



University of
Nottingham

UK | CHINA | MALAYSIA

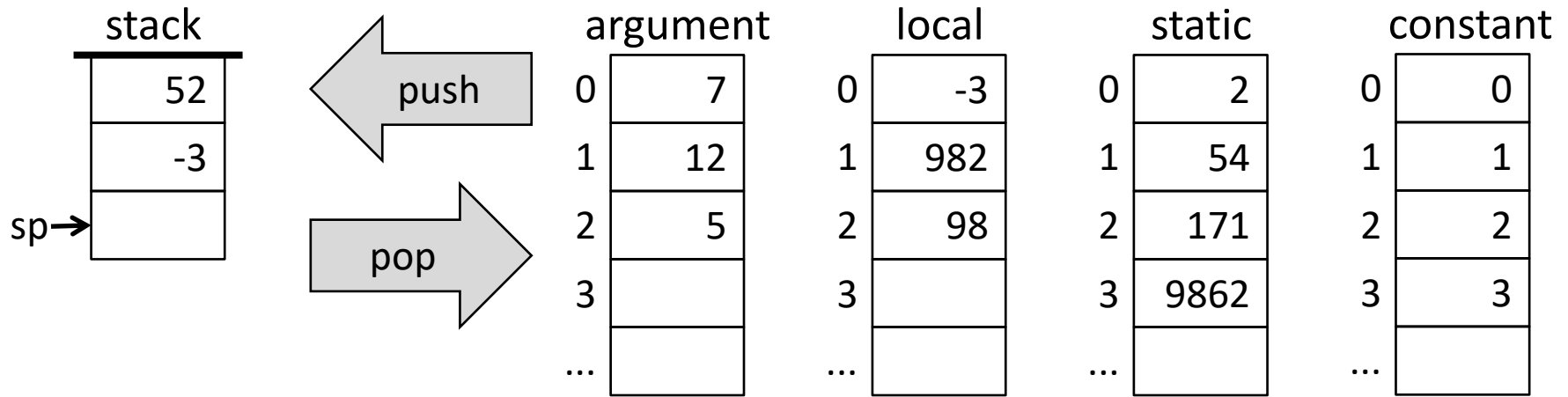
Virtual Machine (Part 2)

Dr. Wooi Ping Cheah

Outlines

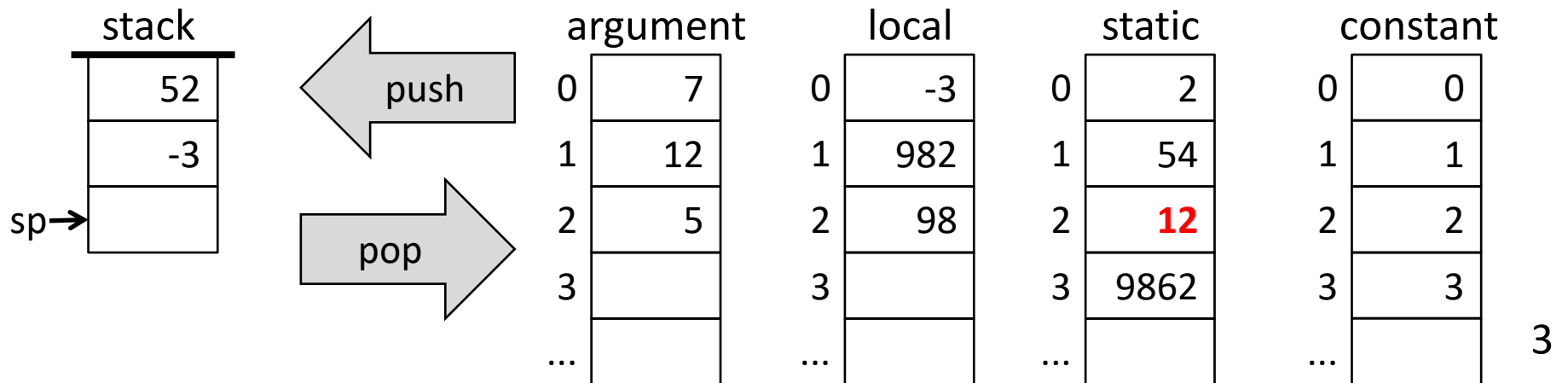
- Introduction to virtual machine
- VM abstraction
- VM implementation
 - Stack
 - Memory segment commands
 - Branching commands
 - Function commands
- VM translator

