Chapter 8

Virtual Machine

(Part II)

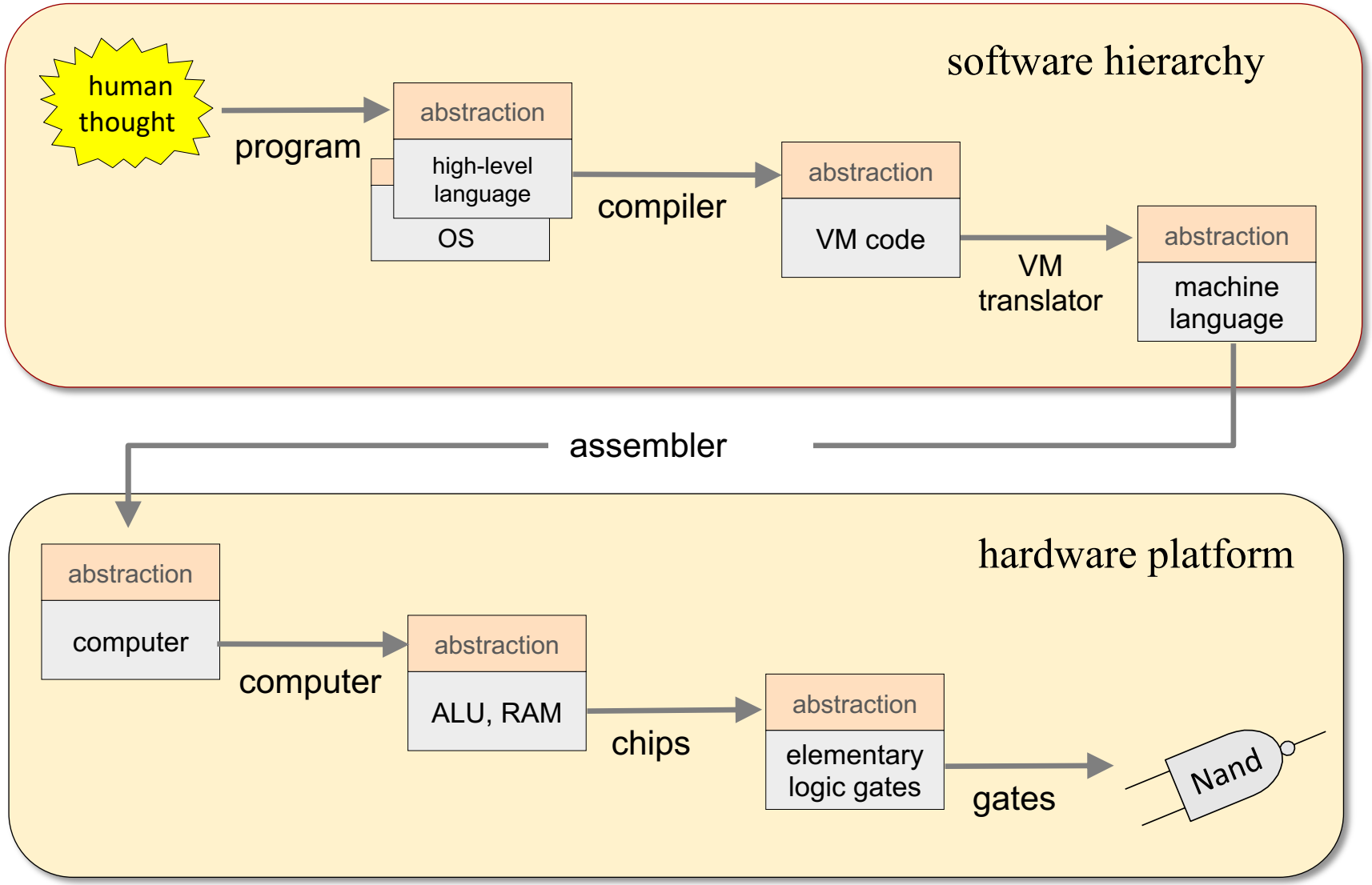
These slides support chapter 8 of the book

The Elements of Computing Systems

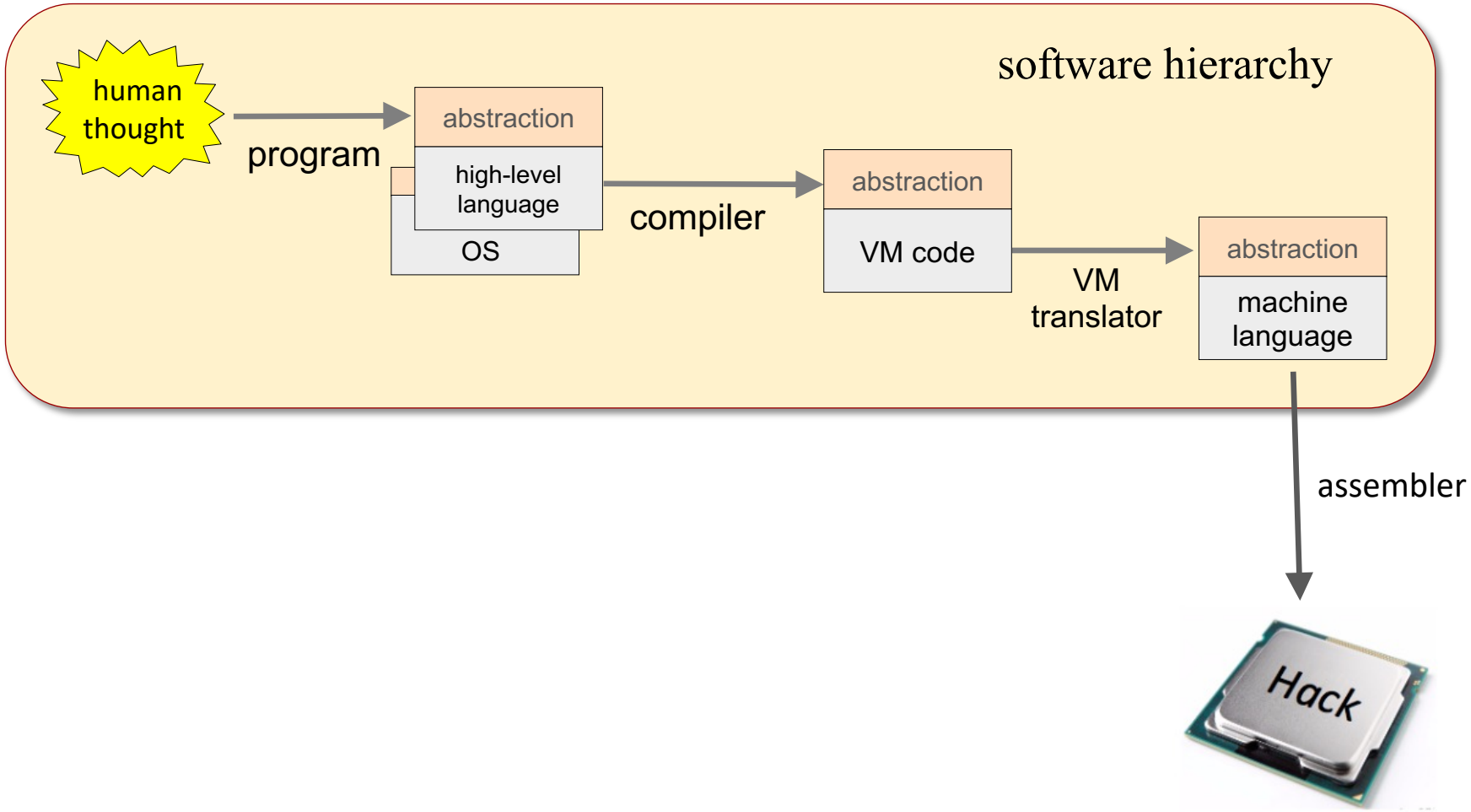
By Noam Nisan and Shimon Schocken

MIT Press

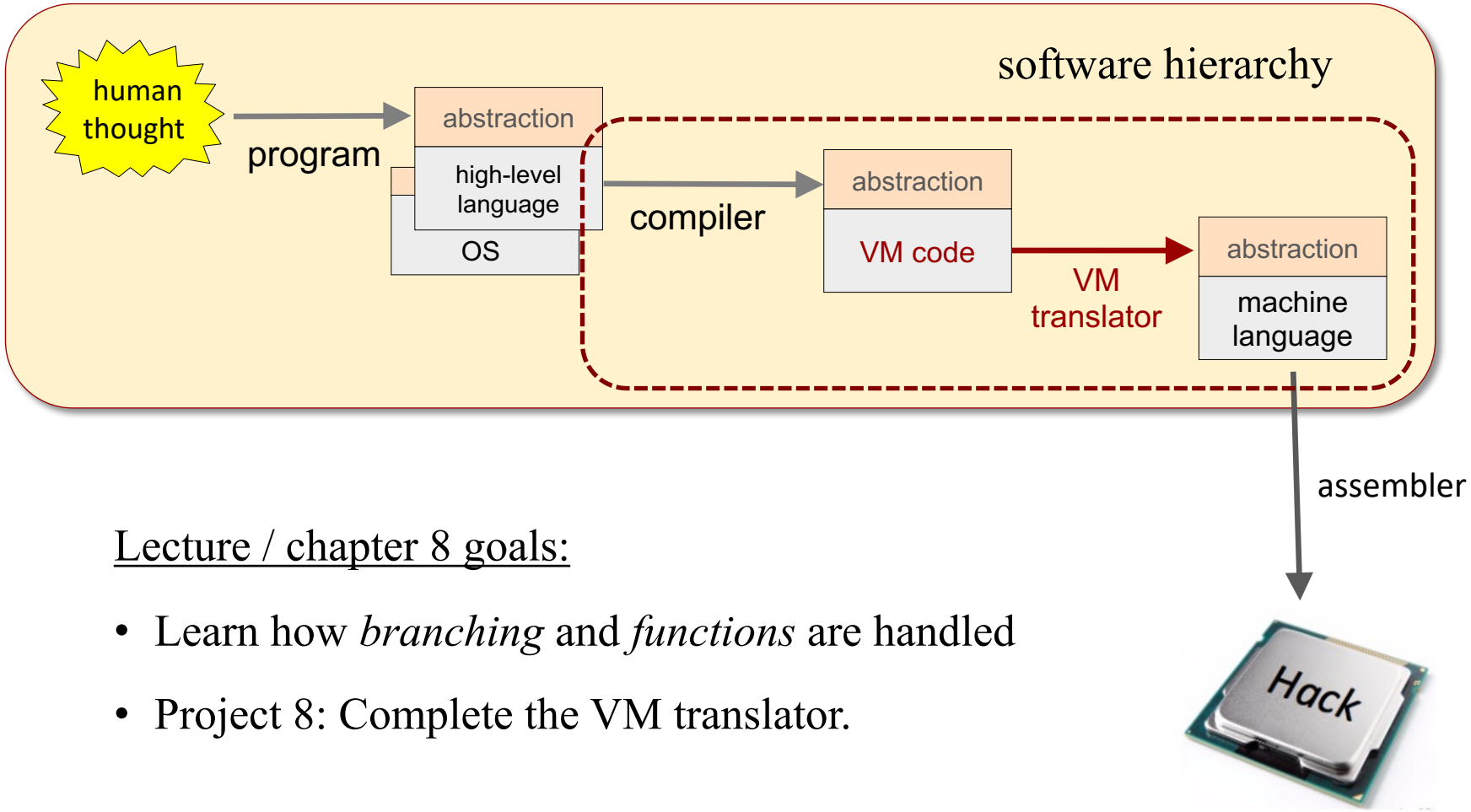
Nand to Tetris Roadmap



Nand to Tetris Roadmap: Part II



Nand to Tetris Roadmap: Part II



Lecture / chapter 8 goals:

- Learn how *branching* and *functions* are handled
- Project 8: 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;
}
```

compiler

(chapters 10 – 11)

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 commands

- `add`, `sub` , `neg`
- `eq` , `gt` , `lt`
- `and`, `or` , `not`

Push / pop commands

- `pop`
- `push`

"imperatives"

Branching commands

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

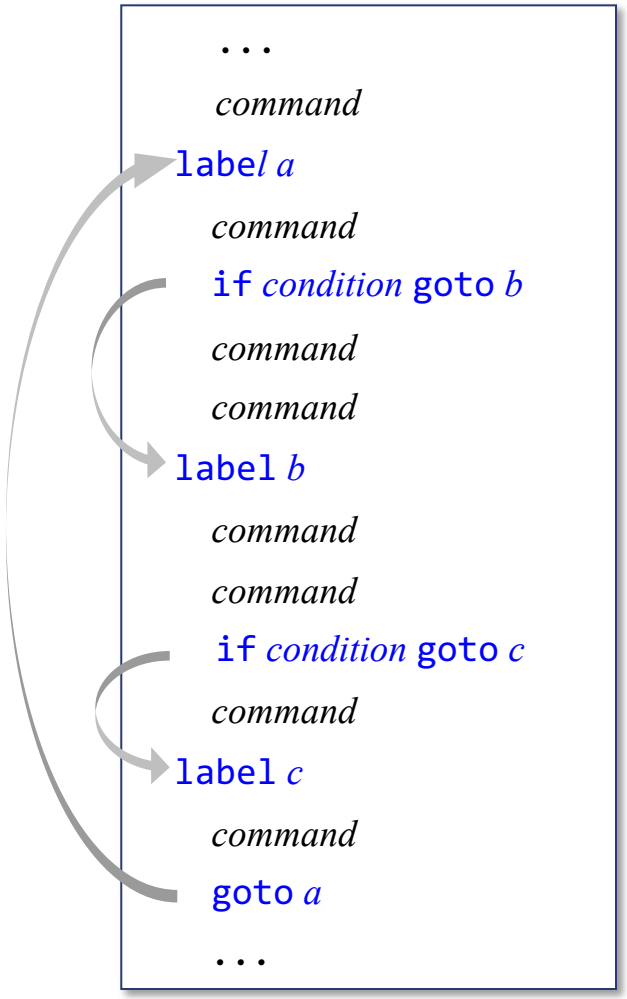
Function commands

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

"control"

Branching

Programs typically include branching logic (conditions, loops):



Branching:

- Unconditional
- Conditional

Implementation issues

- How to realize conditions?
- How to realize labels?
- How to realize branching operations?

Functions

Programs are typically divided into "*program units*":

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

We'll use the term *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

One function (like `foo`) calls another function (like `mult`) for its effect.

Implementation issues

How to pass arguments from the caller to the callee?

How to start executing the caller'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 how programs are executed:

- Branching
- Function call-and-return
- Recursion

Related issues:

- Dynamic memory management
- Stack processing
- Run-time system
- Completing the compiler's backend (VM translator).

Branching commands

VM branching commands

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

VM code (with symbols)

```
// 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 has no symbolic variables
- Our VM code examples sometimes use symbols (like `x`, `y`, `sum`, `n`), to make the code more readable

Branching commands

VM branching commands

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

VM code (with symbols)

```
// 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
```

Syntax

`label` *label*

Semantics

Marks the destination of `goto` commands.

Branching commands

VM branching commands

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

VM code (with symbols)

```
// 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
```

Syntax

`goto` *label*

Semantics

Jump to execute the command just after *label*.

Branching commands

VM branching commands

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

VM code (with symbols)

```
// 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
```

Syntax

`if-goto` *label*

Semantics

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

Branching commands

VM branching commands

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

VM code (with symbols)

```
// 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
```

Syntax

`if-goto` *label*

Semantics

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

Assumption:

The code writer is expected to write code that pushes the condition (a logical expression) to the stack before the `if-goto` command.

In this example, the highlighted code implements the abstraction:
`if (n > y) goto END_LOOP`

Branching commands

VM branching commands

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

VM code (with symbols)

```
// 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
```

Exercise

Now that we've described the VM branching commands, inspect the code and convince yourself that it computes `x * y`

Click "Tip" if you need one

Branching commands

VM branching commands

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

VM code (with symbols)

```
// 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
```

Exercise

Now that we've described the VM branching commands, inspect the code and convince yourself that it computes `x * y`

Tip:

The code adds `x` to `sum`, `y` times.

Lecture plan

Abstraction

- VM branching commands
- VM function commands

Implementation

- VM branching commands
- VM function commands

The Virtual Machine

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

Function commands

VM function commands

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

VM code (with symbols)

```
// 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
```

callee

```
...
// Computes 8 * 5 + 7
push 8
push 5
call mult 2
push 7
add
...
```

caller

Such a calling code pattern can appear in any VM function

Typical scenario

The *caller* calls the *callee*, for its effect

Function commands

VM function commands

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

VM code (with symbols)

```
// 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
```

callee

```
...
// Computes 8 * 5 + 7
push 8
push 5
call mult 2
push 7
add
...
```

caller

Such a calling code pattern can appear in any VM function

Syntax

`call` *functionName nArgs*

Semantics

Calls the function *functionName* for its effect, informing that *nArgs* argument values were pushed onto the stack prior to the call

Function commands

VM function commands

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

VM code (with symbols)

```
// 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
```

callee

```
...
// Computes 8 * 5 + 7
push 8
push 5
call mult 2
push 7
add
...
```

caller

Syntax

`call` *functionName nArgs*

Assumptions

- The code writer is expected to push *nArg* arguments onto the stack before the `call` command
- When *functionName* terminates, the argument values pushed by the caller will be replaced with the value returned by the callee.

