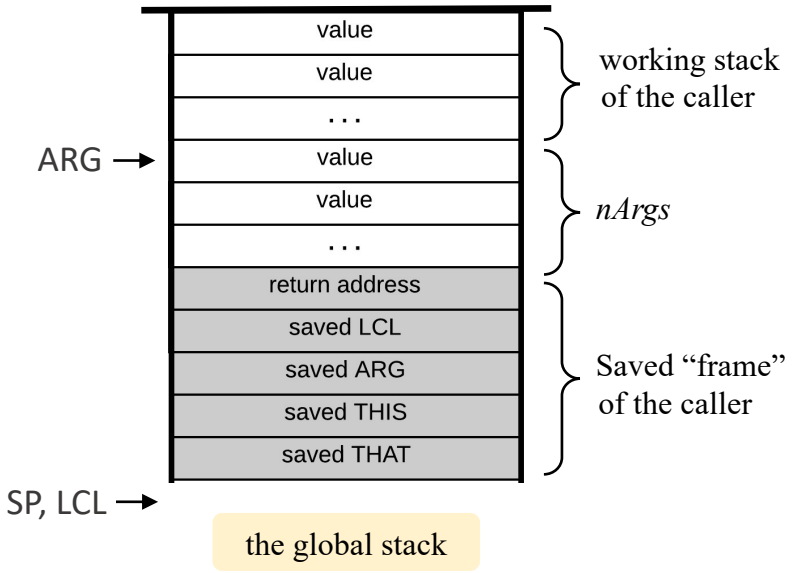
We have to:

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

Translation (detailed): `call` / `function` / `return`

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

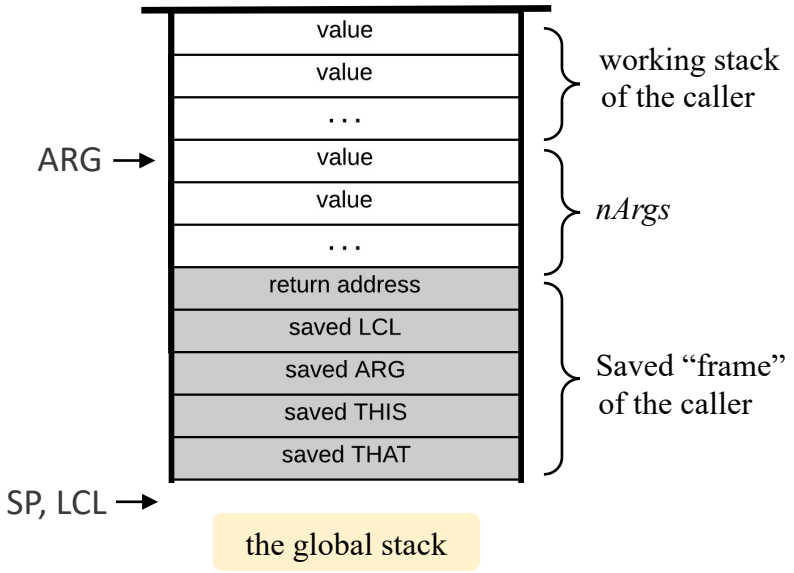
We have to:

- 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

Translation (detailed): `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

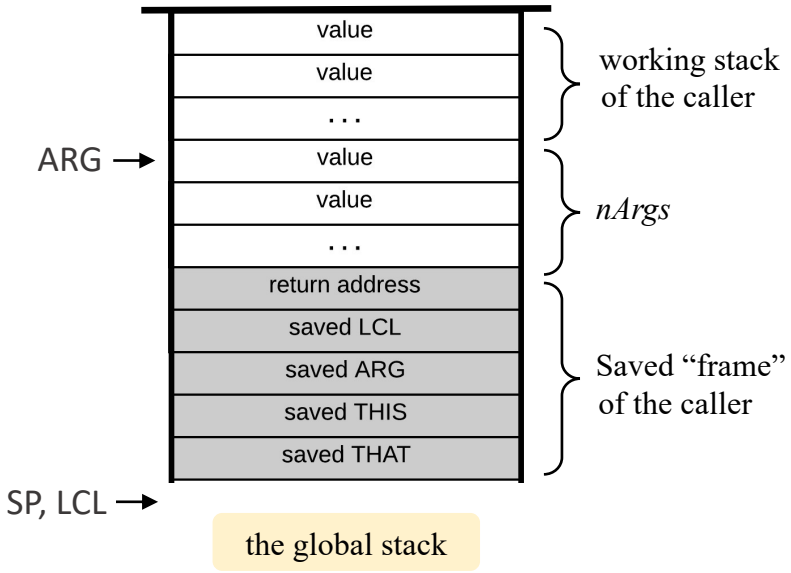
```
// call functionName nArgs
   push retAddrLabel     // Generates and pushes this label
   push LCL              // Saves the caller's LCL
   push ARG              // Saves the caller's ARG
   push THIS             // Saves the caller's THIS
   push THAT             // Saves the caller's THAT
   ARG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

Translation (detailed): `call` / **`function`** / `return`

The callee is entered:

`function` *functionName nVars*



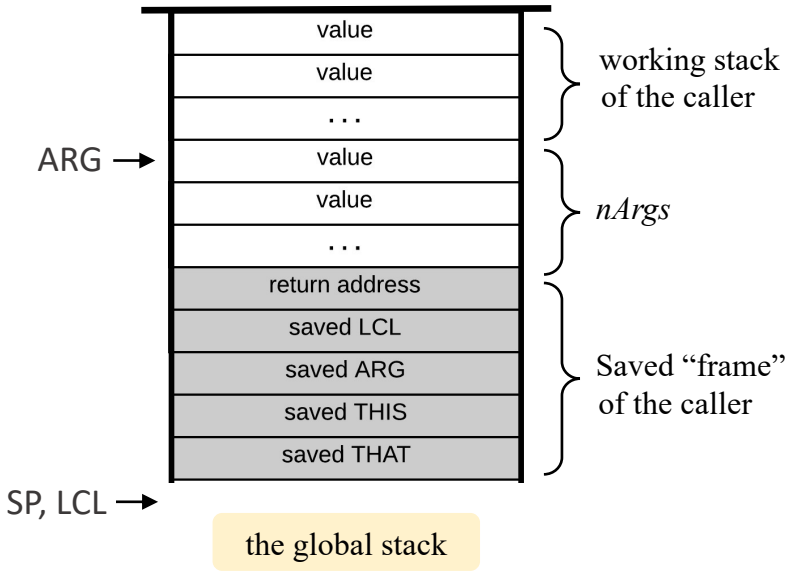
Handling `function` *functionName nVars*

We have to:

Translation (detailed): `call` / **`function`** / `return`

The callee is entered:

`function` *functionName nVars*



Handling `function` *functionName nVars*

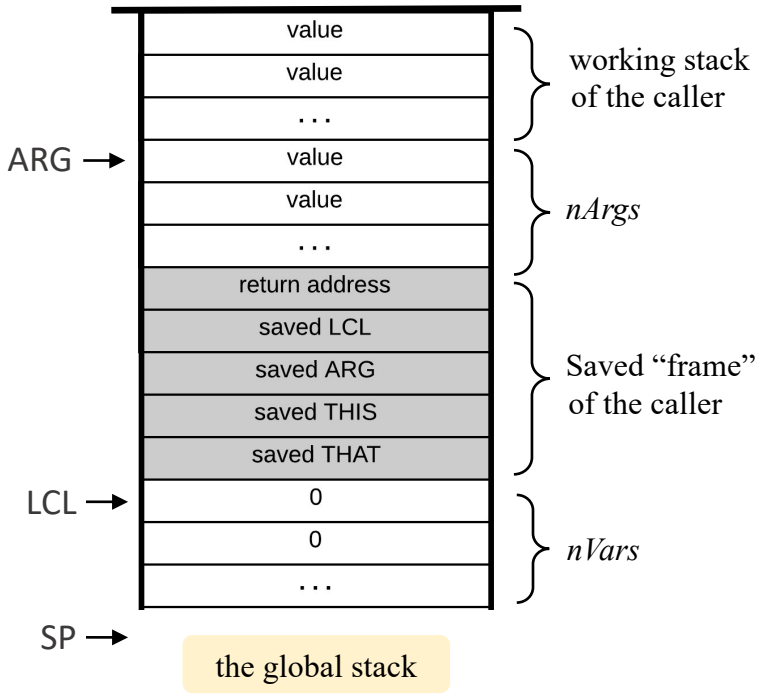
We have to:

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

Translation (detailed): `call` / **`function`** / `return`

The callee is entered:

`function` *functionName nVars*



Handling `function` *functionName nVars*

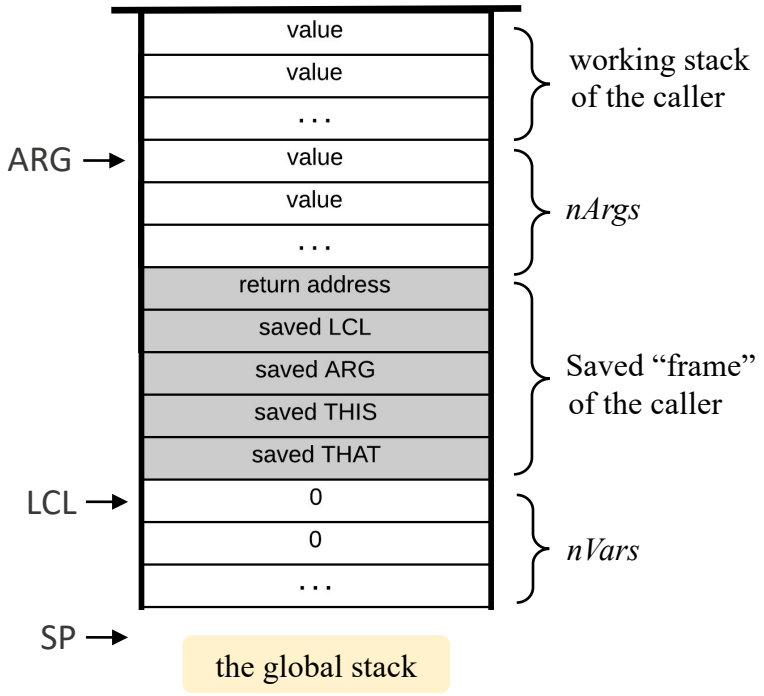
We have to:

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