VM abstraction



let static 2 = argument 1

push argument 1
pop static 2



Pointer manipulation

Pseudo assembly code

```
D = *p // D becomes 23
p--    // RAM[0] becomes 256
D = *p // D becomes 19

*q = 9 // RAM[1024] becomes 9
q++    // RAM[1] becomes 1025
```

In Hack:

@p
A=M
D=M

Notation:

*p // the memory content that p points at

x-- // decrement: $x = x - 1$

x++ // increment: $x = x + 1$

RAM		
0	257	p
1	1024	q
2	1765	
...	...	
256	19	
257	23	
258	903	
...	...	
1024	5	
1025	12	
1026	-3	
...	...	

Pointer manipulation - exercise

Given the initial memory status shown on the right, find:

```
p++ // What is RAM[0]?  
D = *p //What is D?  
*q = D // What is *q?  
q++ //What is RAM[1]?  
*p = *q //What is *p?
```

RAM		
0	257	p
1	1024	
2	1765	q
...	...	
256	19	
257	23	
258	903	
...	...	
1024	5	
1025	12	
1026	-3	
...	...	

Pointer manipulation - answer

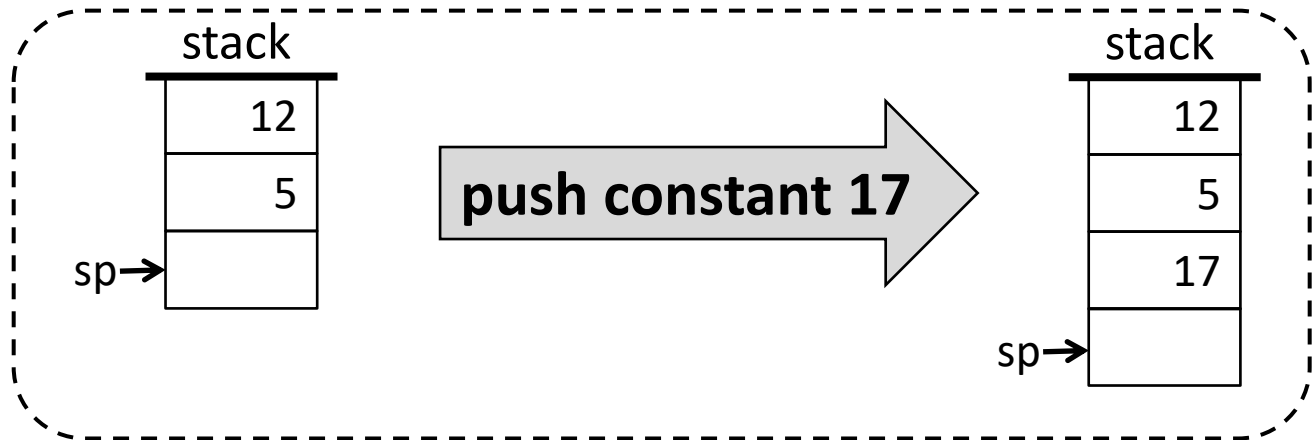
Given the initial memory status shown on the right, find:

```
p++ // What is RAM[0]? 258
D = *p //What is D? 903
*q = D // What is *q? 903
q++ //What is RAM[1]? 1025
*p = *q //What is *p? 12
```

RAM		
0	257	p
1	1024	q
2	1765	
...	...	
256	19	
257	23	
258	903	
...	...	
1024	5	
1025	12	
1026	-3	
...	...	

Stack implementation

Abstraction:



Implementation:

Assumptions:

- SP stored in RAM[0],
- Stack base addr = 256.

RAM		SP
0	258	
1	...	
2		
...		
256	12	
257	5	
258		
...		

Logic:

```
*SP = 17  
SP++
```

Hack assembly:

```
@17 // D=17  
D=A  
@SP // *SP=D  
A=M  
M=D  
@SP // SP++  
M=M+1
```

RAM		SP
0	259	
1	...	
2		
...		
256	12	
257	5	
258	17	
...		