Function commands

VM function commands

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

VM code (with symbols)

```
// 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
```

callee

```
...
// Computes 8 * 5 + 7
push 8
push 5
call mult 2
push 7
add
...
```

caller

Syntax

`function` *functionName nVars*

Semantics

- Here starts the declaration of a function that has name *functionName* and *nVars* local variables
- Marks the entry point (destination) of `call` *functionName* commands

Function commands

VM function commands

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

VM code (with symbols)

```
// 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
```

callee

```
...
// Computes 8 * 5 + 7
push 8
push 5
call mult 2
push 7
add
...
```

caller

Syntax

`function` *functionName nVars*

Assumption

The function will have *nVars* local variables, named `local 0`, `local 1`, ... `local` *nVars*

(The example uses symbols, for readability. In the final code, `sum` and `n` will be replaced with `local 0` and `local 1`, and `x` and `y` with `argument 0` and `argument 1`)

Function commands

VM function commands

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

VM code (with symbols)

```
// 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
```

callee

```
...
// Computes 8 * 5 + 7
push 8
push 5
call mult 2
push 7
add
...
```

caller

Syntax

`return`

Semantics

- Ends the function declaration
- Indicates that control will be transferred to the command just after the `call` command that invoked this function.

Function commands

VM function commands

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

VM code (with symbols)

```
// 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
```

callee

```
...
// Computes 8 * 5 + 7
push 8
push 5
call mult 2
push 7
add
...
```

caller

Syntax

`return`

Assumption:

The writer of the callee's code is expected to write code that pushes a value onto the stack before the `return` command. This value is called the *return value*.

In this example, the return value is the value of `sum`, and control will be passed to the highlighted command.

Lecture plan

✓ Abstraction

- VM branching commands
- VM function commands

Implementation

➡ VM branching commands

- VM function commands

The Virtual Machine

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

Implementation: Branching

VM branching commands

`label` *label* // label declaration command

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

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

Implementation (on some target platform):

For each VM branching command, we have to generate machine language instructions that realize the command on the target platform (like we did in project 7)

Example:

VM code

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

Hack code

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

Observation: The instruction set of any target platform (like Hack) features means for implementing "labeling" and "goto" semantics

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

Lecture plan

✓ Abstraction

- VM branching commands
- VM function commands

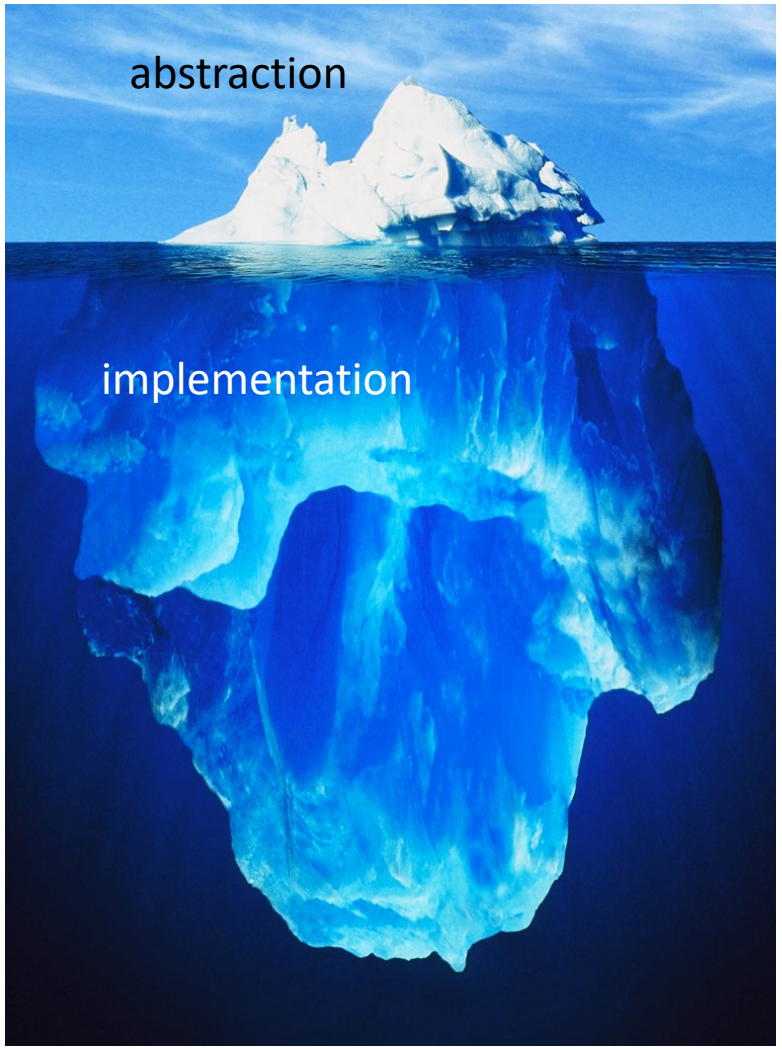
Implementation

✓ VM branching commands

➡ VM function commands

The Virtual Machine

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



Function call and return: Abstraction (revisited)

```
   // Computes 3 + 8 * 5
0  function main 0
1    push constant 3
2    push constant 8
3    push constant 5
4    call mult 2
5    add
6    return
```

caller

Typical function-call-and-return scenario

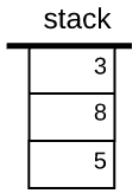
Function `main` (the *caller*) calls function `mult` (the *callee*) for its effect

```
   // 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
5  label LOOP
6    push local 1
7    push argument 1
...  ...
     // Returns the result
20   push local 0
21   return
```

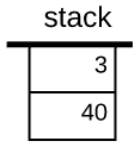
callee

`main` view:

after line 3 is executed:



after line 4 is executed:



Magic!

Let's zoom in on the magic, step by step

Function call and return: Abstraction

```
   // Computes 3 + 8 * 5
0  function main 0
1    push constant 3
2    push constant 8
3    push constant 5
4    call mult 2
5    add
6    return
```

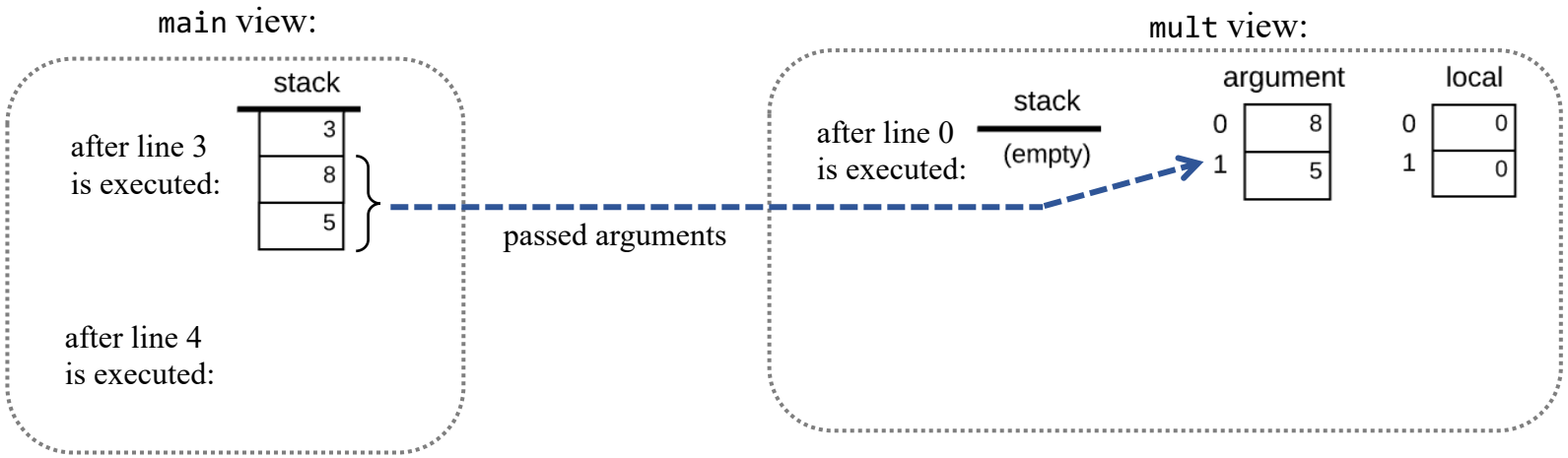
caller

Typical function-call-and-return scenario

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

```
   // 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
5  label LOOP
6    push local 1
7    push argument 1
...  ...
     // Returns the result
20   push local 0
21   return
```

callee



The caller suspends its execution, waiting for the callee to return

Function call and return: Abstraction

```
   // Computes 3 + 8 * 5
0  function main 0
1    push constant 3
2    push constant 8
3    push constant 5
4    call mult 2
5    add
6    return
```

caller

Typical function-call-and-return scenario

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

```
   // 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
5  label LOOP
6    push local 1
7    push argument 1
...  ...
     // Returns the result
20   push local 0
21   return
```

callee

`main` view:

after line 3 is executed:

stack

3
8
5

after line 4 is executed:

`mult` view:

after line 0 is executed:

stack: (empty)

argument

0	8
1	5

local

0	0
1	0

after line 20 is executed:

stack

...
40

argument

0	8
1	5

local

0	40
1	6

The caller suspends its execution, waiting for the callee to return

Function call and return: Abstraction

```
   // Computes 3 + 8 * 5
0  function main 0
1    push constant 3
2    push constant 8
3    push constant 5
4    call mult 2
5    add
6    return
```

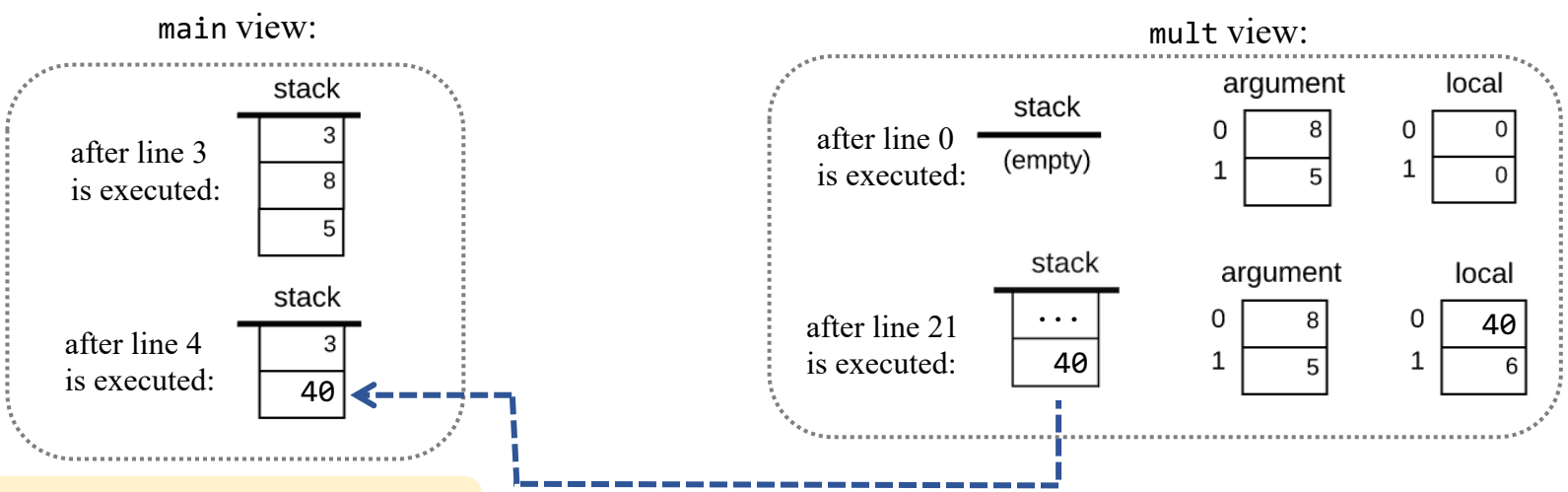
caller

Typical function-call-and-return scenario

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

```
   // 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
5  label LOOP
6    push local 1
7    push argument 1
...  ...
     // Returns the result
20   push local 0
21   return
```

callee



The arguments that the caller pushed were replaced with the return value

Function call and return: Abstraction

```
   // Computes 3 + 8 * 5
0  function main 0
1    push constant 3
2    push constant 8
3    push constant 5
4    call mult 2
5    add
6    return
```

caller

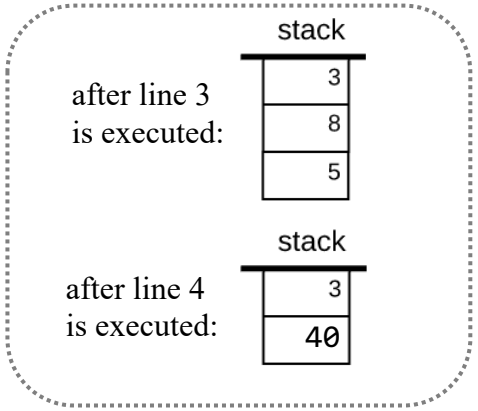
Typical function-call-and-return scenario

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

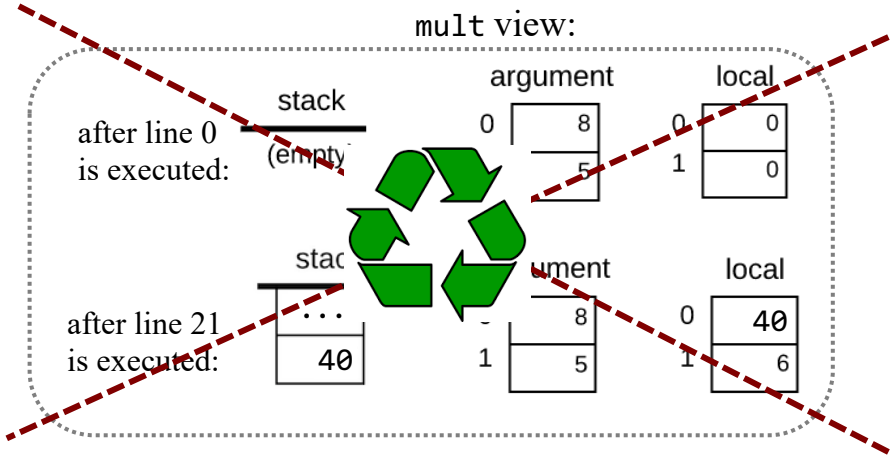
```
   // Returns arg 0 * arg 1
0  function mult 2
1    push constant 0
2    pop local 0
3    push constant 1
4    pop local 1
5  label LOOP
6    push local 1
7    push argument 1
...  ...
     // Returns the result
20   push local 0
21   return
```

callee

`main` view:



`mult` view:



Function call and return: Abstraction

```
// Computes 3 + 8 * 5
0  function main 0
1    push constant 3
2    push constant 8
3    push constant 5
4    call mult 2
5    add
6    return
```

caller

Typical function-call-and-return scenario

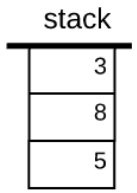
Function `main` (the *caller*) calls function `mult` (the *callee*) for its effect

```
   // 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
5  label LOOP
6    push local 1
7    push argument 1
...  ...
     // Returns the result
20   push local 0
21   return
```

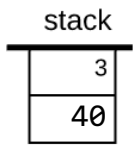
callee

`main` view:

after line 3 is executed:



after line 4 is executed:

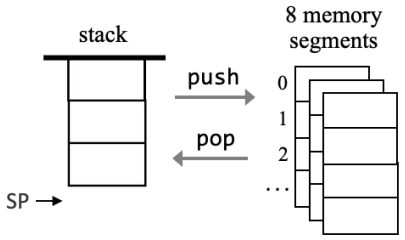


The main function resumes its execution

Function call and return: Implementation

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

caller

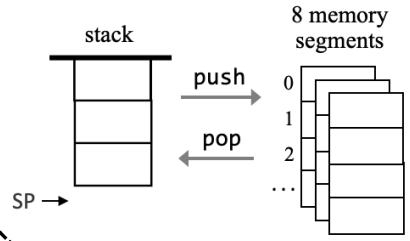


Abstraction (recap)

- A VM program consists of one or more VM functions
- Each function sees its own 8 memory segments
- Each function sees its own working stack
- The functions call each other, for their effect
- Argument passing and return value passing happen, somehow.

```
// Returns arg 0 * arg 1
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  ...
label END
 // Returns the result
  push local 0
  return
```

callee



Implementation (by a VM translator)

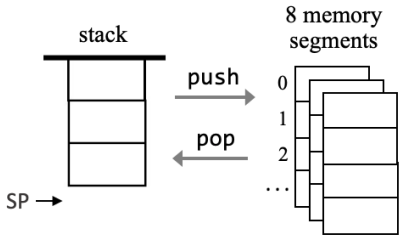
- Input: A text file containing one or more VM functions (like `main` and `mult`)
  Output: A text file containing assembly instructions
- Post translation, the notion of *functions* ceases [illegible] tead, the VM translator creates, and injects into the generated as[illegible] unction *entry* and *return sites*
- The VM translator [illegible]
  - Manages the me[illegible] segments and working stacks of all the functions
  - Handles the passing of arguments and return values.