Translation (detailed): `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

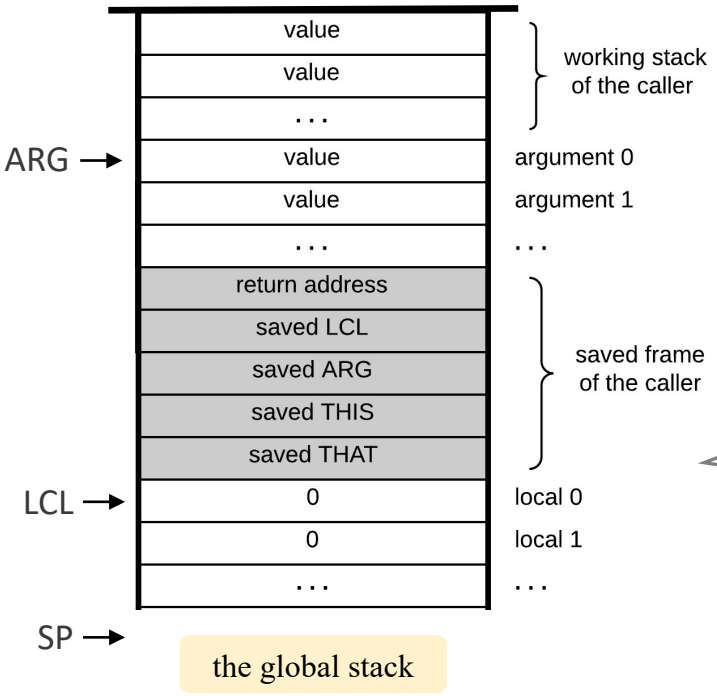
```
// function functionName nVars
(functionName)          // function's entry point (injected label)
// push nVars 0 values (initializes the callee's local variables)
push 0
...
push 0
```

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

Translation (detailed): `call` / **`function`** / `return`

The callee is entered:

`function` *functionName nVars*

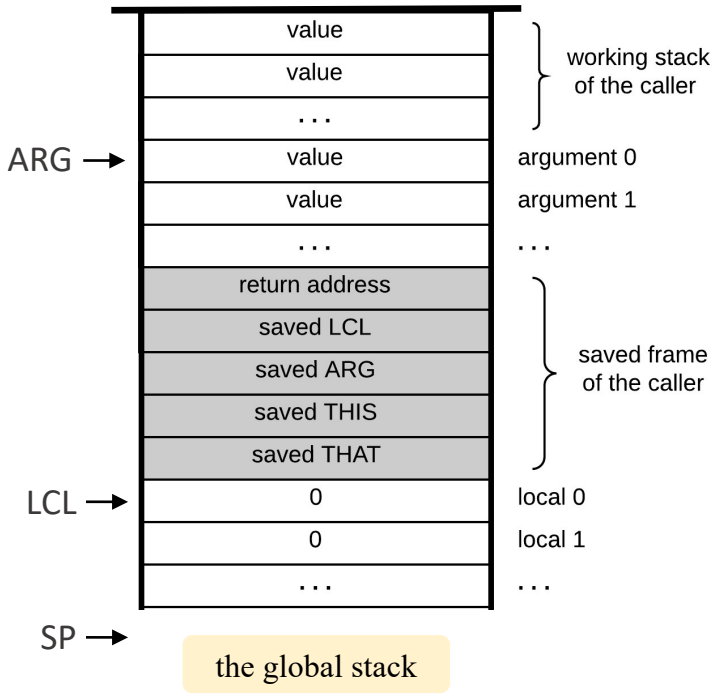


The callee is all set. It has:

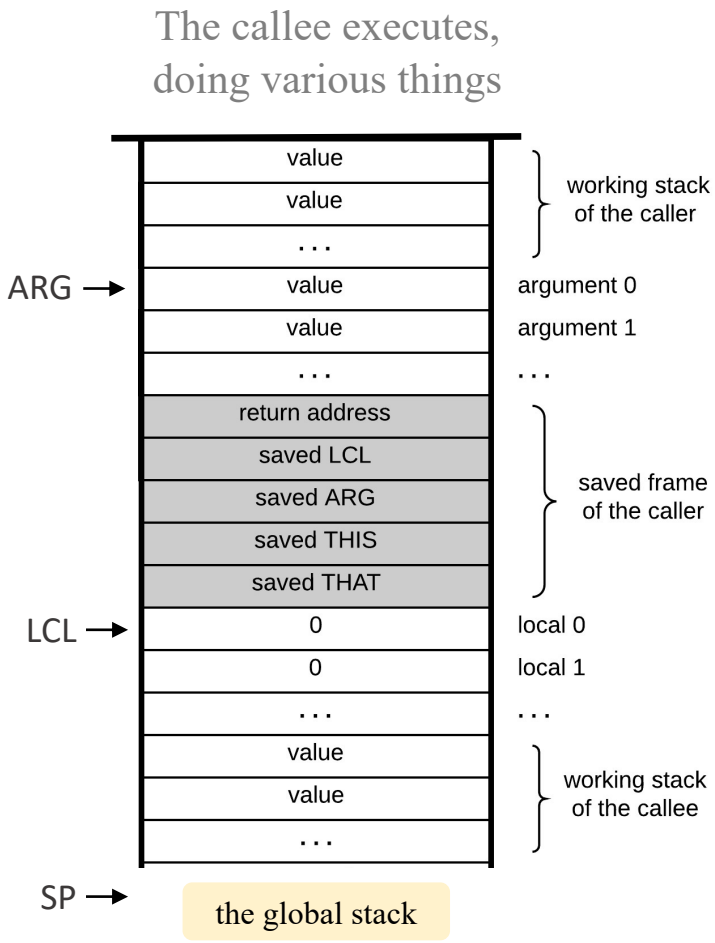
- Arguments (passed by the caller)
- Local variables (all set to 0)
- Working stack (empty)

Translation (detailed): `call / function / return`

The callee executes,
doing various things

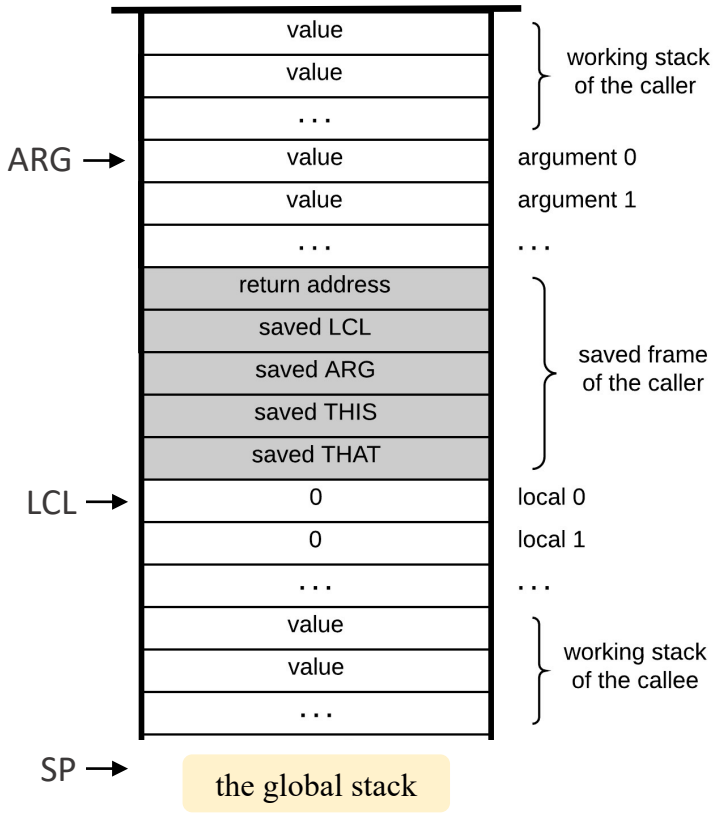


Translation (detailed): `call / function / return`



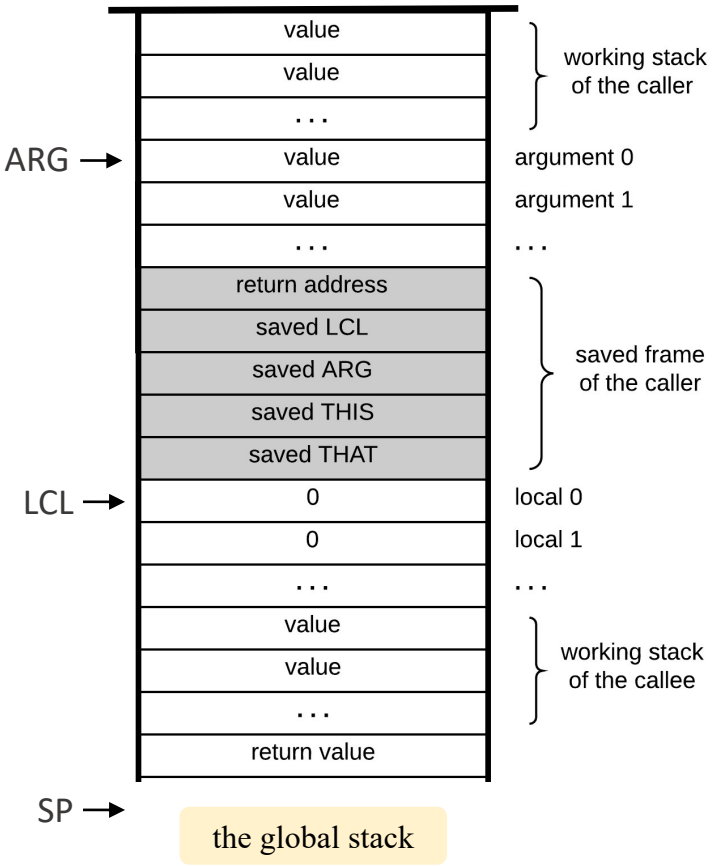
Translation (detailed): `call / function / return`

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



Translation (detailed): `call / function / return`

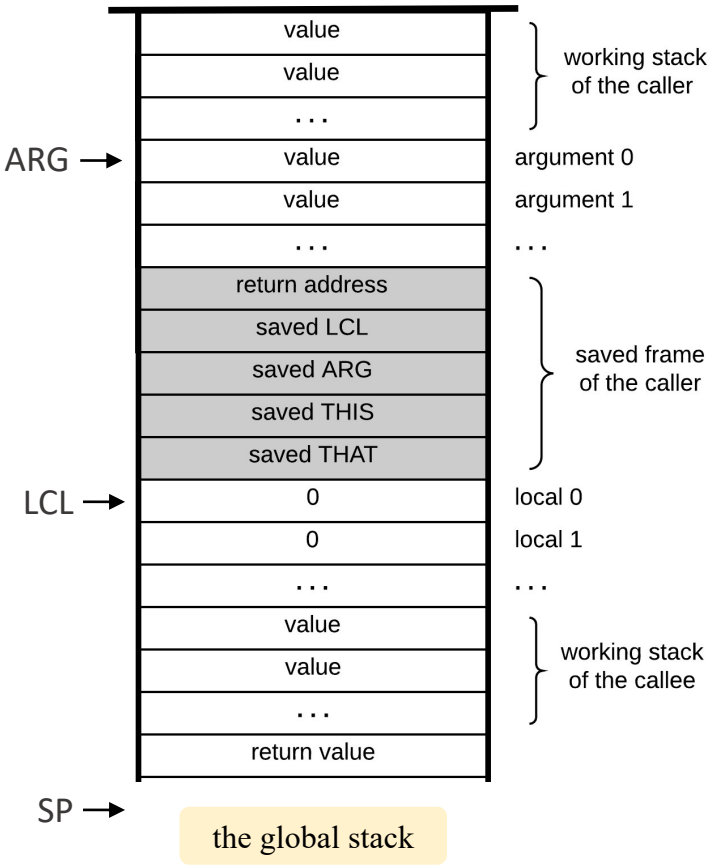
The callee prepares to return:
It pushes a *return value*



Translation (detailed): `call / function / return`

The callee says:

`return`



the global stack