Stack implementation

VM code:

push constant i

VM translator

Assembly psuedo code:

$*SP = i, SP++$

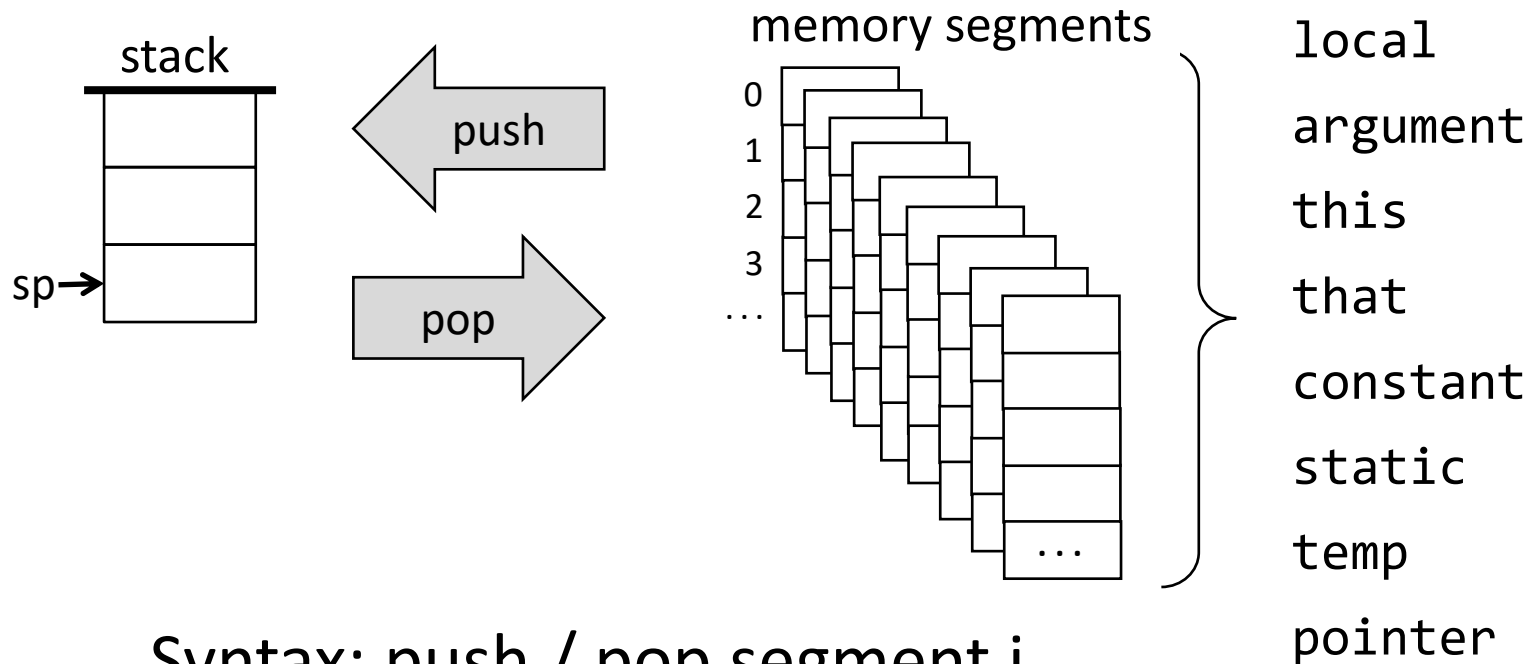
VM Translator

- A program that **translates** VM commands into lower-level commands of some host platform (like the Hack computer).
- Each VM command **generates** one or more low-level commands.
- The low-level commands **realize** the stack and the memory segments on the host platform.

Outlines

- Introduction to virtual machine
- VM abstraction
- VM implementation
 - Stack
 - Memory segment commands
 - Branching commands
 - Function commands
- VM translator

Memory segments (abstraction)