And now for the details…

Function call and return: Implementation

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

caller

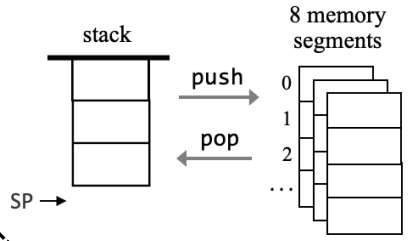


Abstraction (recap)

- A VM program consists of one or more VM functions
- Each function sees its own 8 memory segments
- Each function sees its own working stack
- The functions call each other, for their effect
- Argument passing and return value passing happen, somehow.

```
// Returns arg 0 * arg 1
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  ...
label END
 // Returns the result
  push local 0
  return
```

callee

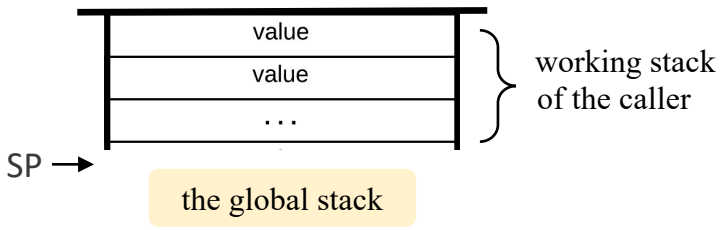


Implementation:

- The working stacks and memory segments of all the functions are managed in the same place: a "global stack" (or simply "stack")
- The semantics of the `function` / `call` / `return` commands (along with the parameter and return values passing) is realized by the machine language instructions generated from each of these commands.

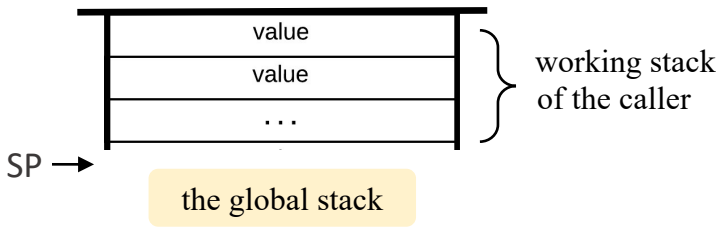
Implementing `call` / `function` / `return`

The caller is running,
doing various things



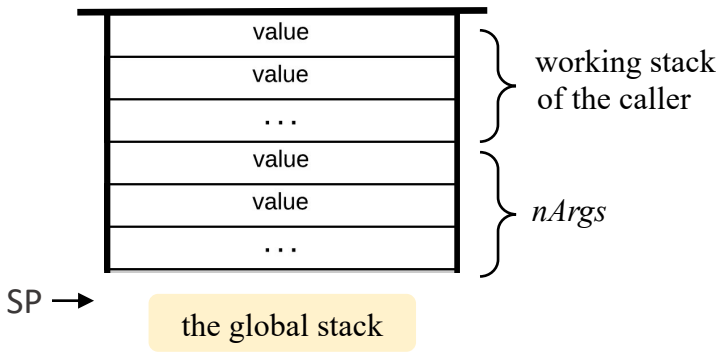
Implementing `call` / `function` / `return`

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



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

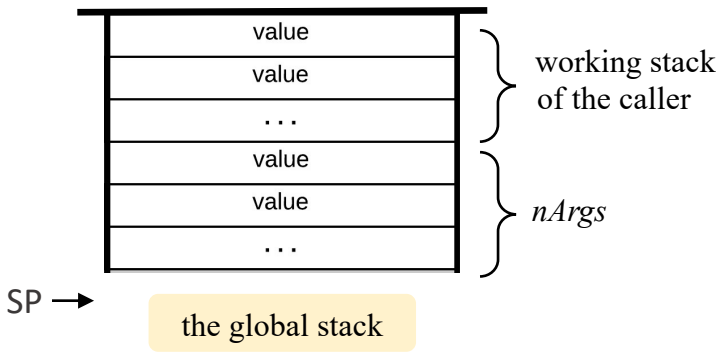
The caller prepares to call another function;
It pushes 0 or more arguments onto the stack



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

The caller says:

`call` *functionName nArgs*



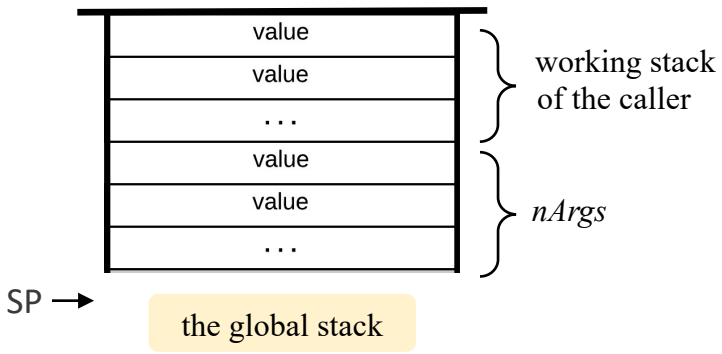
Handling `call` *functionName nArgs*

We have to:

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

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

We have to:

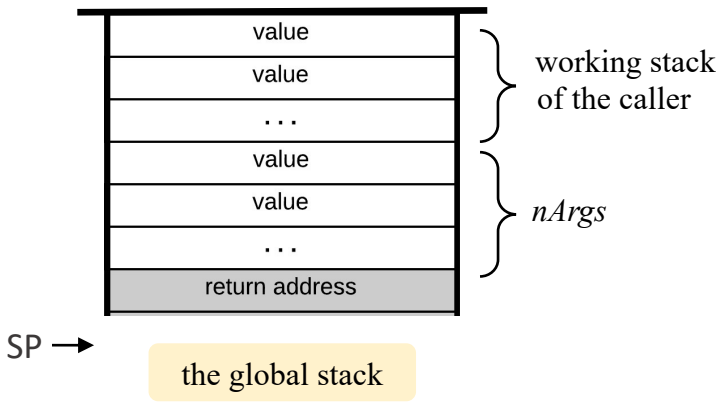
- Save the return address

The address within the code to which we want to return when the callee terminates its execution

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

We have to:

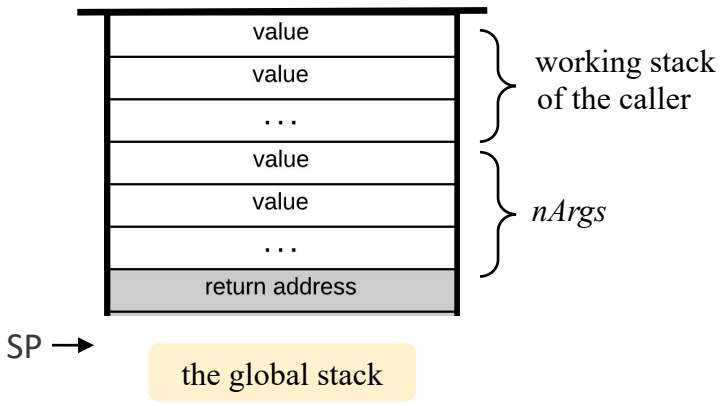
- Save the return address

The address within the code to which we want to return when the callee terminates its execution

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

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

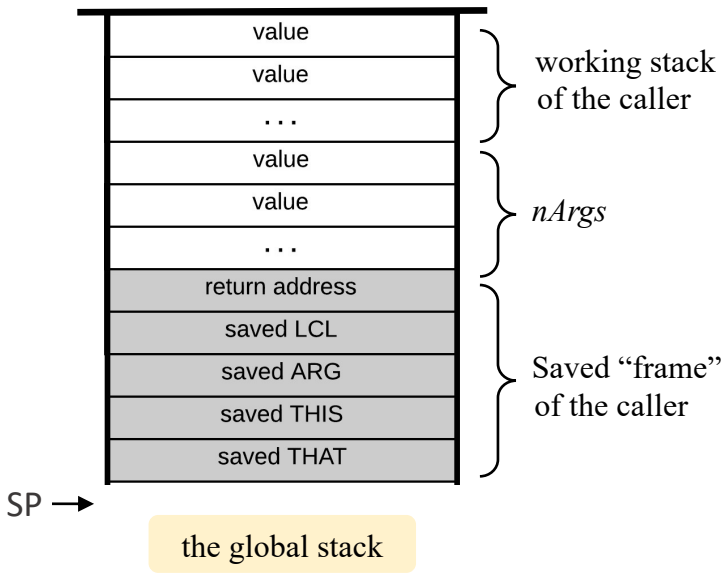
We have to:

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

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

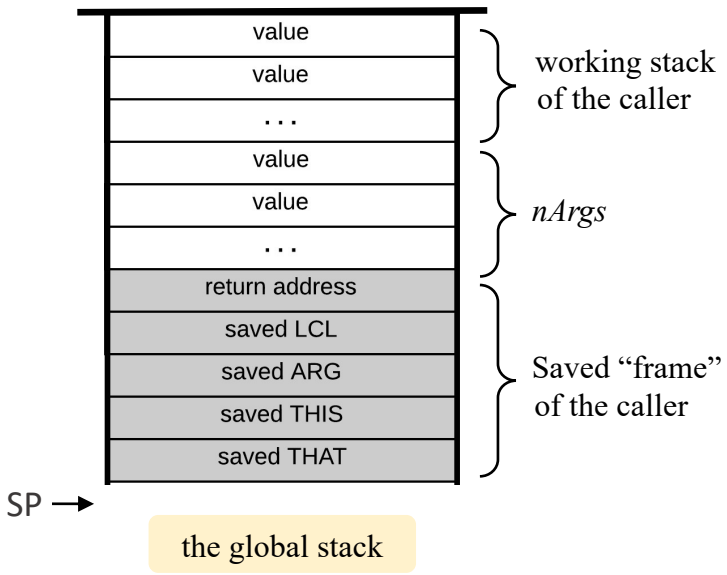
We have to:

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

Implementing `call` / function / return

The caller says:

`call` *functionName nArgs*



Handling `call` *functionName nArgs*

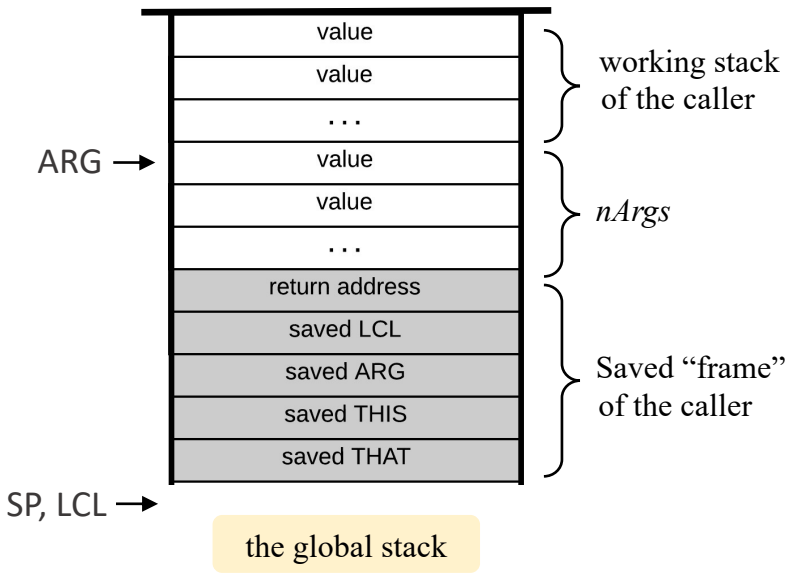
We have to:

- 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*

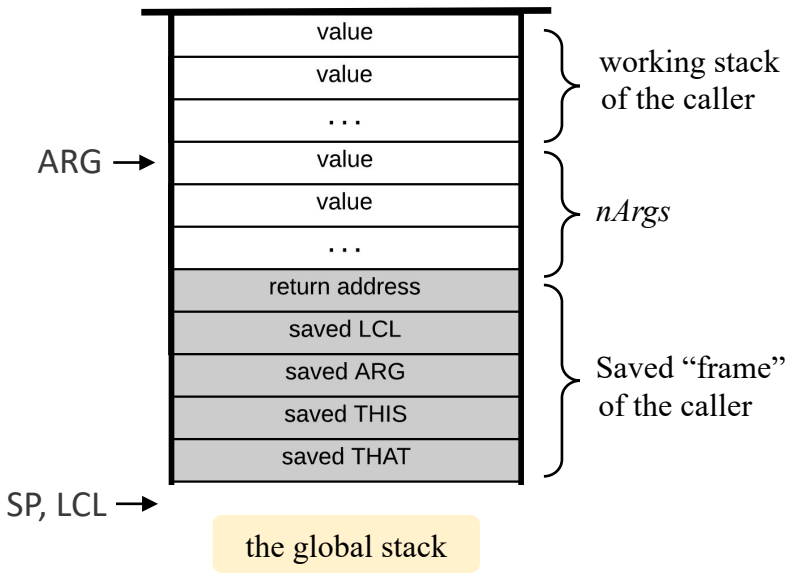
We have to:

- 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*

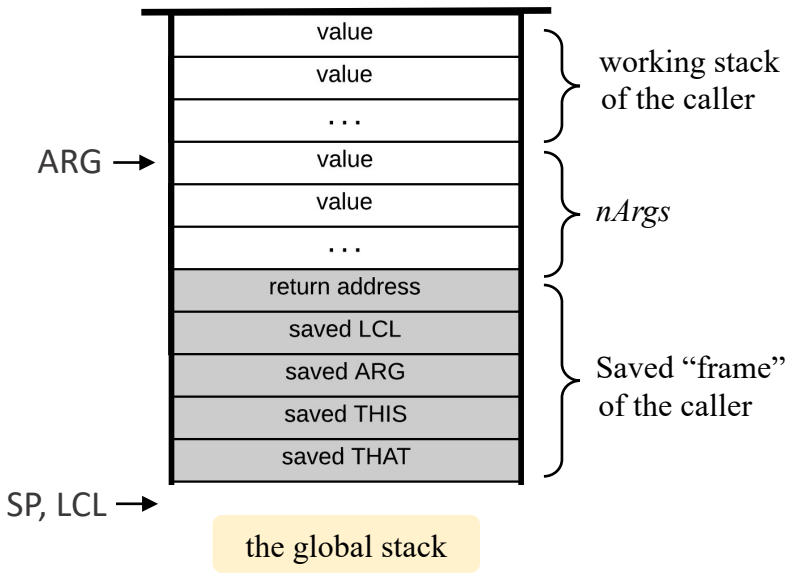
We have to:

- 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

Assembly pseudo code

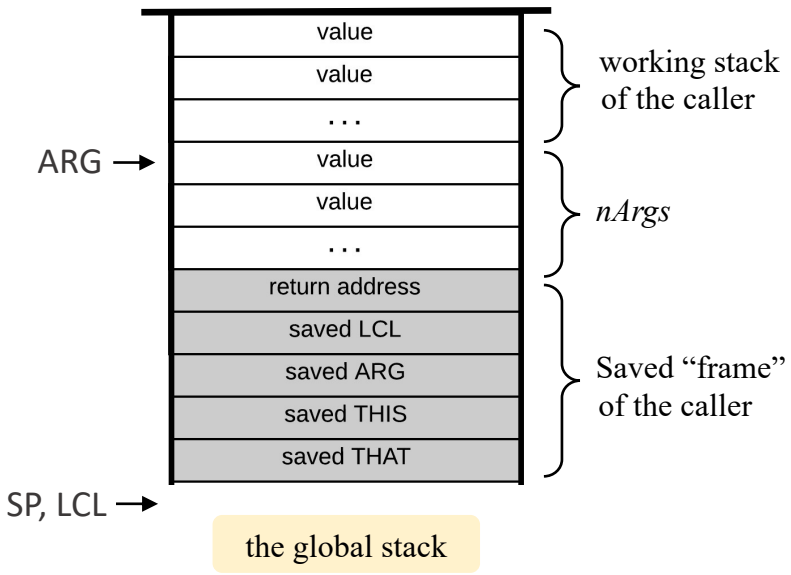
```
// call functionName nArgs
    push retAddrLabel      // Generates and pushes this label
    push LCL               // Saves the caller's LCL
    push ARG               // Saves the caller's ARG
    push THIS              // Saves the caller's THIS
    push THAT              // Saves the caller's 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
```