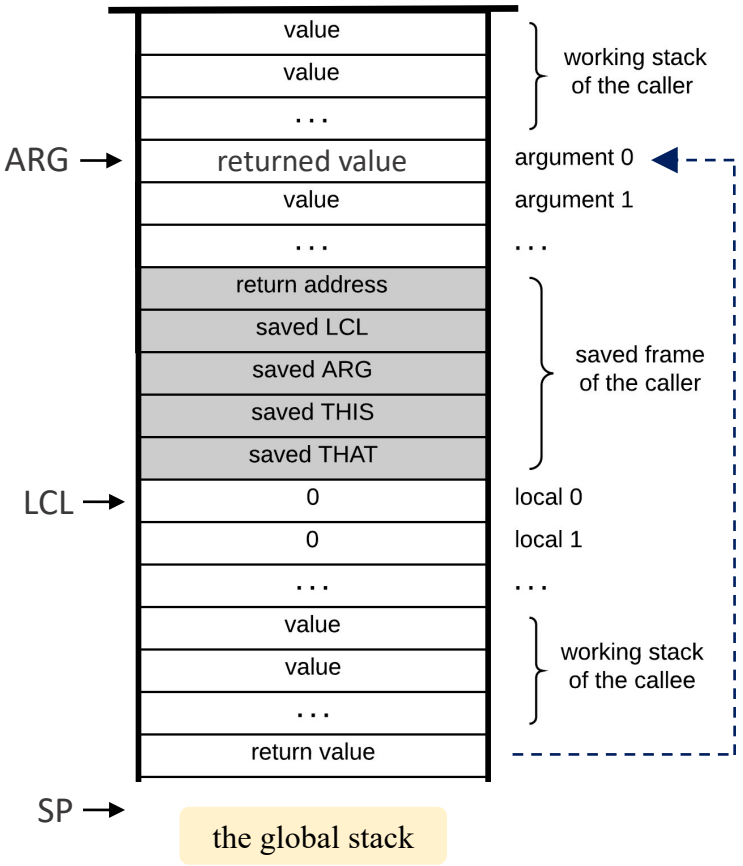
Handling `return`:

We have to:

Translation (detailed): `call / function / return`

The callee says:

`return`



Handling `return`:

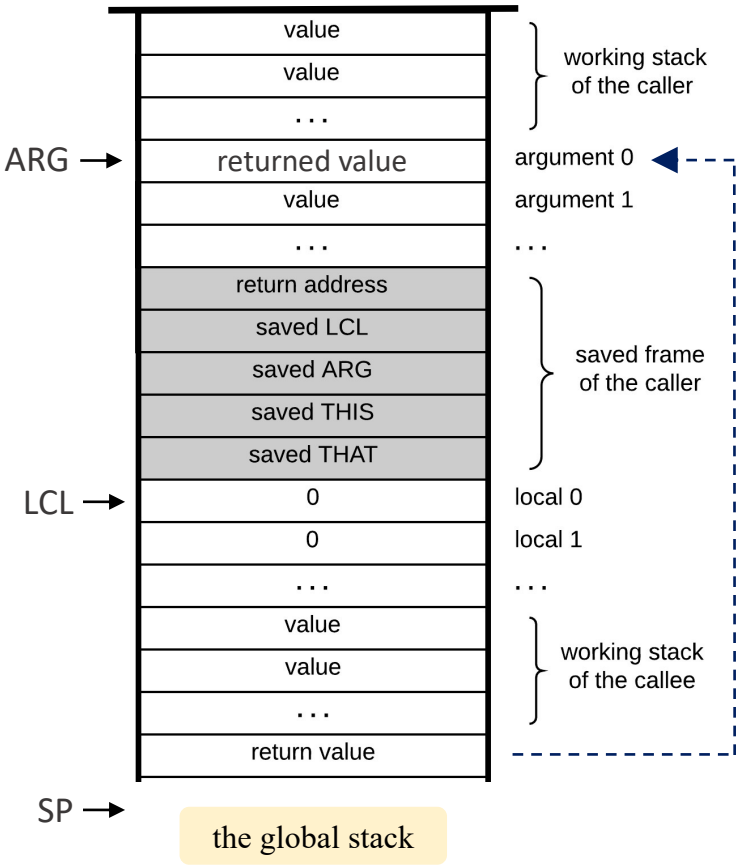
We have to:

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

Translation (detailed): `call / function / return`

The callee says:

`return`



Handling `return`:

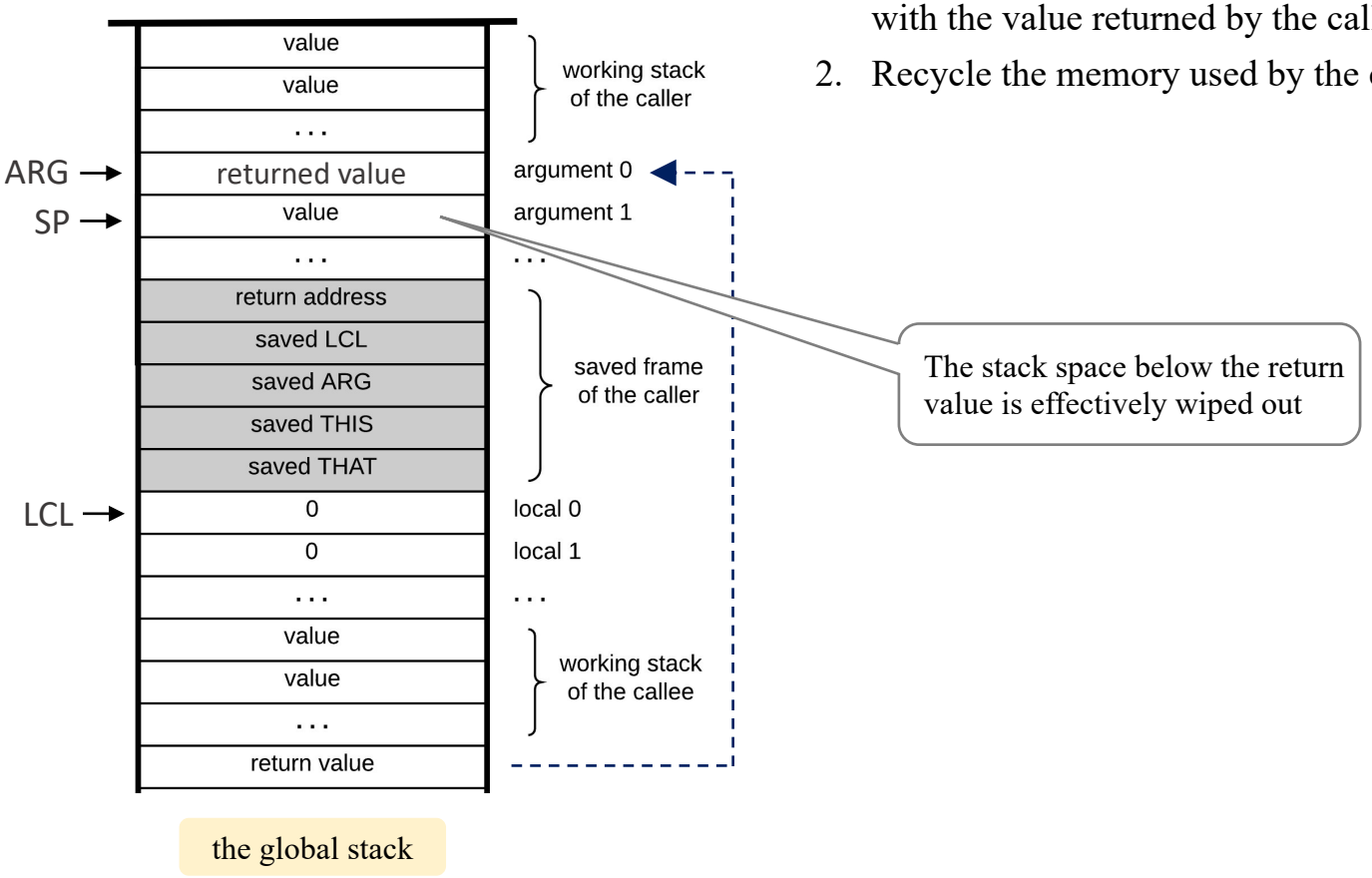
We have to:

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

Translation (detailed): `call / function / return`

The callee says:

`return`



Handling `return`:

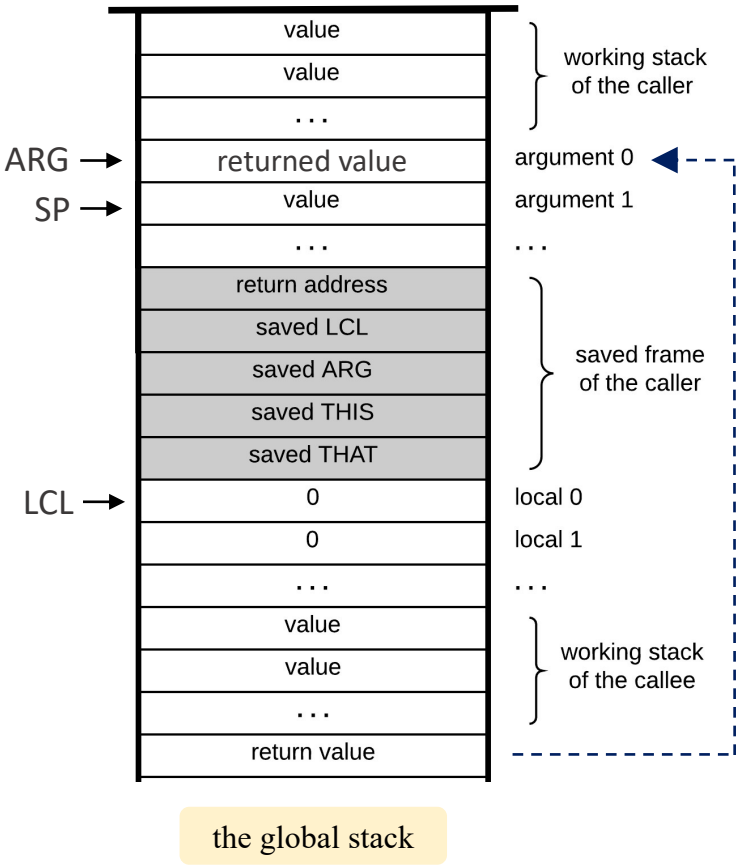
We have to:

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

Translation (detailed): `call / function / return`

The callee says:

`return`



Handling `return`:

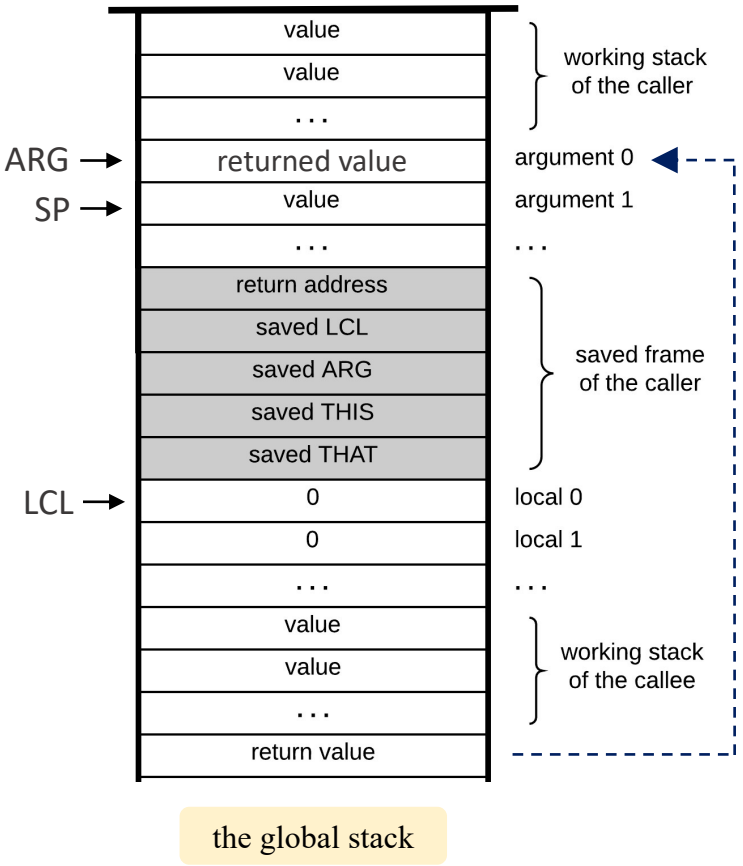
We have to:

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

Translation (detailed): `call / function / return`

The callee says:

`return`



Handling `return`:

We have to:

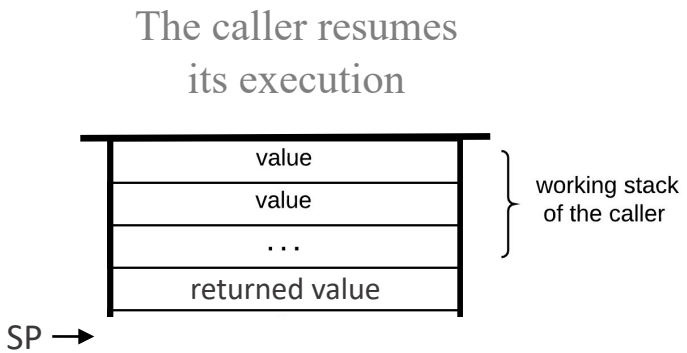
1. Replace the arguments that the caller pushed with the value returned by the callee
2. Recycle the memory used by the callee
3. Restore the caller's segment pointers
4. Jump to the return address

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)