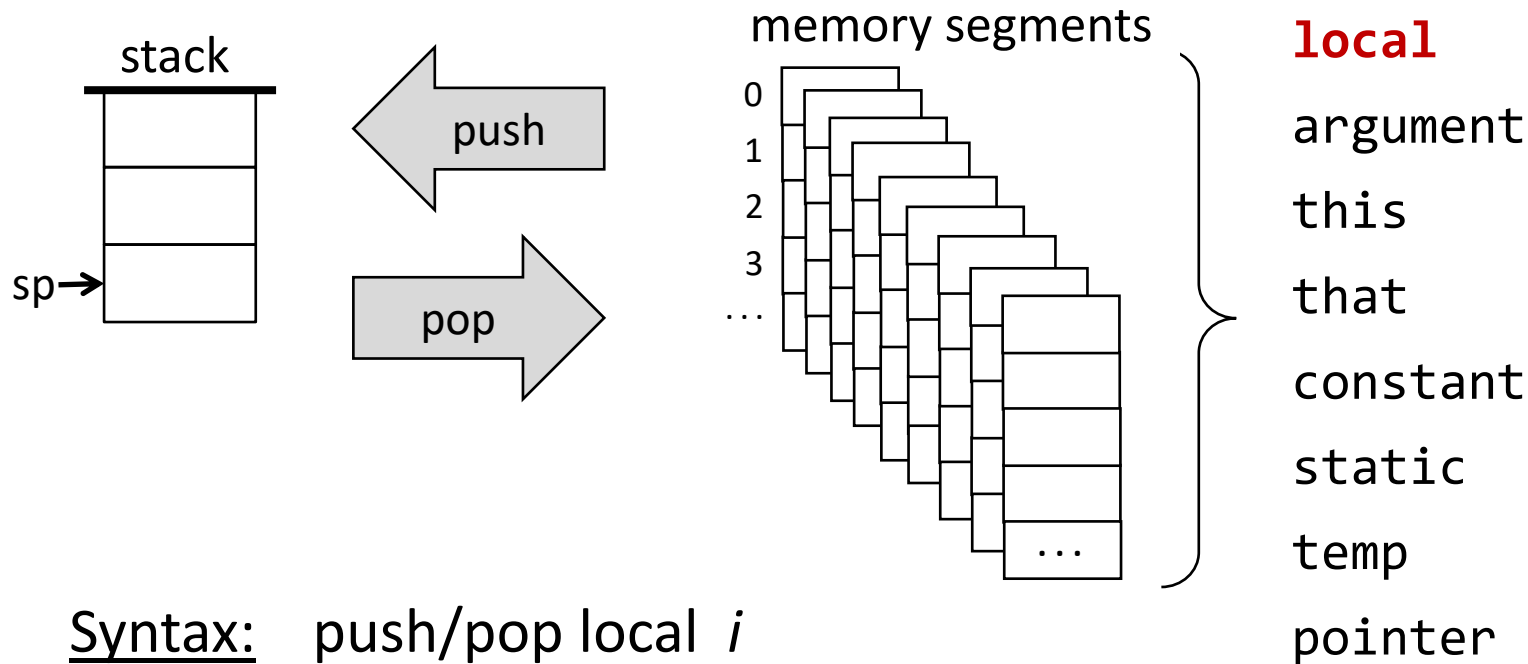


Syntax: push / pop segment i

Examples:

- push constant 17
- pop local 2
- pop static 5
- push argument 3

Implement `push/pop local i`



Syntax: `push/pop local i`

Why do we need a local segment?

- High-level code on ***local variables*** are translated into VM operations on the entries of the segment ***local***.

Implement `pop local i`

Abstraction:



stack pointer

base address of
the local segment

Implementation:

the local segment
is stored some-
where in the RAM

RAM		
0	258	SP
1	1015	LCL
2		
...		
256	12	
257	5	
258		
...		
1015	...	
1016	...	
1017	...	
...		

Implementation:

```
addr=LCL+ i, SP--, *addr=*SP
```

Hack assembly:

On next slide!

RAM		
0	257	SP
1	1015	LCL
2		
...		
256	12	
257	5	
258		
...		
1015	...	
1016	...	
1017	5	
...		

Implement `pop local i`

Abstraction

`pop local i`

Implementation:

`addr=LCL+ i, SP--, *addr=*SP`

i is a constant here!!!
but LCL is a variable.