(Actual assembly code should be generated by the VM translator)

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

The callee is entered:

`function` *functionName nVars*



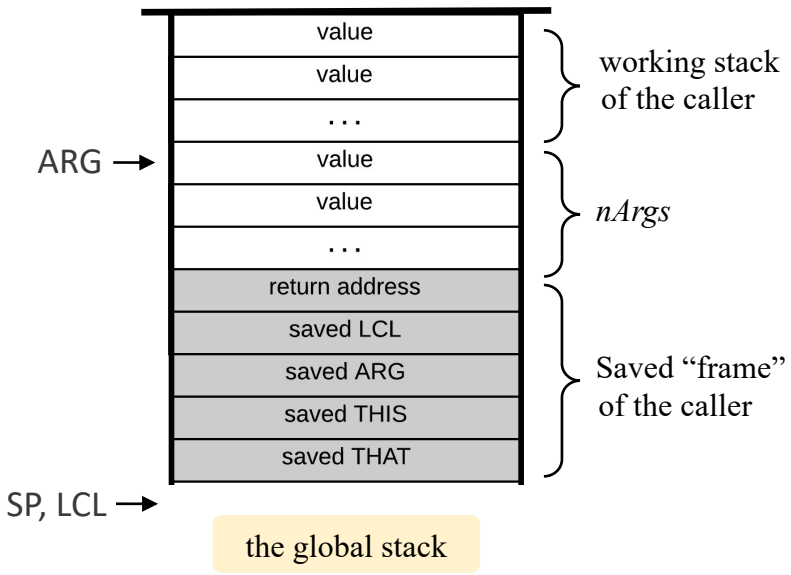
Handling `function` *functionName nVars*

We have to:

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

The callee is entered:

`function` *functionName nVars*



Handling `function` *functionName nVars*

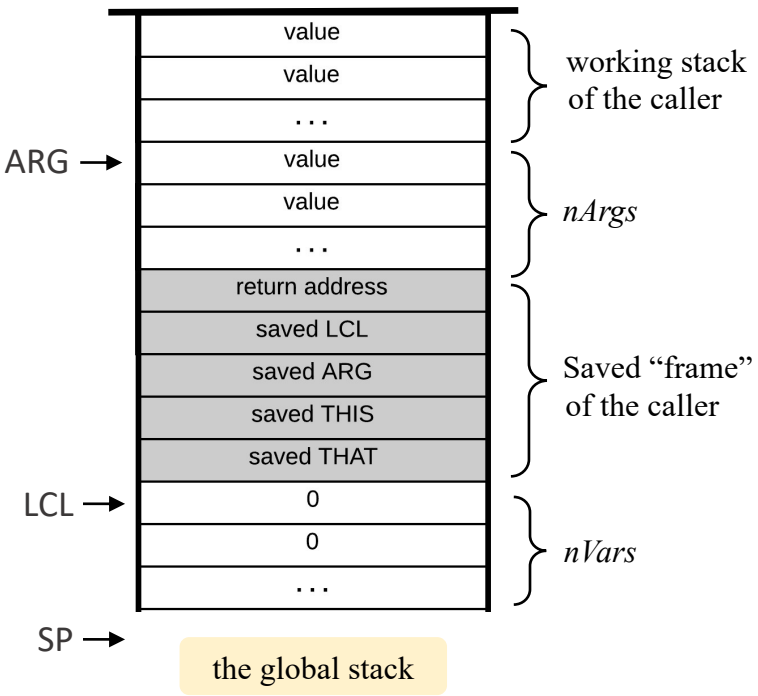
We have to:

- 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*



<u>Handling</u> `function` *functionName nVars*

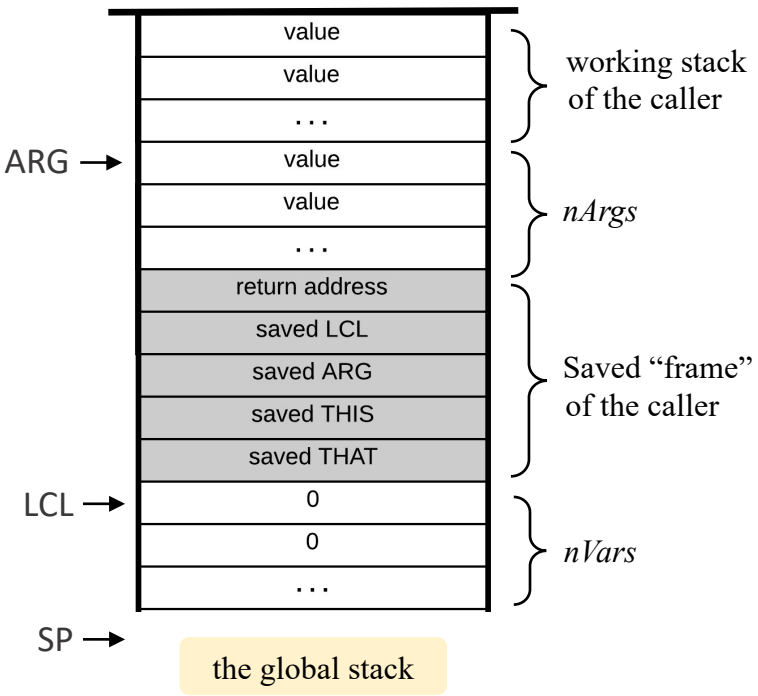
We have to:

- 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

Assembly pseudo code

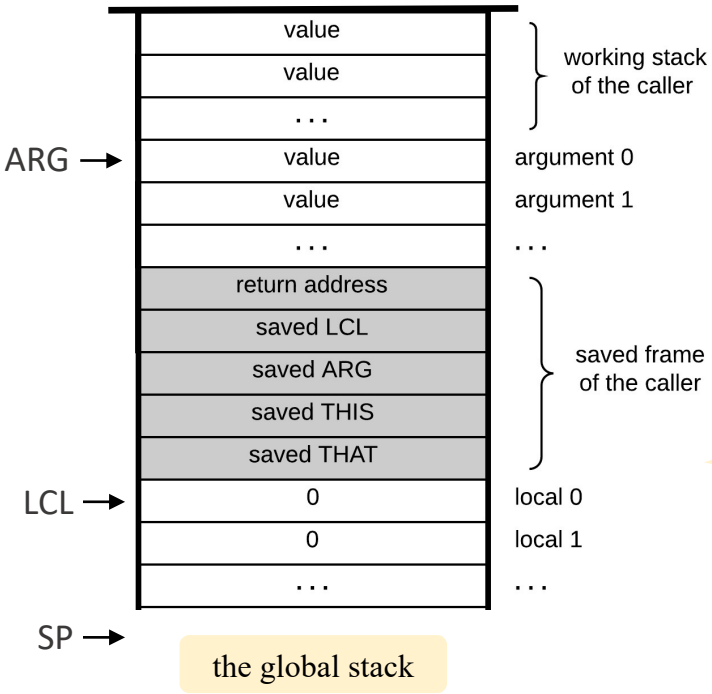
```
// function functionName nArgs
(functionName)          // function's entry point (label)
   repeat nVars times:  // initializes nVars local variables
   push 0               // to 0
```

(Actual assembly code should be generated by the VM translator)

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

The callee is entered:

`function` *functionName nVars*

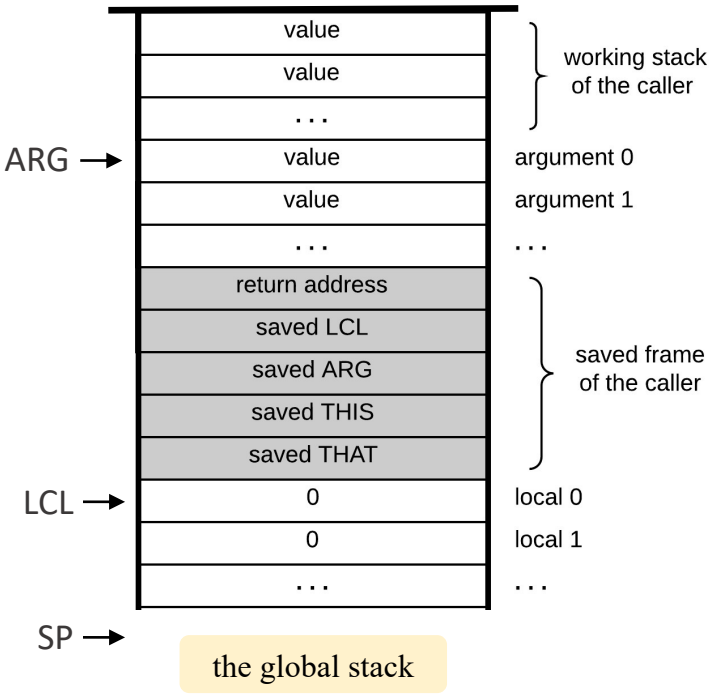


The callee is all set. It has:

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

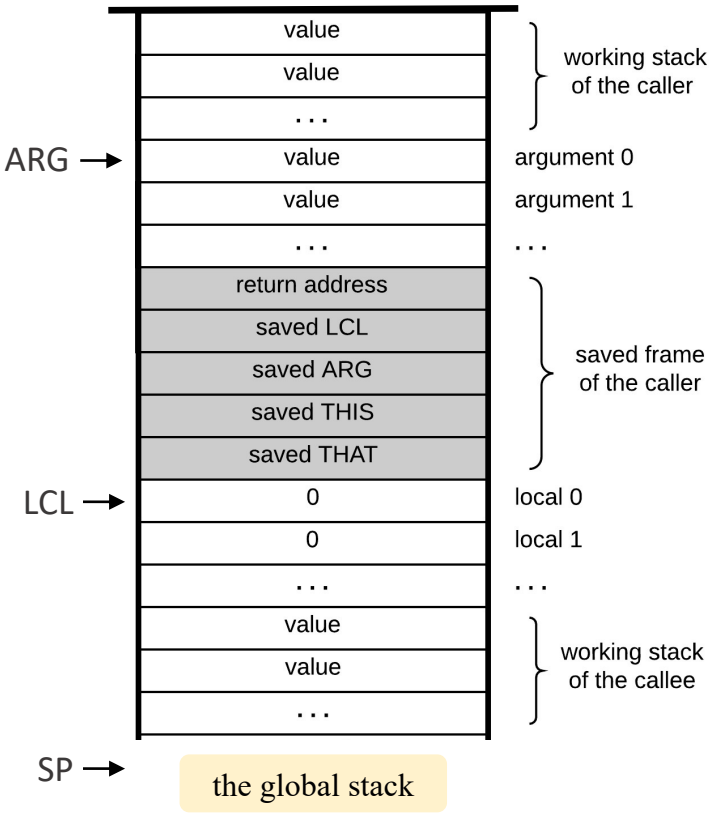
Implementing `call` / `function` / **`return`**

The callee starts running,
doing various things



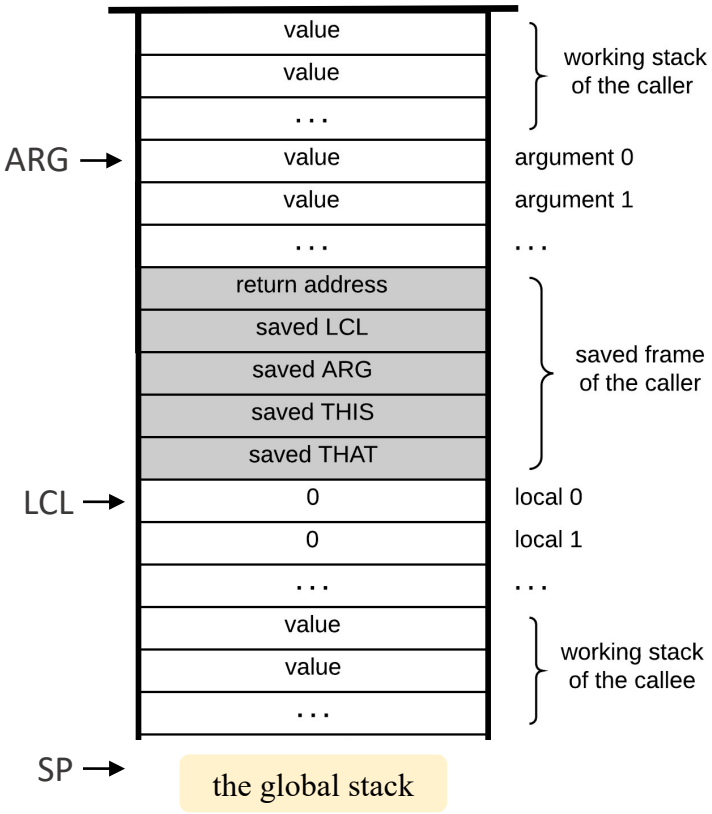
Implementing `call` / `function` / **`return`**

The callee starts running,
doing various things



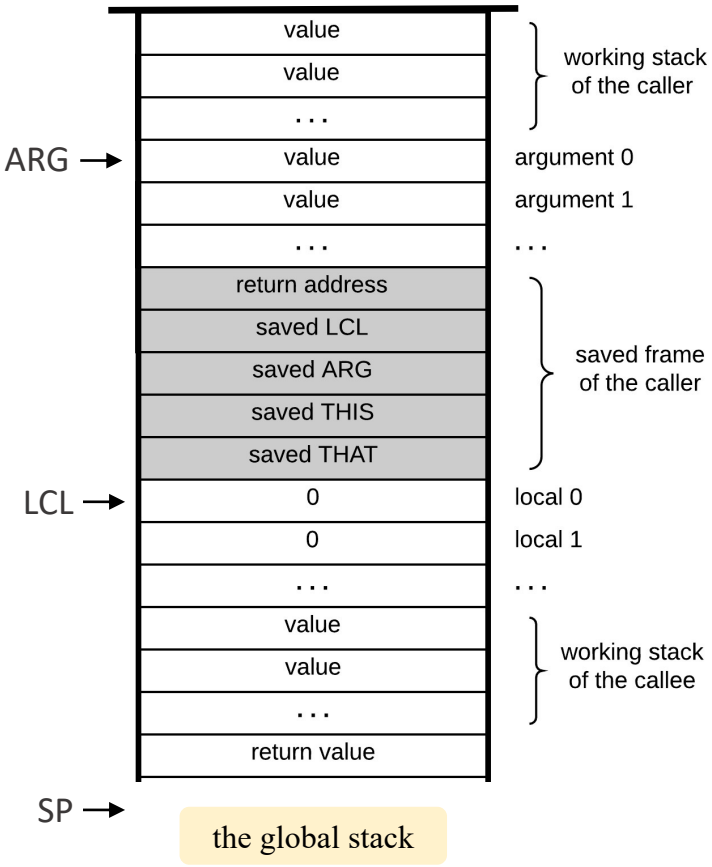
Implementing `call` / `function` / **`return`**

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



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

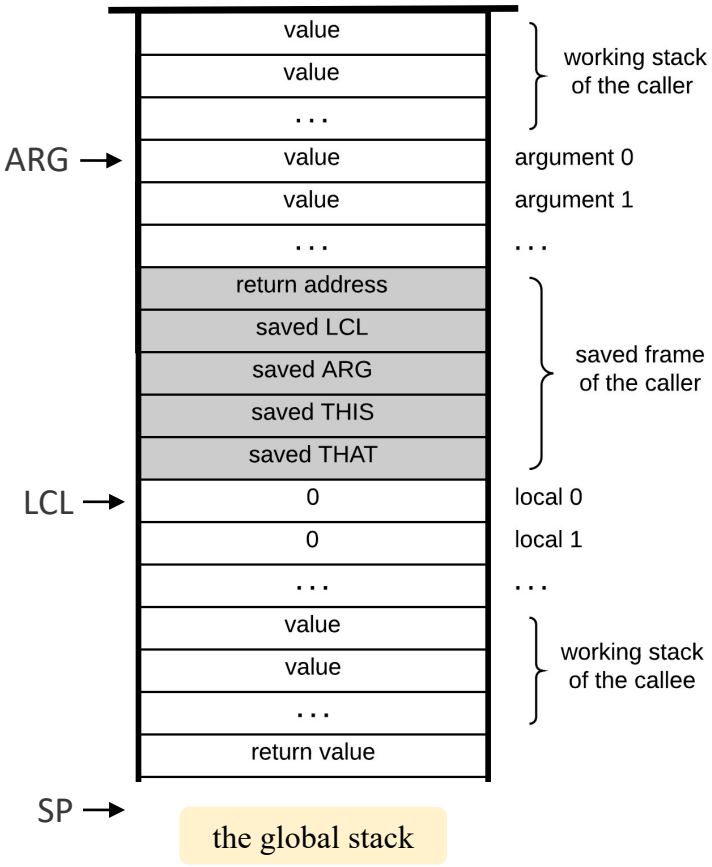
The callee prepares to return:
It pushes a *return value*