Translation (detailed): `call / function / return`

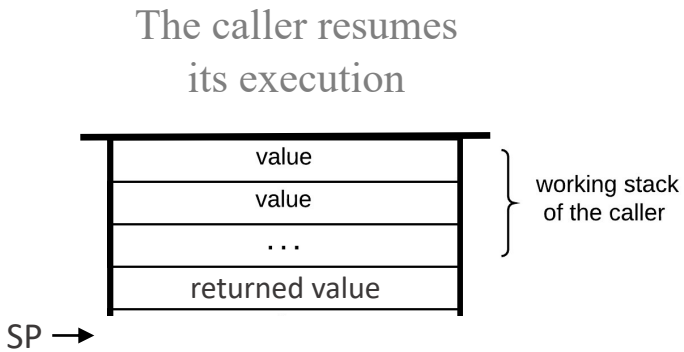


Result: The caller's world is exactly the same as before the call, except that the arguments that it pushed before the call were replaced by the value returned by the callee.

Any sufficiently advanced technology is indistinguishable from magic.

– Arthur C. Clarke, 1962

Translation (detailed): `call / function / return`



What if the calling chain consists of multiple calls?

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

And what about recursion?

Implementation

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

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

Lecture plan

✓ Abstraction

- VM branching commands
- VM function commands

✓ Implementation (conceptual)

- VM branching commands
- VM function commands

Related topics

➡ Bootstrap

- Conventions
- Architecture
- Project 8

The big picture: Compilation

`myProg` folder

`Main.jack`

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

`Foo.jack`

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

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):

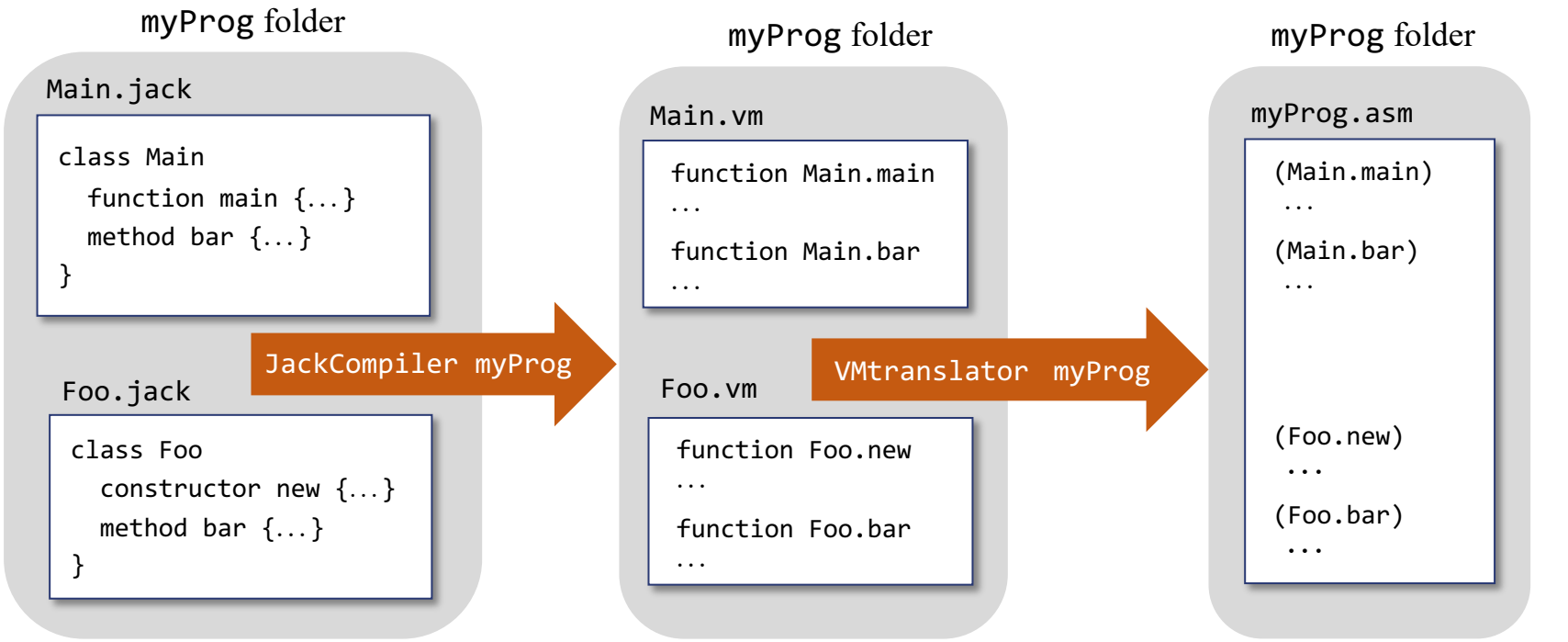
The OS is written in Jack (just like Unix is written in C);

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