Hack assembly:

```
@i      // addr=LCL+i
D=A
@LCL
D=D+M
@addr
M=D
@SP      // SP--
M=M-1
@SP      // D=*SP
A=M
D=M
@addr    // *addr=D
A=M
M=D
```

Implement push/pop local i

VM code:

```
pop local i
```

```
push local i
```

VM Translator

Assembly pseudo code:

```
addr = LCL + i, SP--, *addr = *SP
```

```
addr = LCL + i, *SP = *addr, SP++
```

Stack pointer

Base address of the local segment

Implementation:

The local segment is stored somewhere in the RAM

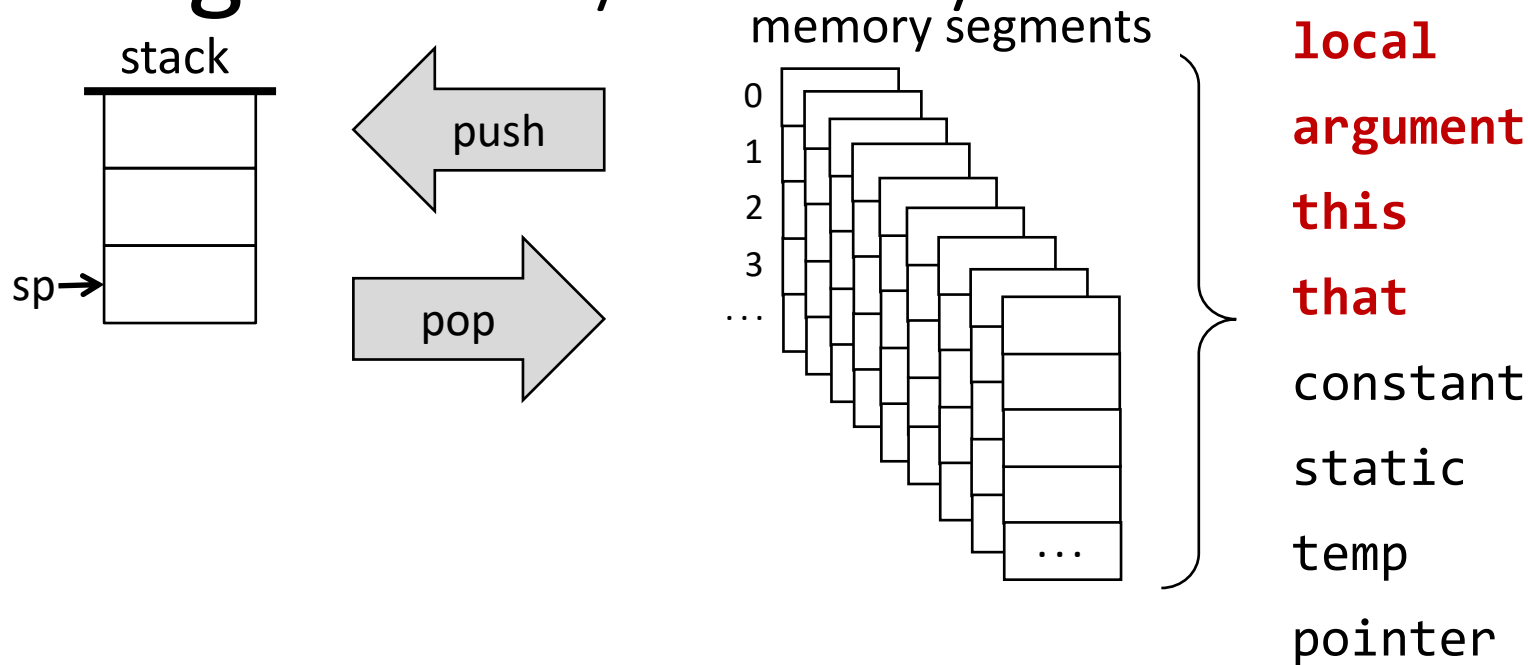
RAM		
0	258	SP
1	1015	LCL
2		
...		
256	12	
257	5	
258		
...		
1015	...	
1016	...	
1017	...	
...		