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

The callee says:

`return`



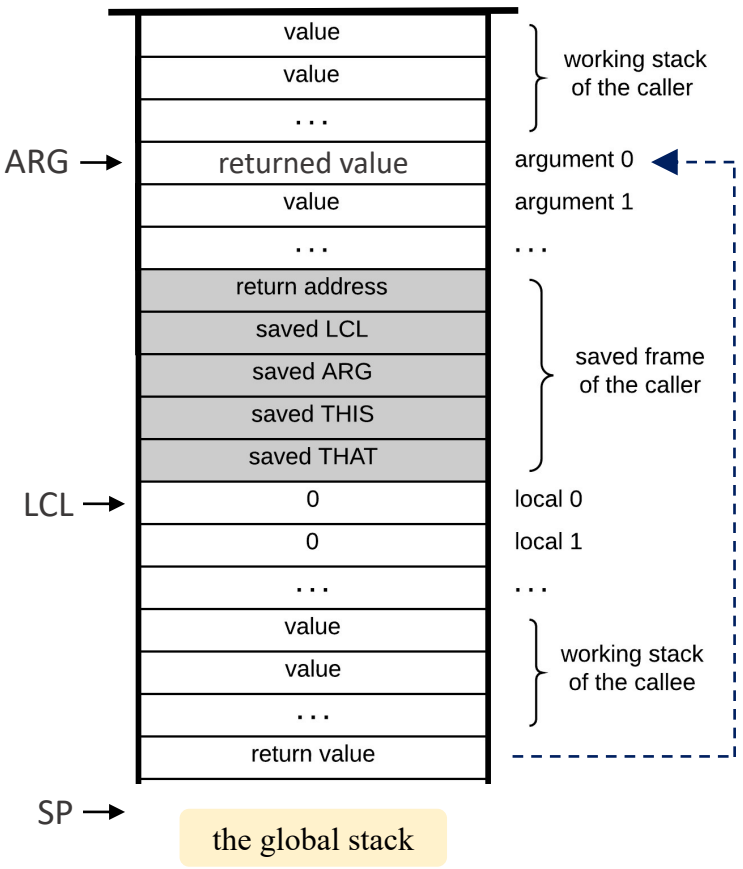
Handling `return`:

We have to:

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

The callee says:

`return`



Handling `return`:

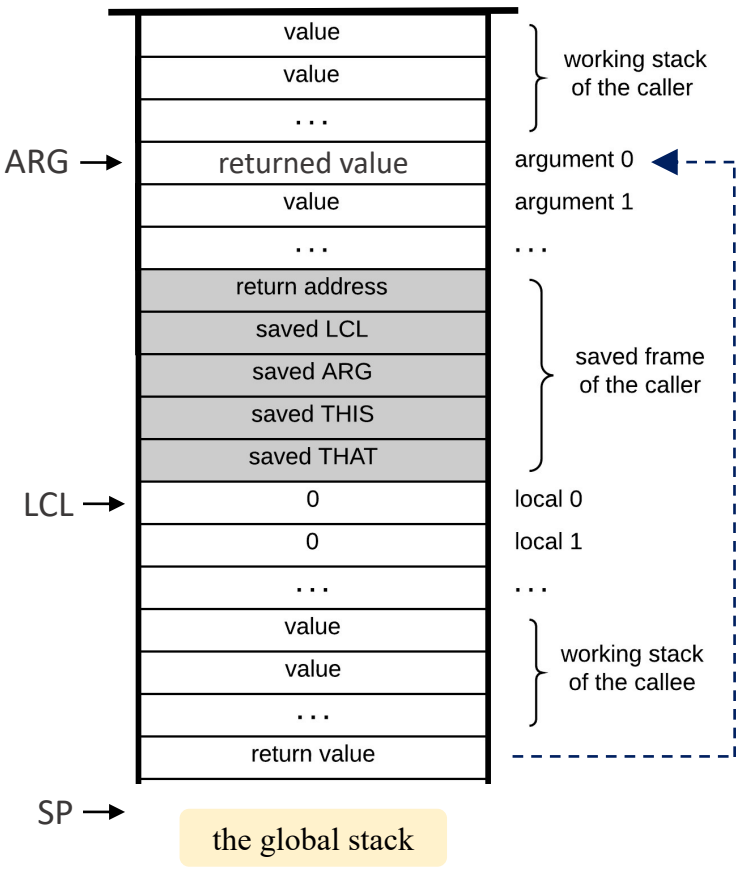
We have to:

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`:

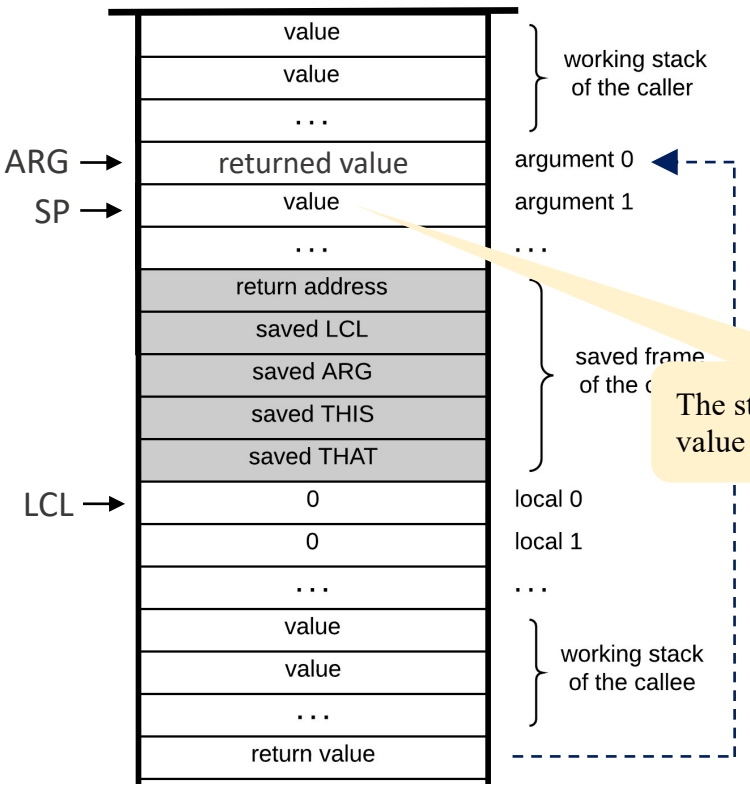
We have to:

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`



The stack space below the return value is effectively wiped out

the global stack

Handling `return`:

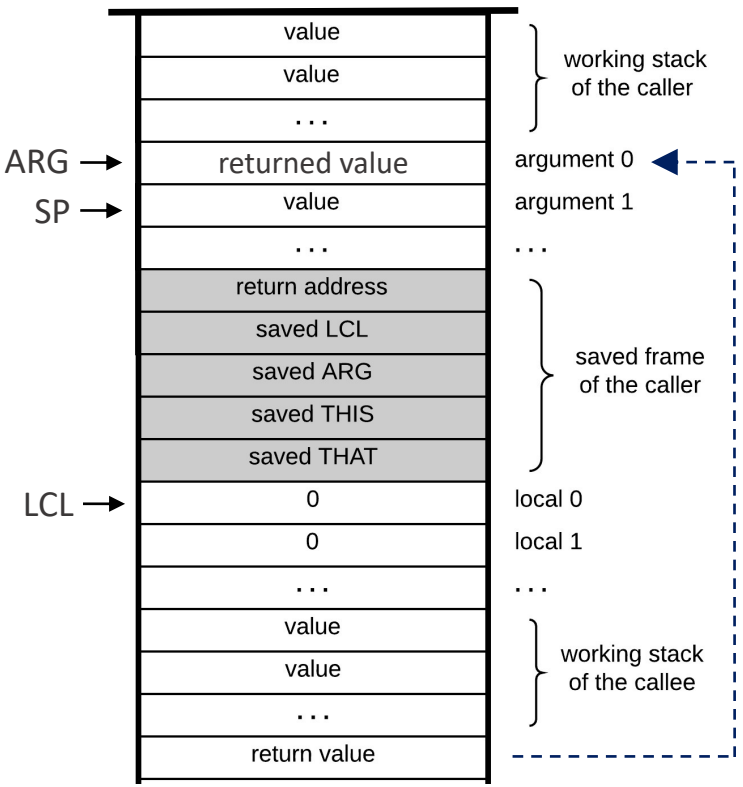
We have to:

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

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

The callee says:

`return`



the global stack

Handling `return`:

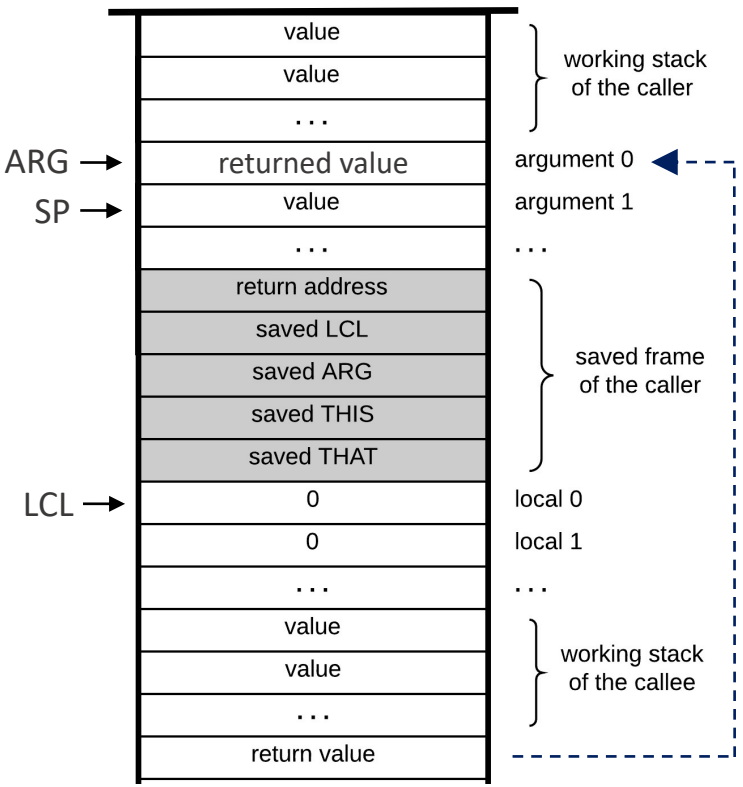
We have to:

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

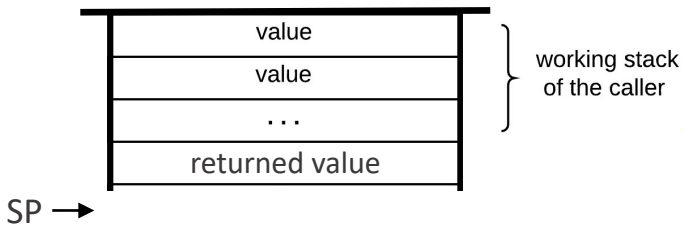
Assembly pseudo code

```
// 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
```

(Actual assembly code should be generated by the VM translator)

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

The caller resumes
its execution



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.

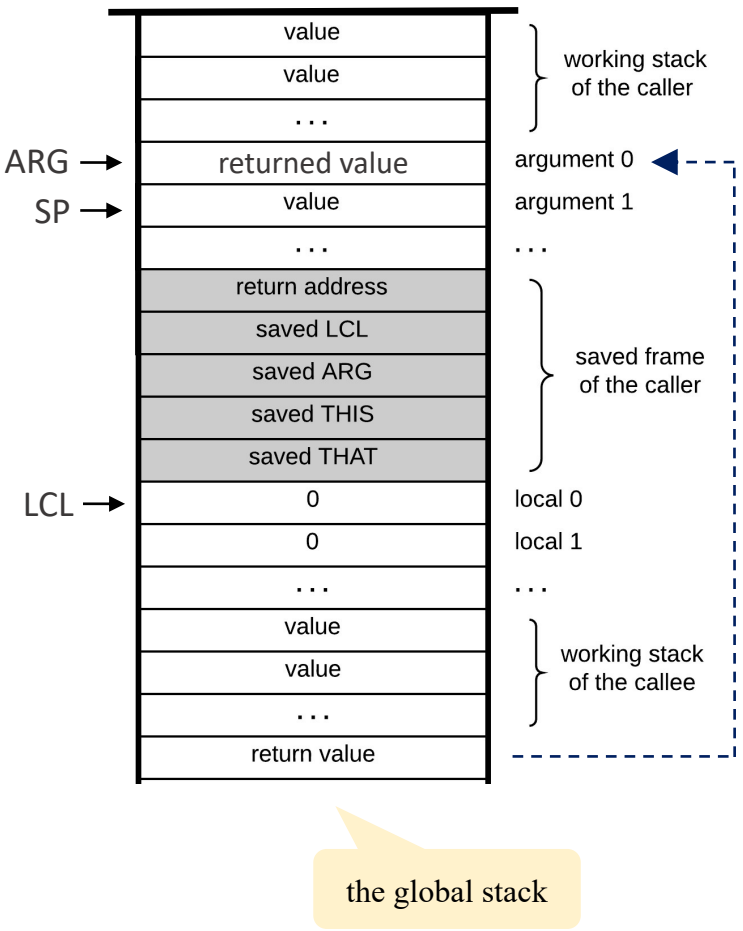
Any sufficiently advanced technology is indistinguishable from magic.

– Arthur C. Clarke, 1962

Implementing the calling chain `foo > bar > baz > ...`

The callee says:

`return`



What if the calling chain consists of many 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, and everything will work perfectly well: *Last in, first out*

Now that is the wisdom of a man: In every instance of his labor, to hitch his wagon to a star, and see his chore done by the gods themselves.

– Emerson, 1870

(change *star* with *stack*)

Lecture plan

✓ Abstraction

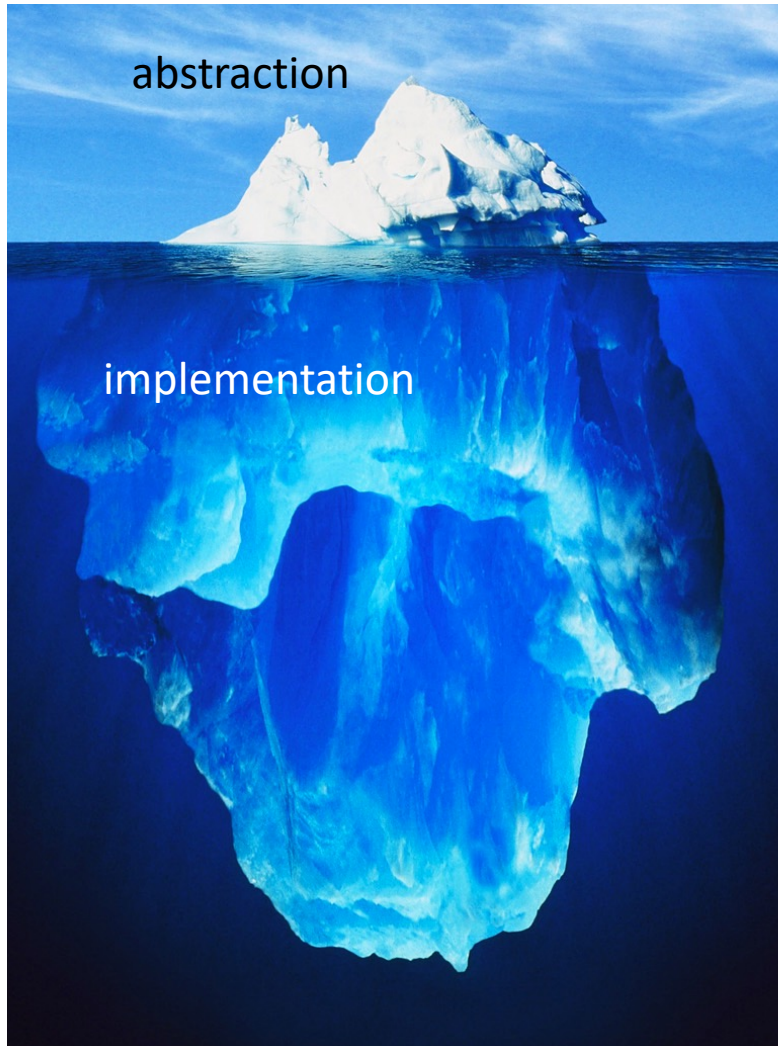
- VM branching commands
- VM function commands

✓ 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

Assembly code

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

Assembly code

```
(Foo.main)  // function's entry point label, injected by the 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

Assembly code

```
(Foo.main)  // function's entry point label, injected by the 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