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

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

All the VM functions are translated into a single assembly file

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

Bootstrap code

Run-time conventions

The compiled code base includes the program: A set of VM functions, 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` initializes the operating system, calls `Main.main`, and enters an infinite loop

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

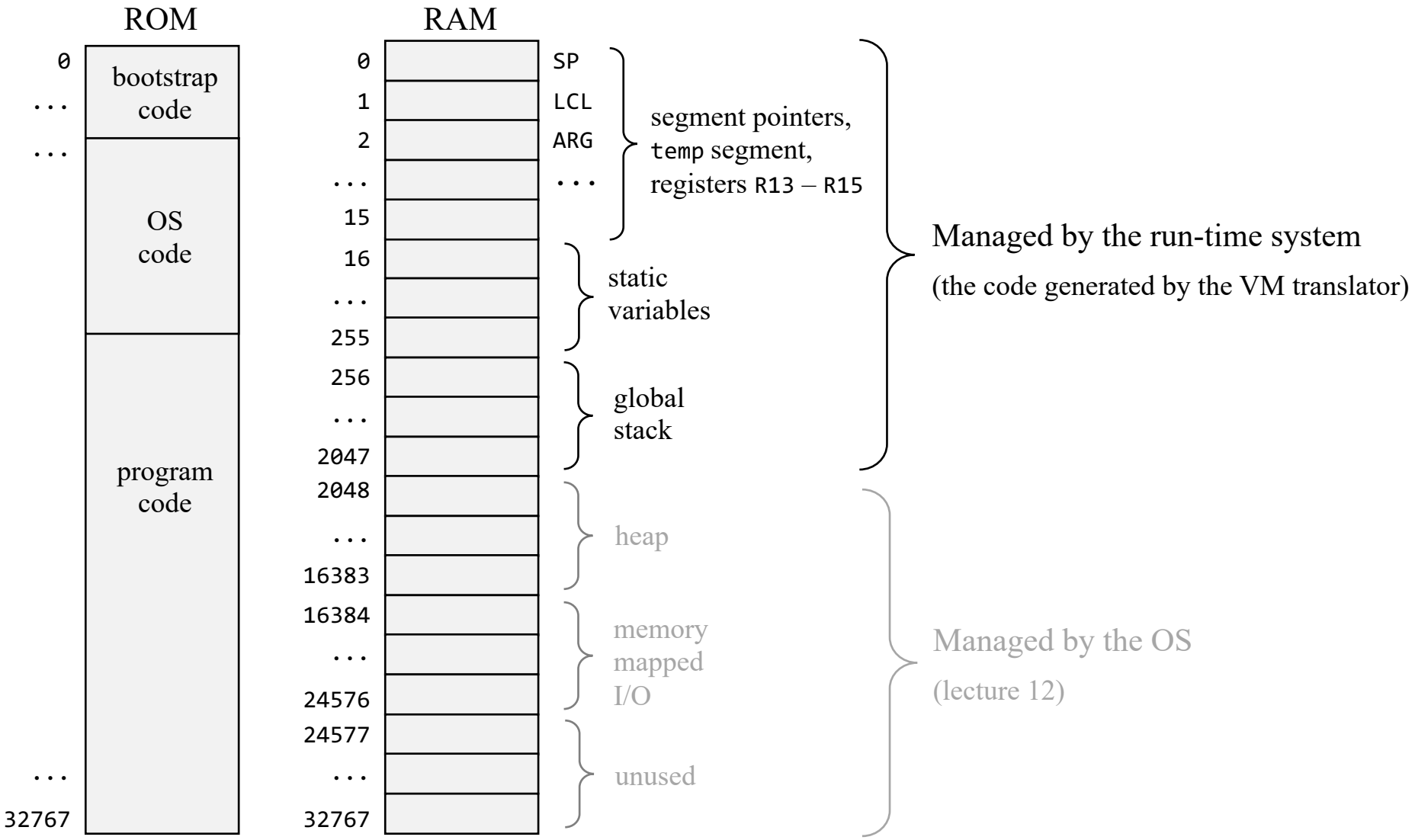
To make this happen:

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 mapping of the VM on the Hack platform



Symbols

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.

Naming conventions,

Read carefully for project 8.

Lecture plan

✓ Abstraction

- VM branching commands
- VM function commands

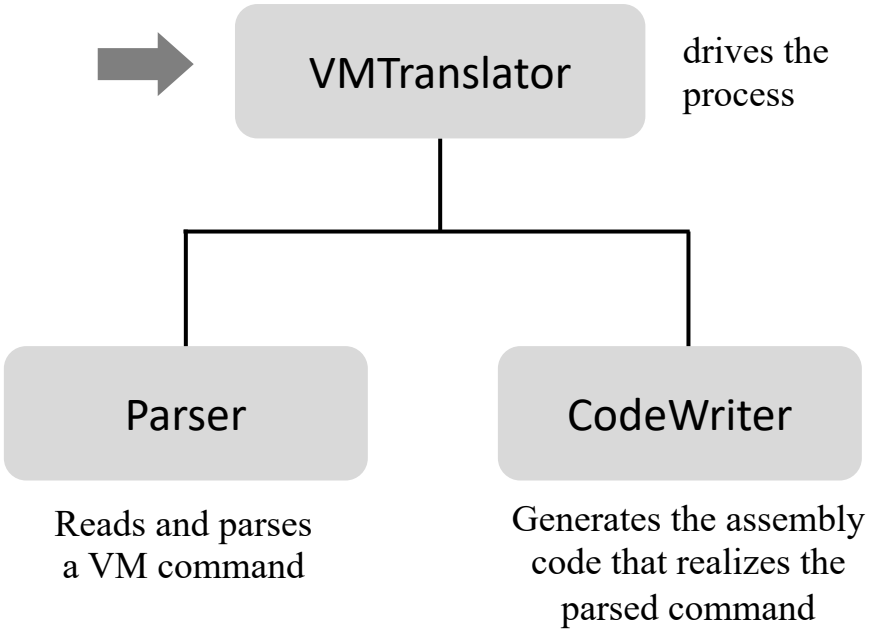
✓ Implementation (conceptual)

- VM branching commands
- VM function commands

VM translator

- Bootstrap
- Conventions
- Architecture
- Project 8

VM translator



Each module extends the corresponding module developed in project 7,

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

VM translator

Usage: (if the translator is implemented in Java)