Hack assembly:

```
// implement
// push local i
// addr=LCL+i
@i
D=A
D=D+M
@addr
M=D
```

```
// *SP = *addr
@addr // D=*addr
A=M
D=M
@SP // *SP=D
A=M
M=D
// SP++
@SP
M=M+1
```

Implement push / pop local / argument / this / that *i*



Syntax: push/pop local/argument/this/that *i*

	High-level language	VM code
local	<i>local</i> variable	<i>local i</i>
argument	<i>argument</i> in a function call	<i>argument i</i>
this	<i>field</i> variables of the current object	<i>this i</i>
that	<i>array entries</i>	<i>that i</i>

Implement `push / pop local / argument / this / that i`

VM code:

```
push segment i
```

```
pop segment i
```

VM translator

Assembly pseudo code:

```
addr = segmentPointer + i, *SP = *addr, SP++
```

```
addr = segmentPointer + i, SP--, *addr = *SP
```

$segment = \{local, argument, this, that\}$

Host RAM

0		SP
1		LCL
2		ARG
3		THIS
4		THAT
5		
...		
12		
13		
14		
15		
...		

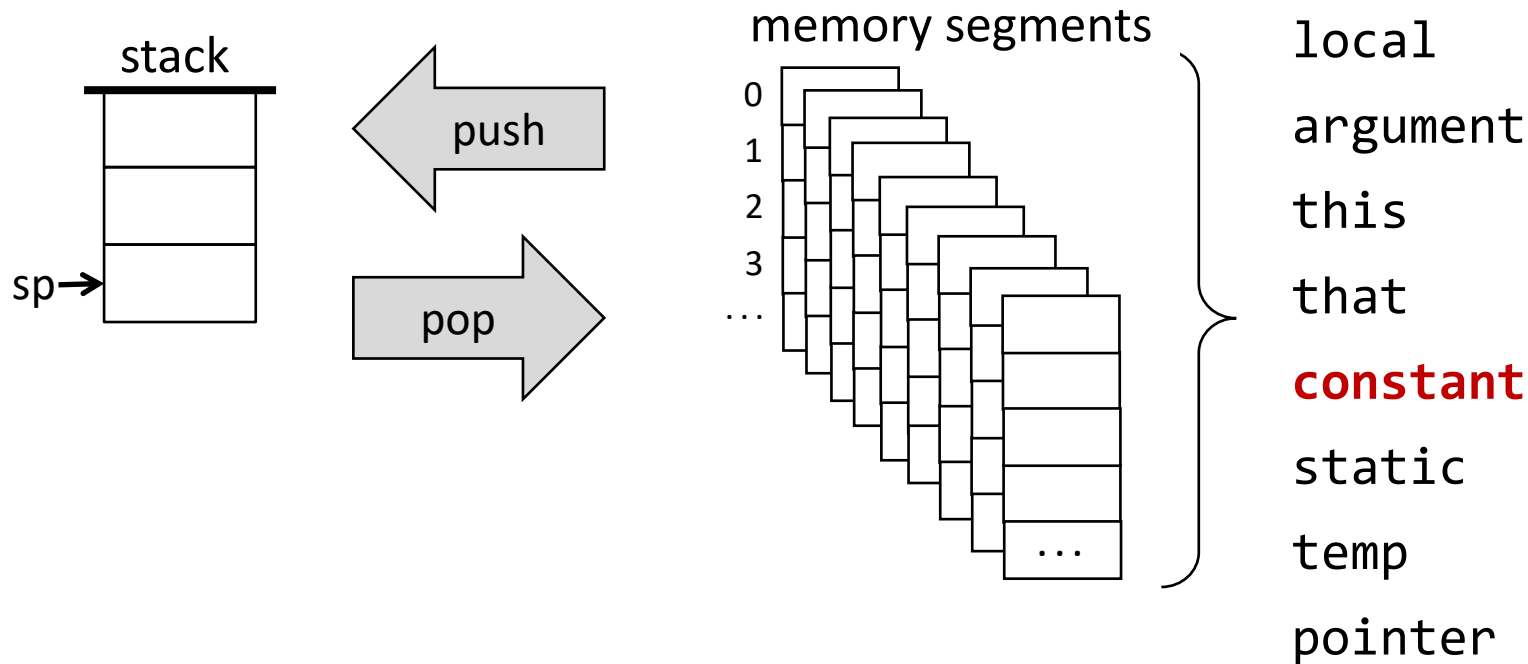
base addresses of
the four segments
are stored in these
pointers

the four segments
are stored
somewhere in the
RAM

- `push/pop local i`
- `push/pop argument i`
- `push/pop this i`
- `push/pop that i`

implemented
precisely the
same way.

Implement push constant i



Syntax: push constant i

Why do we need a constant segment?

- High-level code on the *constant* i are translated into VM operations on the segment entry constant i .