Assembly code

```
(Foo.main)  // function's entry point label, injected by the 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

Assembly code

```
(Foo.main)  // function's entry point label, injected by the 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 by creating and saving
    // a return address label, saving the segment pointers of Foo.main, and then:
    goto Foo.mult  // (in assembly)
(Foo$ret.1)        // return address label, injected by the translator.
```

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

Assembly code

```
(Foo.main)  // function's entry point label, injected by the 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 by creating and saving
    // a return address label, saving the segment pointers of Foo.main, and then:
    goto Foo.mult  // (in assembly)
(Foo$ret.1)        // return address label, injected by the translator.
    // assembly code that handles push constant 7
```

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

Assembly code

```
(Foo.main)   // function's entry point label, injected by the 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 by creating and saving
    // a return address label, saving the segment pointers of Foo.main, and then:
    goto Foo.mult  // (in assembly)
(Foo$ret.1)        // return address label, injected by the translator.
    // assembly code that handles push constant 7
    ...
(Foo.mult)   // function's entry point label, injected by the 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

Assembly code

```
(Foo.main)   // function's entry point label, injected by the 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 by creating and saving
    // a return address label, saving the segment pointers of Foo.main, and then:
    goto Foo.mult  // (in assembly)
(Foo$ret.1)        // return address label, injected by the translator.
    // assembly code that handles push constant 7
    ...
(Foo.mult)   // function's entry point label, injected by the 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

Assembly code

```
(Foo.main)   // function's entry point label, injected by the 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 by creating and saving
    // a return address label, saving the segment pointers of Foo.main, and then:
    goto Foo.mult  // (in assembly)
(Foo$ret.1)        // return address label, injected by the translator.
    // assembly code that handles push constant 7
    ...
(Foo.mult)   // function's entry point label, injected by the translator.
    // assembly code that initializes the function's local variables.
    ...
    // Assembly code that handles push local 0
    // Assembly code that handles return by getting the return address
    // (which happens to be Foo$ret.1), copying the return value, reinstating
    // the segments of Foo.main, and then:
    goto Foo$ret.1     // (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

Assembly code

```
(Foo.main)  // function's entry point label, injected by the 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 by creating and saving
   // a return address label, saving the segment pointers of Foo.main, and then:
   goto Foo.mult  // (in assembly)
(Foo$ret.1)        // return address label, injected by the translator.
   // assembly code that handles push constant 7
   ...
(Foo.mult)  // function's entry point label, injected by the translator.
   // assembly code that initializes the function's local variables.
   ...
   // Assembly code that handles push local 0
   // Assembly code that handles return by getting the return address
   // (which happens to be Foo$ret.1), copying the return value, reinstating
   // the segments of Foo.main, and then:
   goto Foo$ret.1    // (in assembly)
```

<u>The translation created a run-time system:</u>

When executed, the translated code realizes the program semantics, as well as the function call-and-return overhead, passing arguments and return values, saving and reinstating function states, etc.

Lecture plan

✓ Abstraction

- VM branching commands
- VM function commands

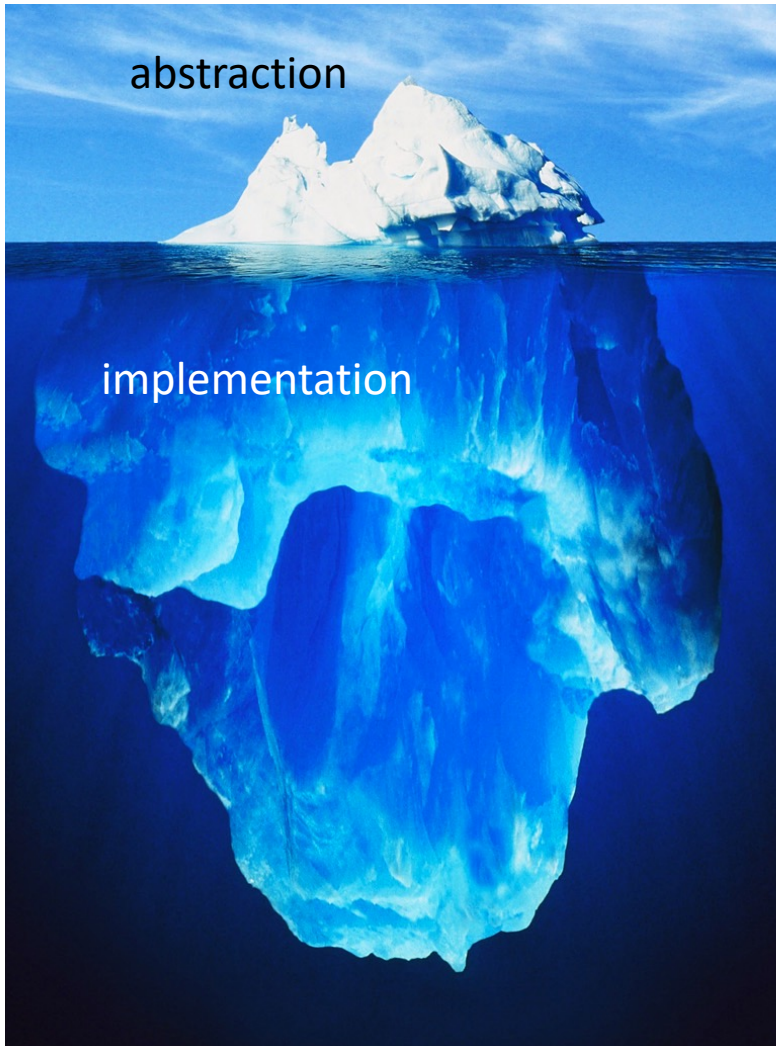
✓ Implementation (conceptual)

- VM branching commands
- VM function commands

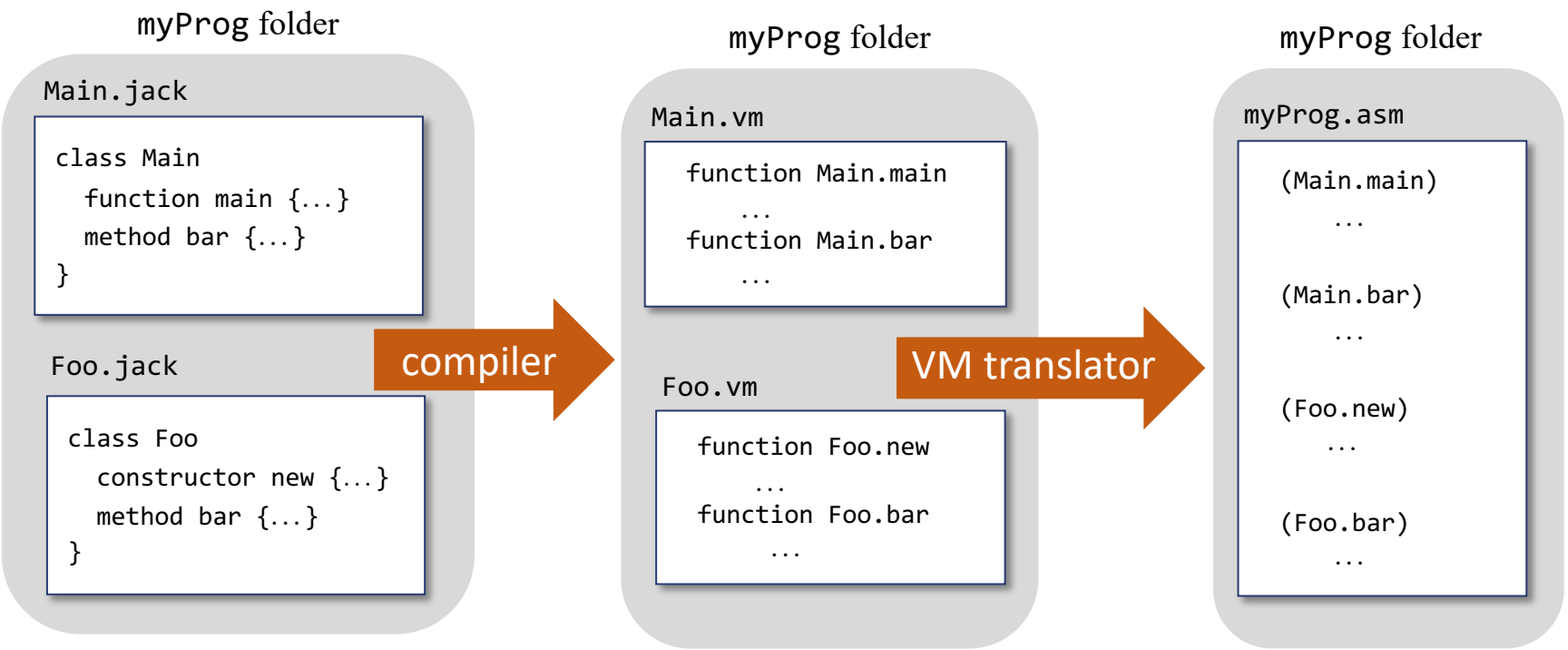
Implementation (nuts and bolts)

➡ Run-time system

- Standard mapping
- VM translator
- Project 8



The big picture: Compilation



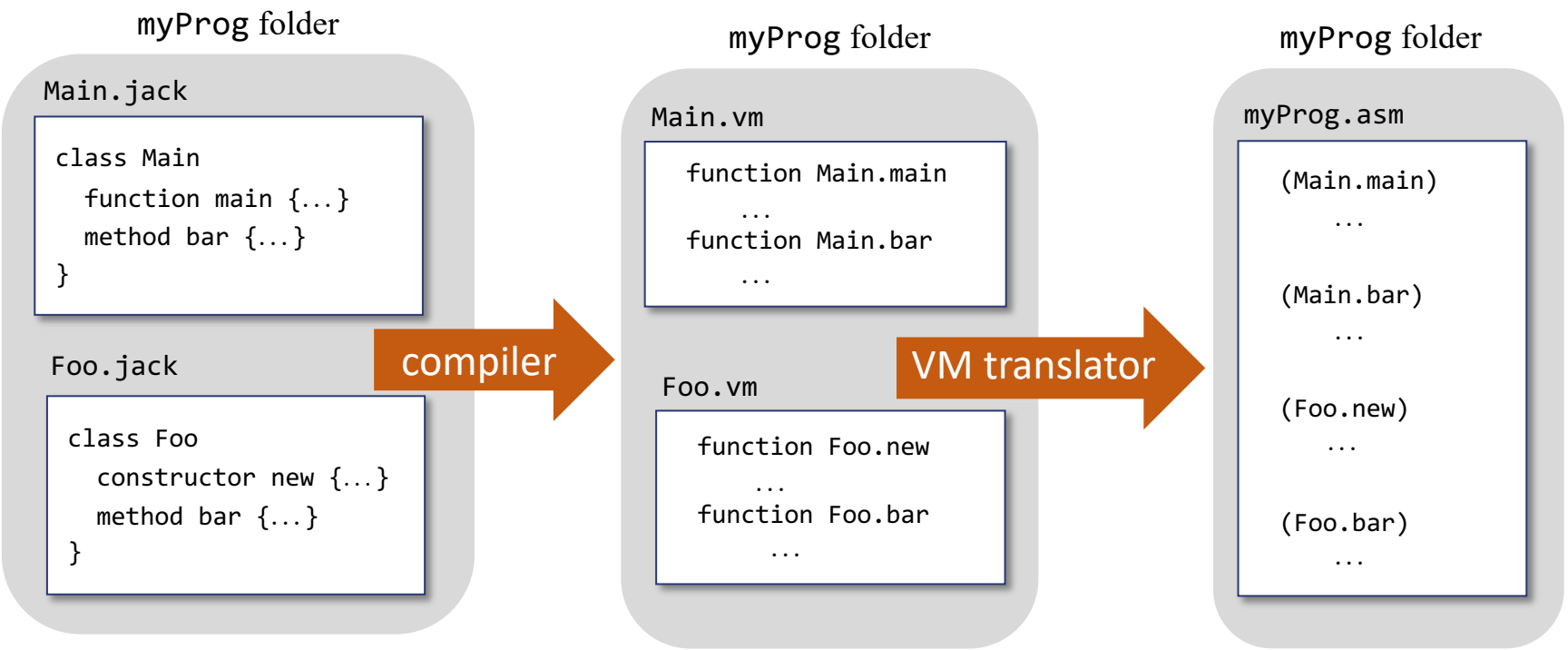
A *Jack class* is a collection of *constructors*, *methods*, and *functions* (static methods)

(We'll introduce the Jack language in the next lecture)

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

All the VM functions are translated into a single assembly file

The big picture: Compilation



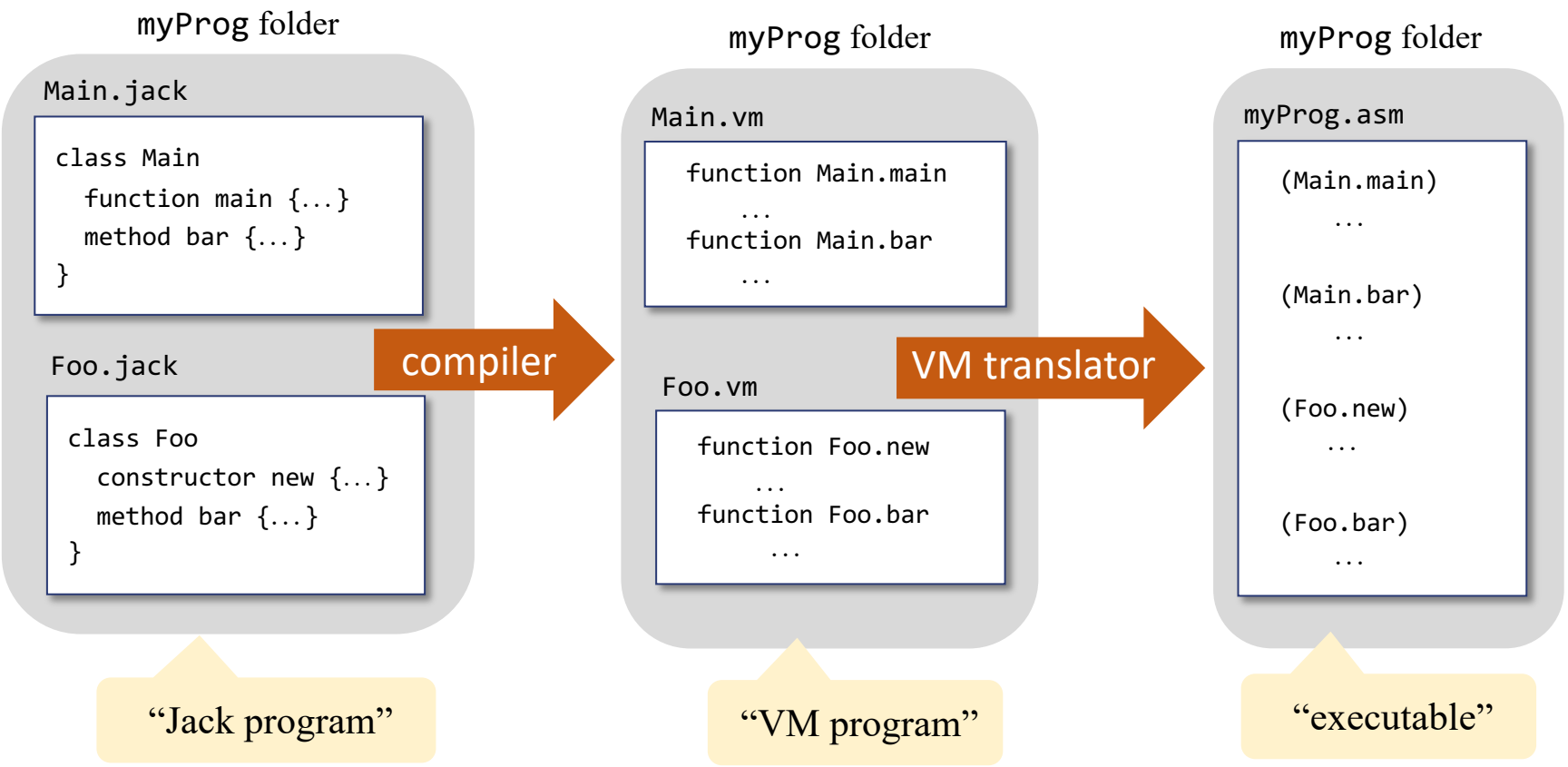
developed in projects 10 - 11

<u>Compiler usage:</u> `JackCompiler` *folderName* (creates `.vm` files, one for each `.jack` file)

<u>VM translator usage:</u> `VMTranslator` *folderName* (creates a single `.asm` file)

developed in projects 7 - 8

The big picture: Compilation



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

Conventions

- One function in the VM program is expected to be named `Main.main`
- In addition, there should be a VM function named `sys.init`