```
$ java VMTranslator source
```

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

(The *source* may contain a file path; the first character of *filename* must be an uppercase letter)

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

Action

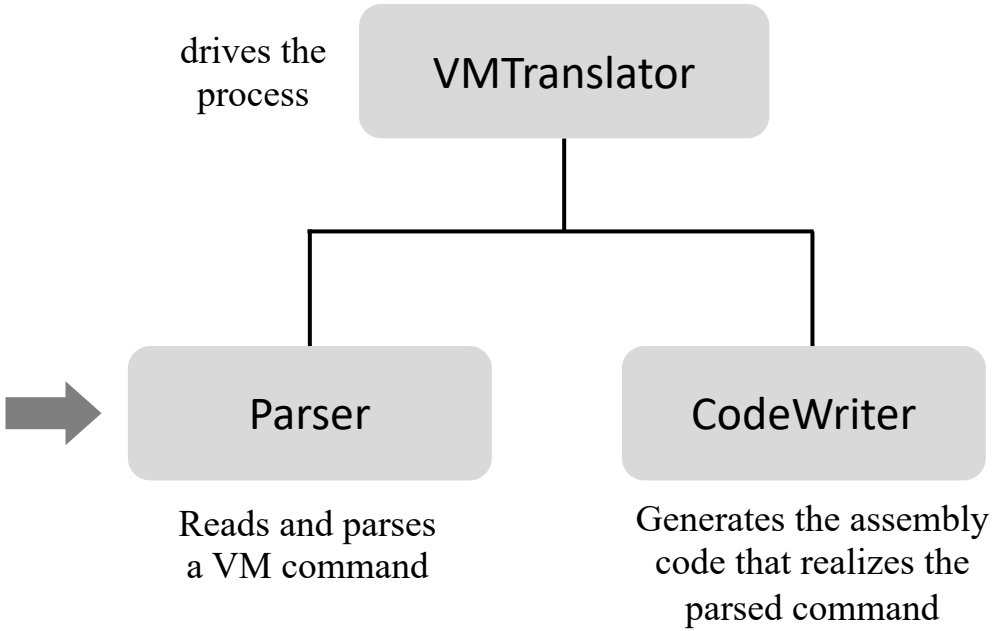
- Constructs a `CodeWriter`
- If *source* is a `.vm` file:

  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 *source* is a folder:

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

VM translator



The same `Parser` developed in project 7:

Handles the parsing of a `.vm` file:

- Skips white space and comments;
- Reads a VM command, parses the command into its lexical components, and provides convenient access to these components

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;

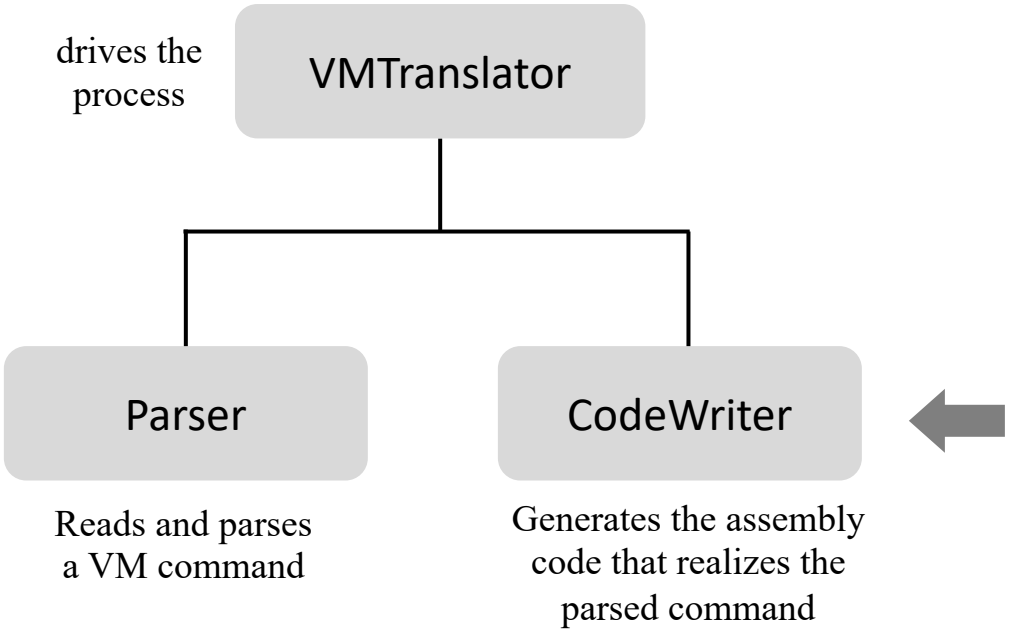
If your project 7 `Parser` did not handle the parsing of the VM commands:

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

`call`, `function`, `return`,

add this parsing functionality now.

VM translator: Proposed design



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 adhere to the symbol naming conventions.

Lecture plan

Abstraction

- VM branching commands
- VM function commands

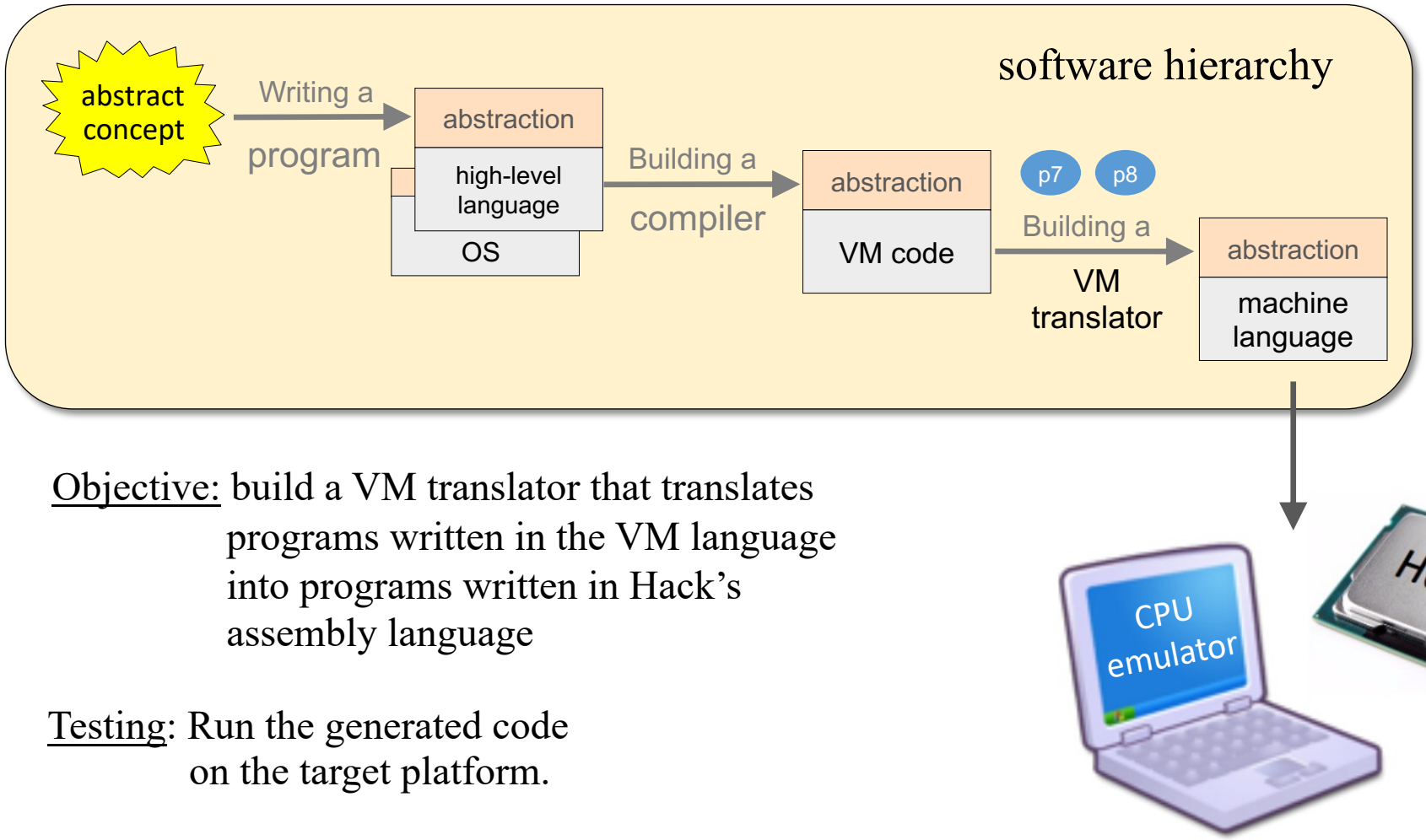
Implementation (conceptual)

- VM branching commands
- VM function commands

VM translator

- Bootstrap
- Conventions
- Architecture
- 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
- FibonacciElement
- StaticsTest

Test programs: BasicLoop

ProgramFlow:

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

FunctionCalls:

- SimpleFunction
- 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

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- FibonacciElement
- StaticsTest

Test programs: FibonacciSeries

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

FunctionCalls:
- SimpleFunction
- 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

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- FibonacciElement
- StaticsTest

Test programs: SimpleFunction

ProgramFlow:
- BasicLoop
- FibonacciSeries

FunctionCalls:
- SimpleFunction
  - **SimpleFunction.vm**
  - SimpleFunctionVME.tst
  - SimpleFunction.tst
  - SimpleFunction.cmp
- 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

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- FibonacciElement
- StaticsTest

Test programs: FibonacciElement

ProgramFlow:
- BasicLoop
- FibonacciSeries

FunctionCalls:
- SimpleFunction
- 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
```