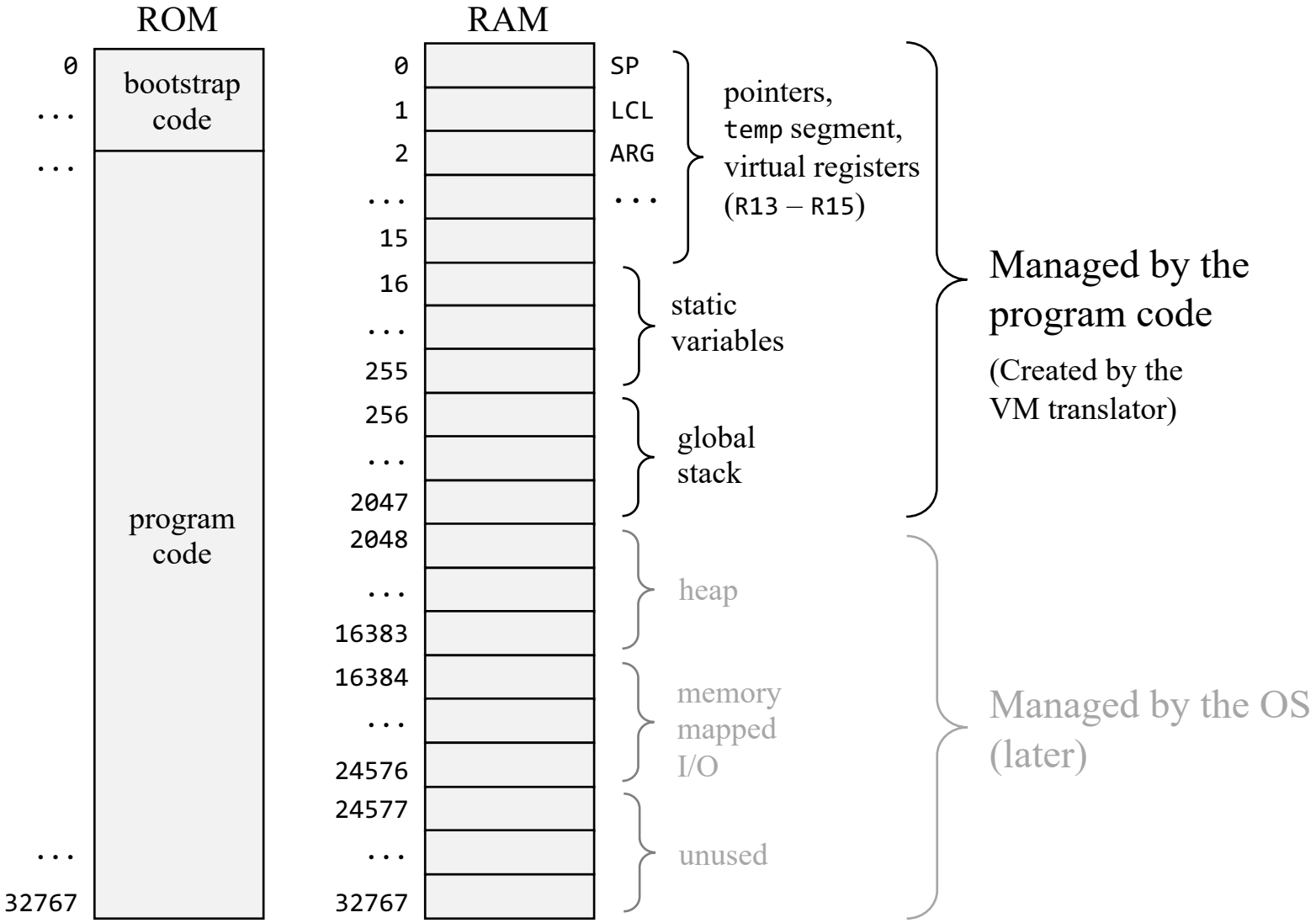
  `Sys.init` should call `Main.main`, and enter an infinite loop
- When the translated VM code starts running, or is reset, it should start executing `Sys.init`
- The stack is stored in the host RAM, starting at address 256

Implementation (on the Hack platform)

```
// Bootstrap code (written in machine language)
SP = 256
call Sys.init   // (no arguments)
```

Should be put 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: Special 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.

Lecture plan

✓ Abstraction

- VM branching commands
- VM function commands

✓ Implementation (conceptual)

- VM branching commands
- VM function commands

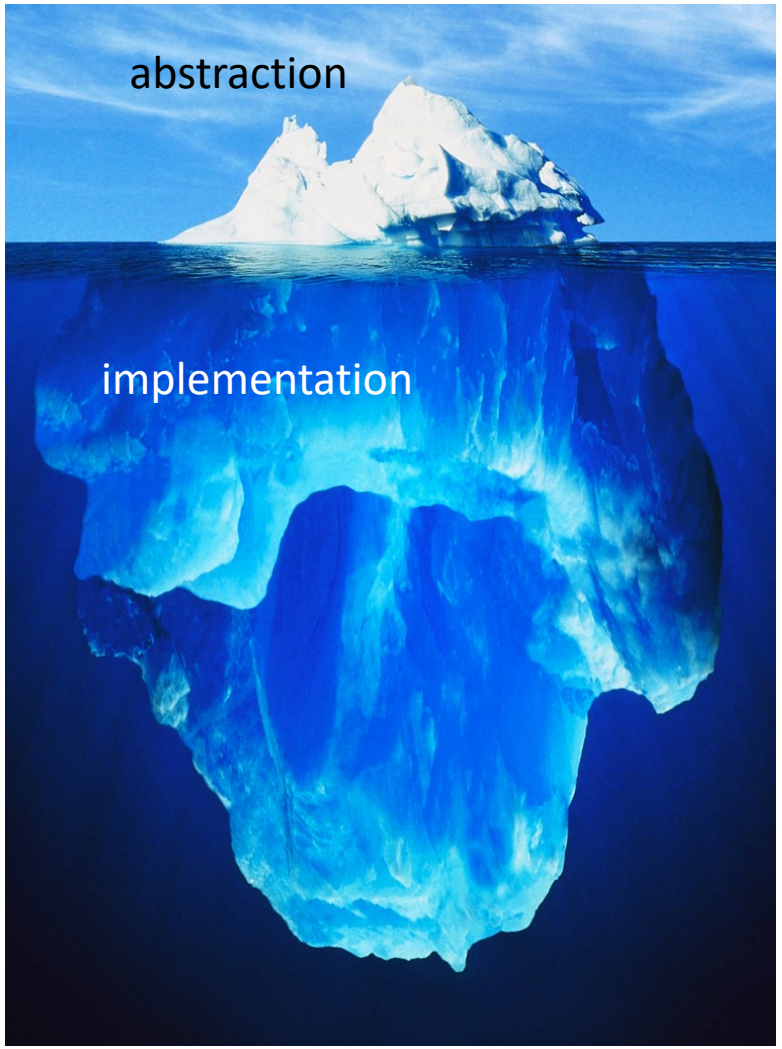
Implementation (nuts and bolts)

✓ Run-time system

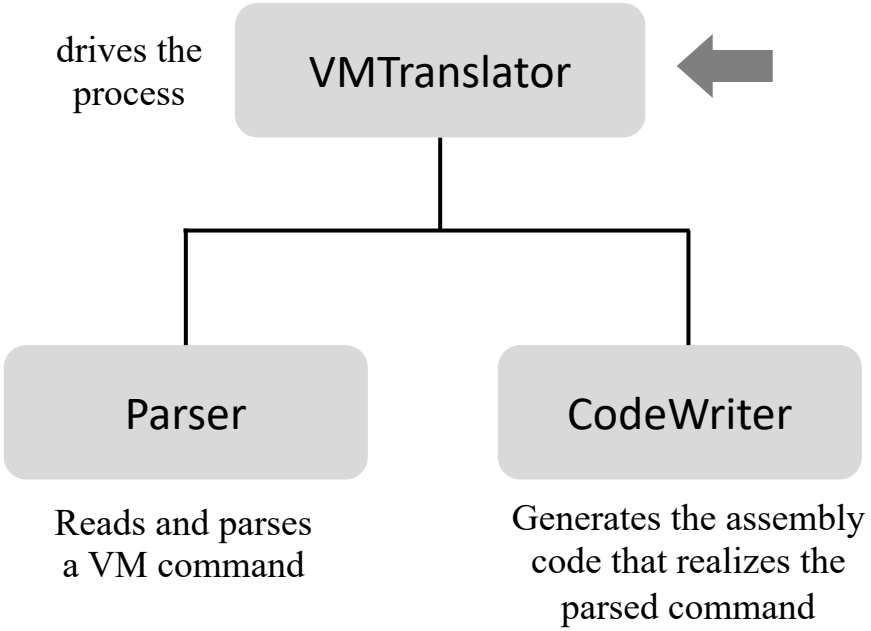
✓ Standard mapping

➡ VM translator

- Project 8



The VM translator: Implementation



<u>VM translator:</u>

- An extension of the basic VM translator written in project 7
- Adds the implementation of the *branching* and *function* commands

VMTranslator

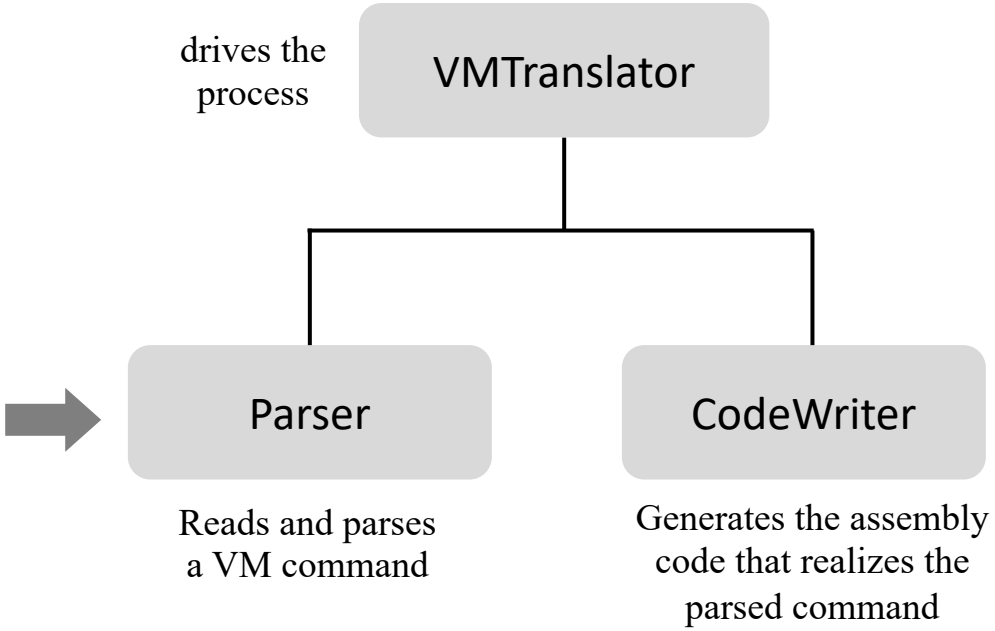
Input: *fileName*`.vm`: name of a single source file, or
folderName: name of a folder containing one or more `.vm` source files

Output: *fileName*`.asm` or
folderName`.asm`

<u>Process:</u>

- 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.

The VM translator: Implementation



<u>VM translator:</u>

- An extension of the basic VM translator written in project 7
- Adds the implementation of the *branching* and *function* commands

Parser

<u>The same `Parser` developed in project 7:</u>

- 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

The same `Parser` developed in project 7:

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`.

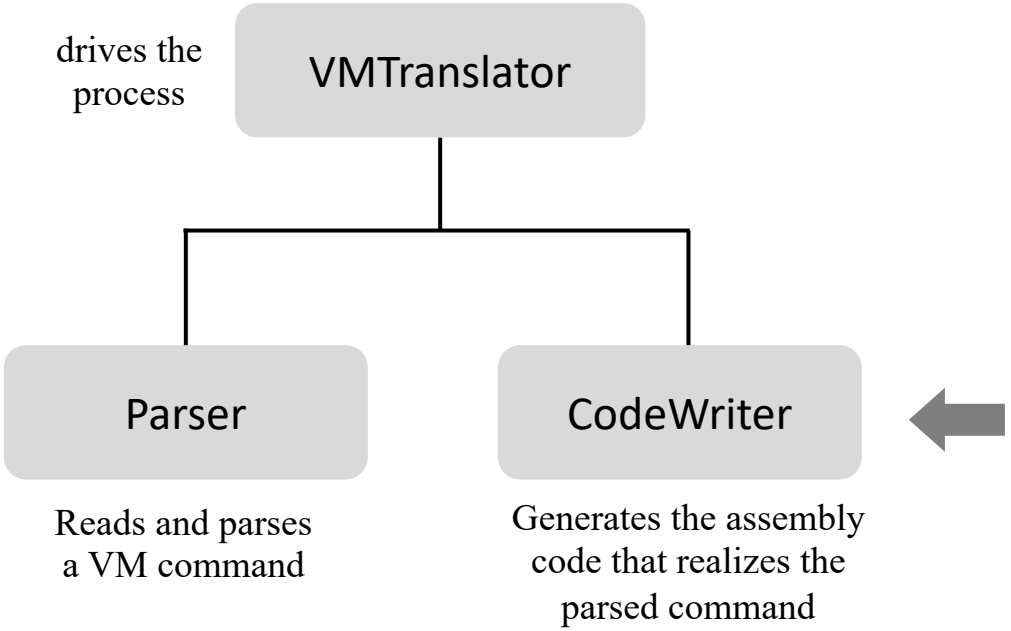
Implementation note:

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.

The VM translator: Implementation



<u>VM translator:</u>

- An extension of the basic VM translator written in project 7
- Adds 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.

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: "Standard Mapping of the VM on the Hack Platform".