- 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 bootstrap code initializes the stack and calls `Sys.init`

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
```

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

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

Test programs: FibonacciElement

- ProgramFlow:
  - BasicLoop
  - FibonacciSeries
- FunctionCalls:
  - SimpleFunction
  - 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
```

Usage: `$ VMTranslator FibonacciElement` (translates a *folder*)

(Should generates a single output file: `FibonacciElement.asm`)

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

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- FibonacciElement
- StaticsTest

Test programs: StaticsTest

- ProgramFlow:
  - BasicLoop
  - FibonacciSeries
- FunctionCalls:
  - SimpleFunction
  - FibonacciElement
  - StaticsTest
    - **Class1.vm**
    - **Class2.vm**
    - **Sys.vm**
    - StaticsTestVME.tst
    - StaticsTest.cmp

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

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

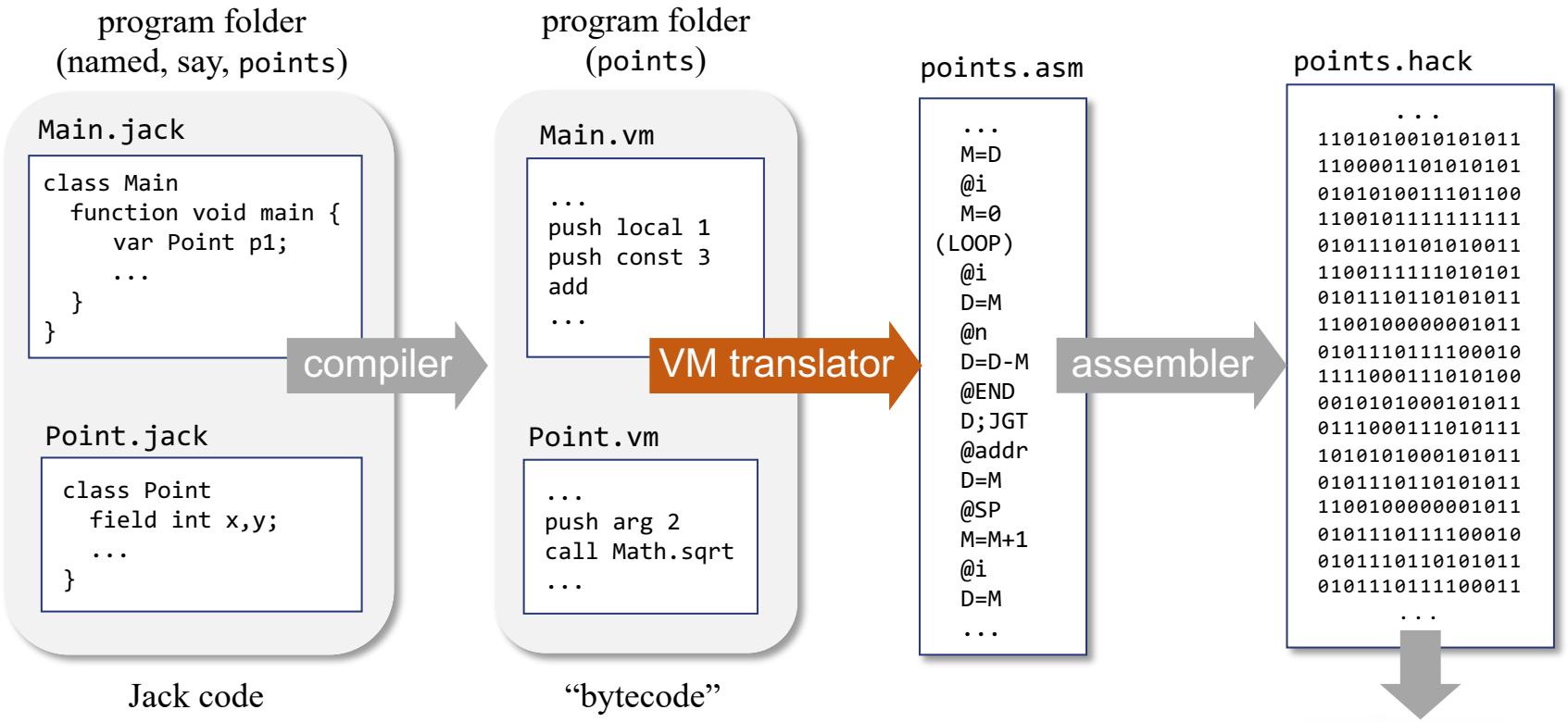
- ProgramFlow:
  - BasicLoop
  - FibonacciSeries
- FunctionCalls:
  - SimpleFunction
  - FibonacciElement
  - StaticsTest

Testing routine for every test program Xxx:

0. Recommended: Load and run XxxVME.tst on the VM emulator;
   This script loads the Xxx 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 (if the test includes more than one .vm file, apply your translator to the folder name)
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 the files mentioned above are supplied, except for Xxx.asm

Recap / Next



The VM translator developed in projects 7 – 8 is the compiler's backend.

Next: Introducing the Jack language (project 9) and completing the compiler's frontend (projects 10 – 11).