Lecture plan

✓ Abstraction

- VM branching commands
- VM function commands

✓ Implementation

- VM branching commands
- VM function commands

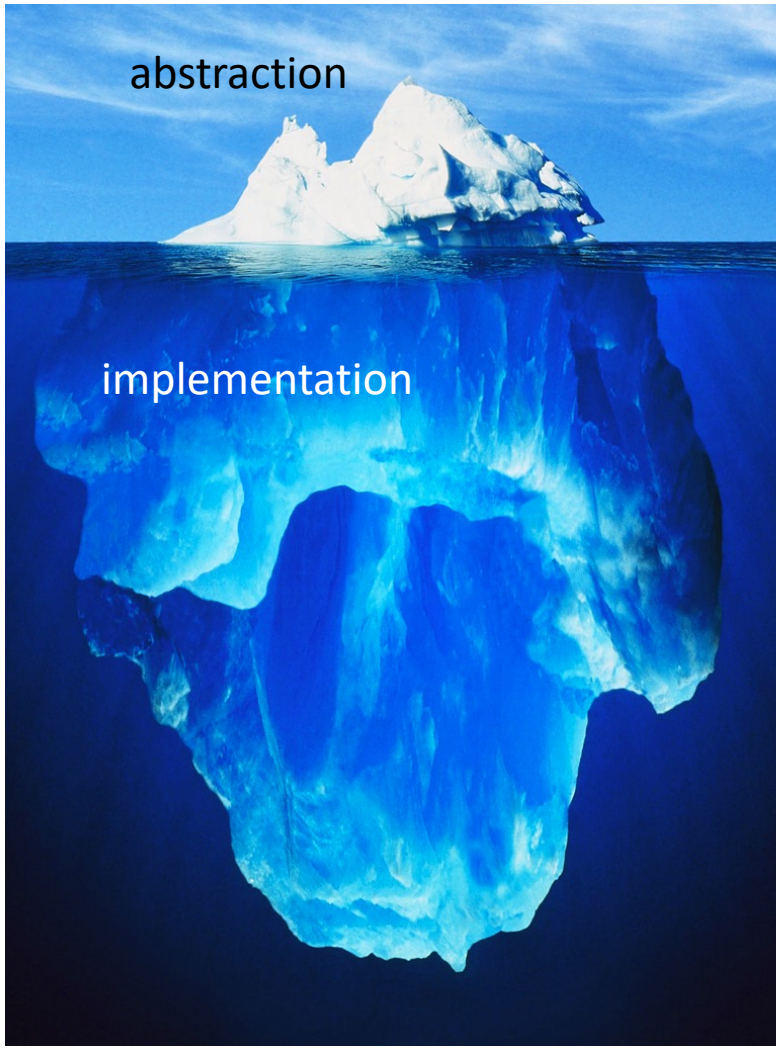
The Virtual Machine

✓ Program translation

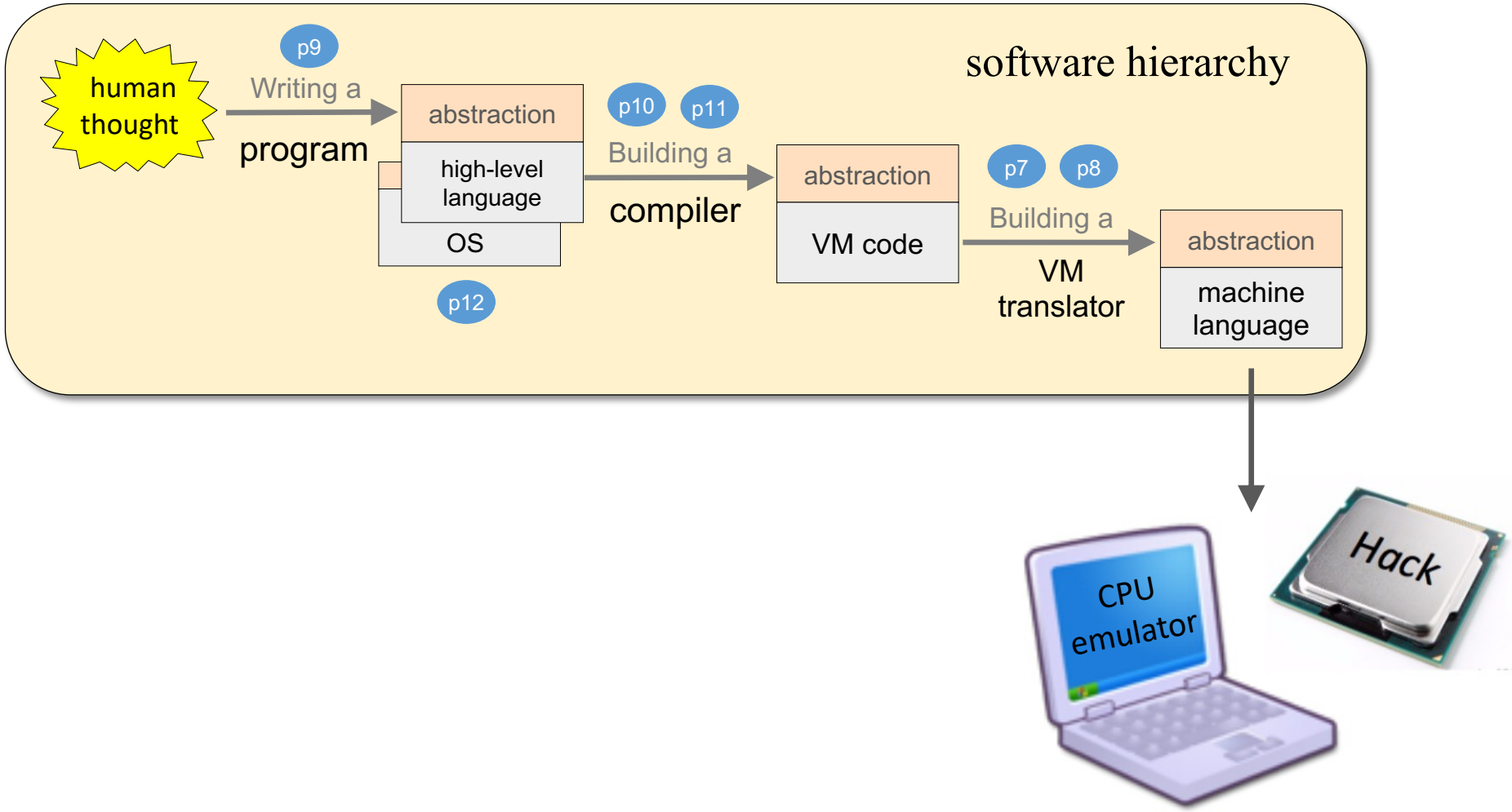
✓ Standard mapping

✓ VM translator

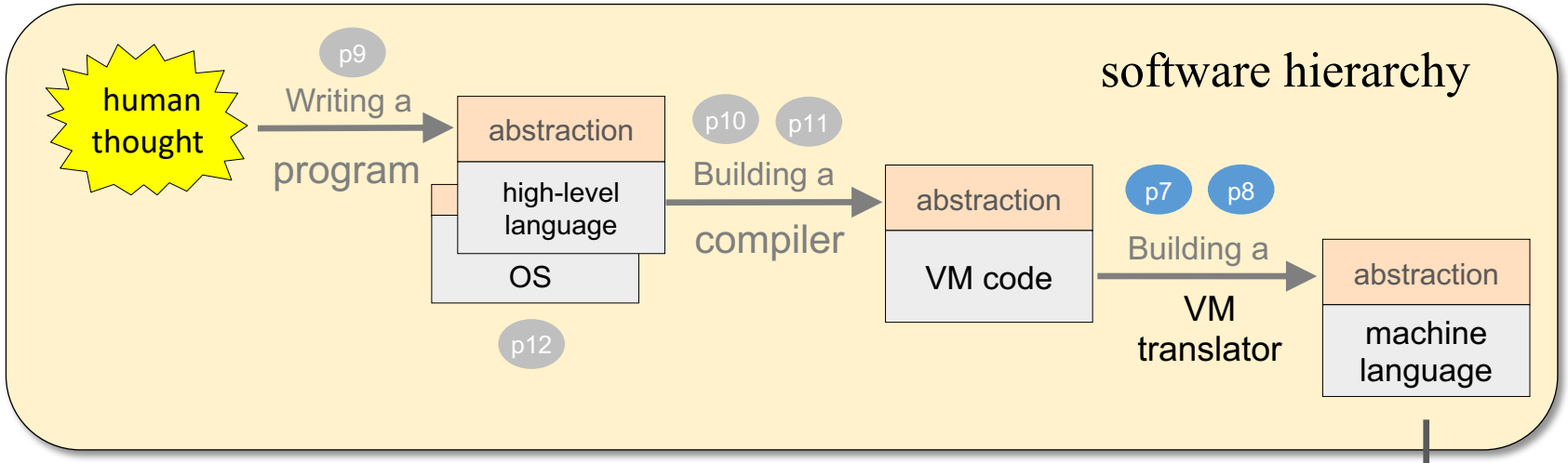
➡ Project 8

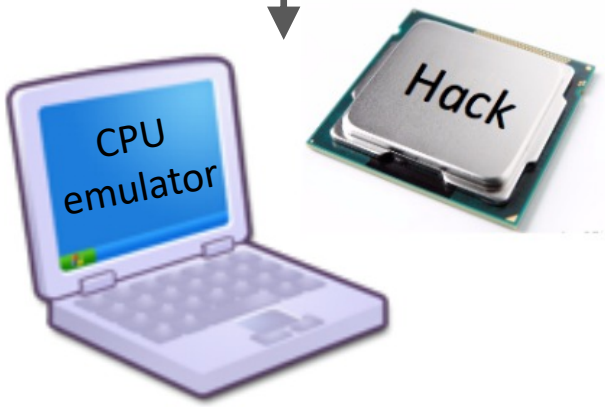


The Big Picture



The Big Picture





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

<u>To test the translator</u>: Run the generated code on the target platform.

Test programs

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Test programs

```
ProgramFlow:
  BasicLoop
      BasicLoop.vm
      BasicLoopVME.tst
      BasicLoop.tst
      BasicLoop.cmp
  FibonacciSeries

FunctionCalls:
  SimpleFunction
  NestedCall
  FibonacciElement
  StaticsTest
```

Testing routine (for every `Xxx` test)

0. (optional) 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`; The result: A file named `Xxx.asm`
2. Load and run `Xxx.tst` on the CPU emulator; This script loads `Xxx.asm` into the CPU emulator, executes it, and compares the output to `Xxx.cmp`

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

Test programs: `BasicLoop`

```
ProgramFlow:
  BasicLoop
      BasicLoop.vm
      BasicLoopVME.tst
      BasicLoop.tst
      BasicLoop.cmp
  FibonacciSeries

FunctionCalls:
  SimpleFunction
  NestedCall
  FibonacciElement
  StaticsTest
```

Tests the handling of the VM commands:

- `label`
- `if-goto`

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
```

Test programs

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Test programs: `FibonacciSeries`

ProgramFlow:

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

A more elaborate test of handling the VM commands:

- `Label`
- `goto`
- `if-goto`

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
```

Test programs

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Test programs: `SimpleFunction`

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

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

Tests the handling of the VM commands:

- `function`
- `return`

Simple test, which involves no caller

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
```

Test programs

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Test programs: `FibonacciElement`

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

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

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

Should yield a single output file: `FibonacciElement.asm`

Test programs: `FibonacciElement`

ProgramFlow:
- BasicLoop
- FibonacciSeries

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

VM version of the recursive logic:

```
function fib(n):
  if n < 2
     return n
  else
     return fib(n-1) + fib(n-2)
```

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.fibona
   push argument 0
   push constant 1
   sub
   call Main.fibona
   add
   return
```

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

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

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).
function Sys.init 0
   push constant 4
   call Main.fibonacci 1
label WHILE
   goto WHILE
```

Test programs

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

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

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Test programs: `StaticsTest`

`ProgramFlow:`

- `BasicLoop`
- `FibonacciSeries`

`FunctionCalls:`

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

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

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

Should yield 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.tst
  - StaticsTest.cmp

class1.vm

```
// Stores two supplied arguments in static 0 and static 1
function Class1.set 0
   push argument 0
   pop static 0
   push argument 1
   pop static 1
   push constant 0
   return

// Returns (static 0) – (static 1)
function Class1.get 0
   push static 0
   push static 1
   sub
   return
```

Test programs: `StaticsTest`

ProgramFlow:
- BasicLoop
- FibonacciSeries

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

class1.vm

```
// Stores two supplied arguments in static 0 and static 1
function Class1.set 0
   push argument 0
   pop static 0
   push ar
   pop sta
   push co
   return

// Returns
function
   push
   push
   sub
   retu
```

class2.vm

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

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
```

- The two class files have the same logic
- But, different `.vm` files should have different `static` segments

Project 8: Tools and resources

- Source, test, compare files: `nand2tetris/projects/08`
- Reference: chapter 8 in *The Elements of Computing Systems*
- Experimenting with the source VM programs: the supplied *VM emulator*
- Translating the source VM programs into assembly: your *VM translator*
- Testing the resulting assembly code: the supplied *CPU emulator*
- Programming language for implementing your VM translator: Java, Python, ...
- Tutorials: VM emulator, CPU emulator (`nand2tetris` web site)

Lecture plan

✓ Abstraction

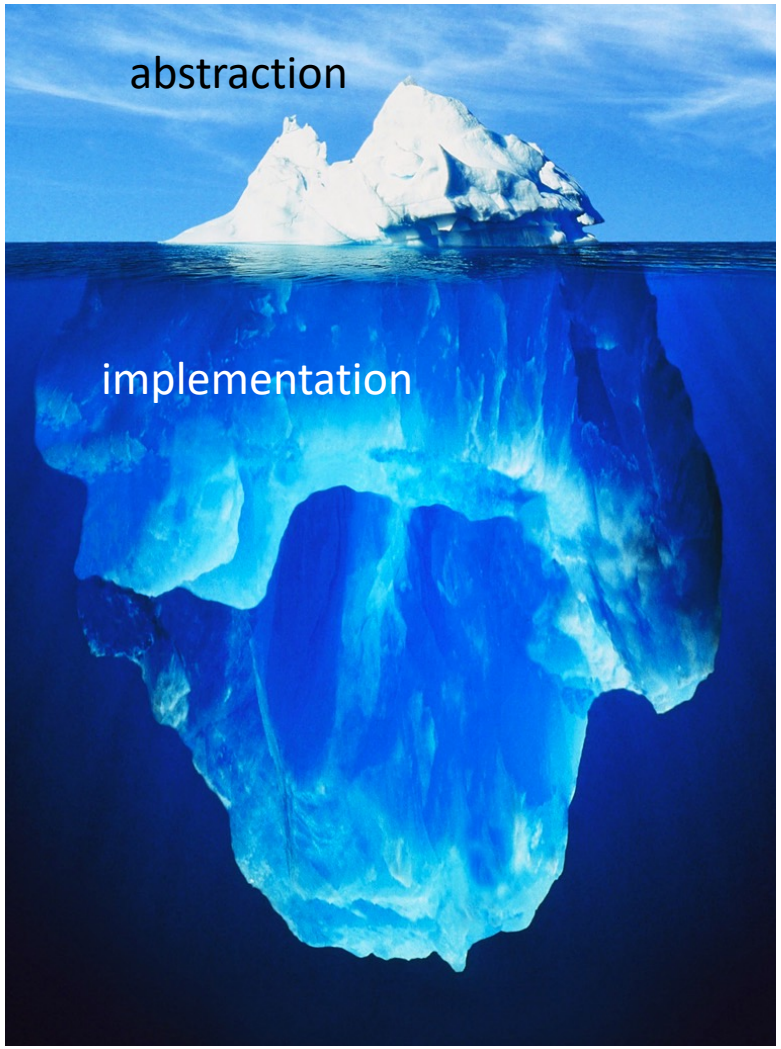
- VM branching commands
- VM function commands

✓ Implementation

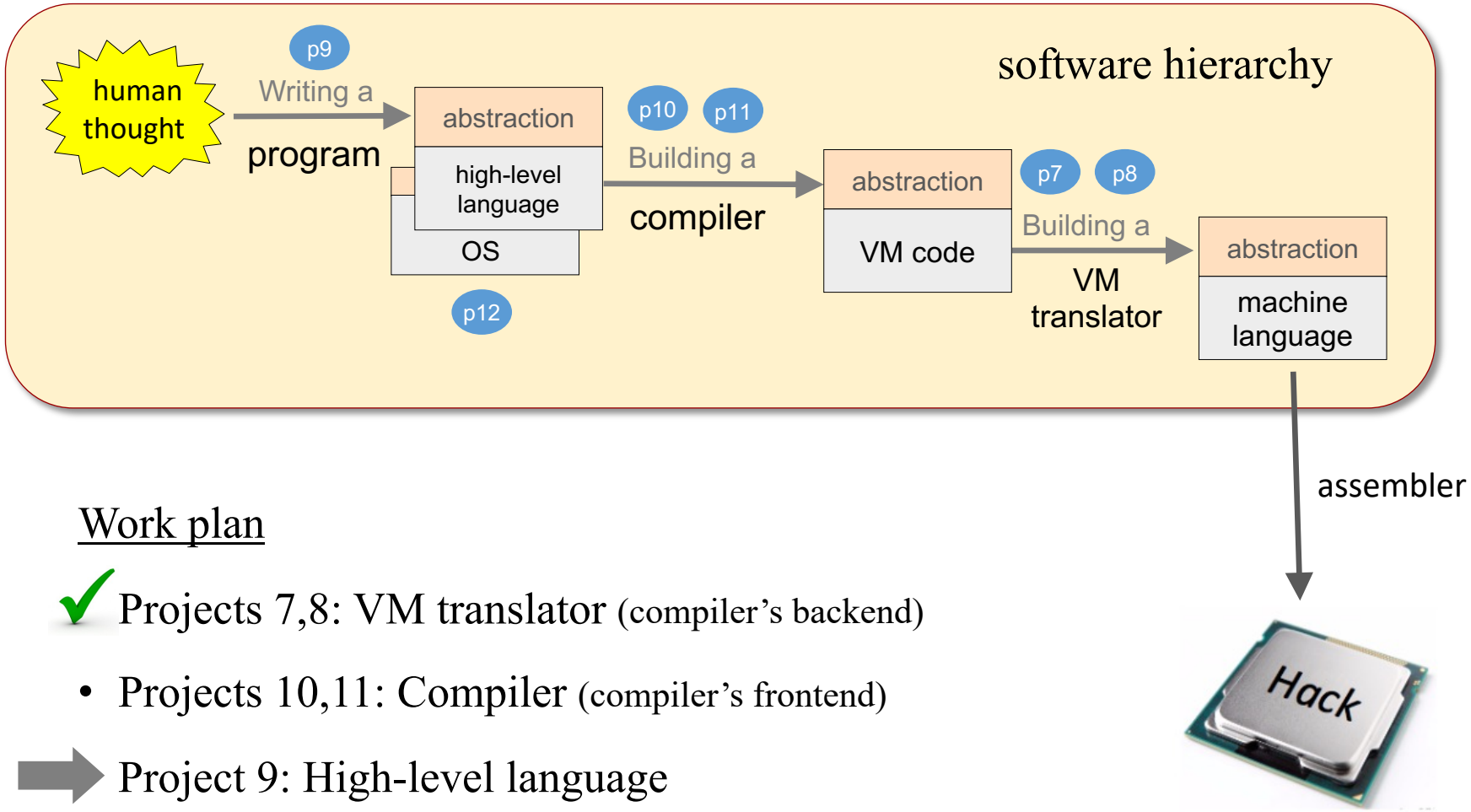
- VM branching commands
- VM function commands

✓ The Virtual Machine

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



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.

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