Implement push constant i

VM code:

push constant i

VM Translator

Assembly psuedo code:

$*SP = i, SP++$

(no pop constant operation)

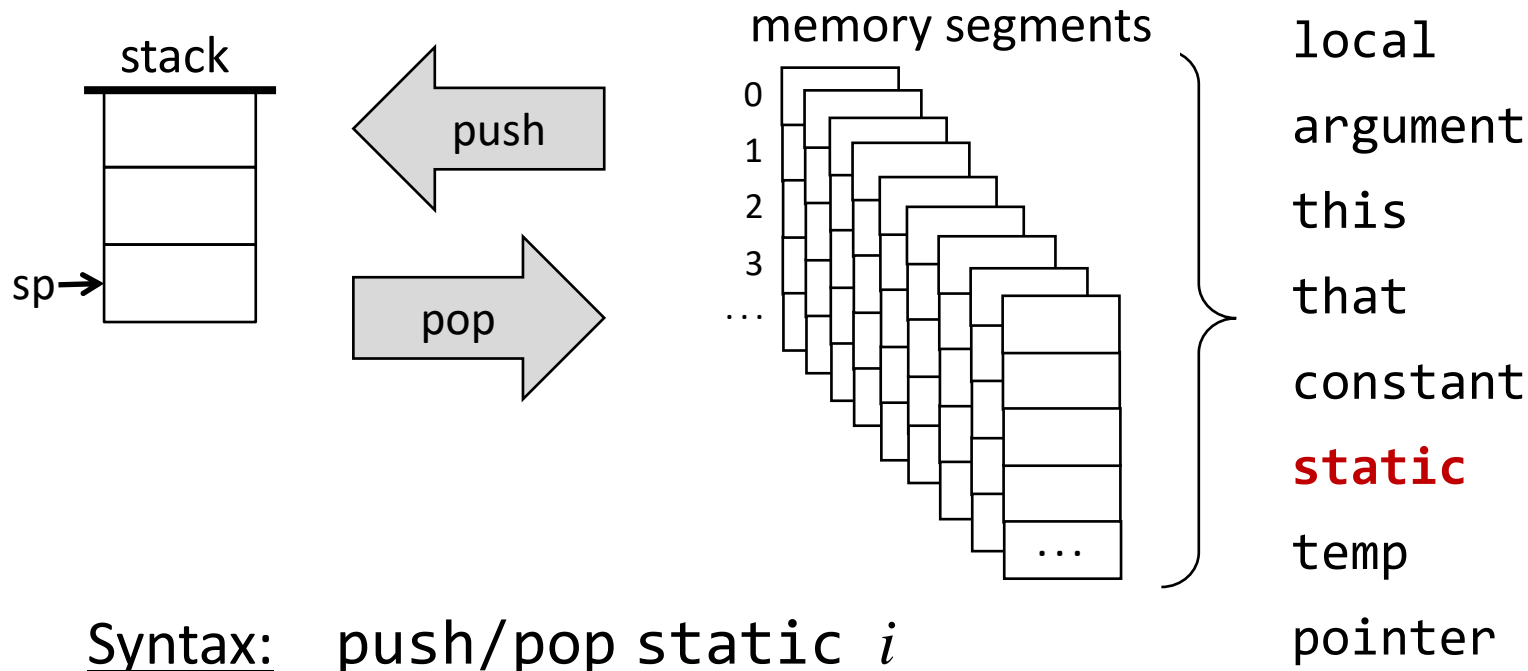
Implementation:

Supplies the specified constant.

Hack assembly:

```
// D = i
@i
D=A
// *SP=D
@SP
A=M
M=D
// SP++
@SP
M=M+1
```

Implementing `push/pop static i`



Syntax: `push/pop static i`

Why do we need a static segment?

- High-level operations on *static* variables are translated into VM operations on entries of the segment *static*.
- Static variables can be used as “global” variables, or to store constant values.

Implement push/pop static *i*

VM code:

```
// File Foo.vm
...
pop static 5
...
pop static 2
...
```

VM translator

Generated assembly code:

```
...
// D = stack.pop (code omitted)
@Foo.5
M=D
...
// D = stack.pop (code omitted)
@Foo.2
M=D
...
```

The challenge:

Static variables should be seen by all the methods in a program.

Solution:

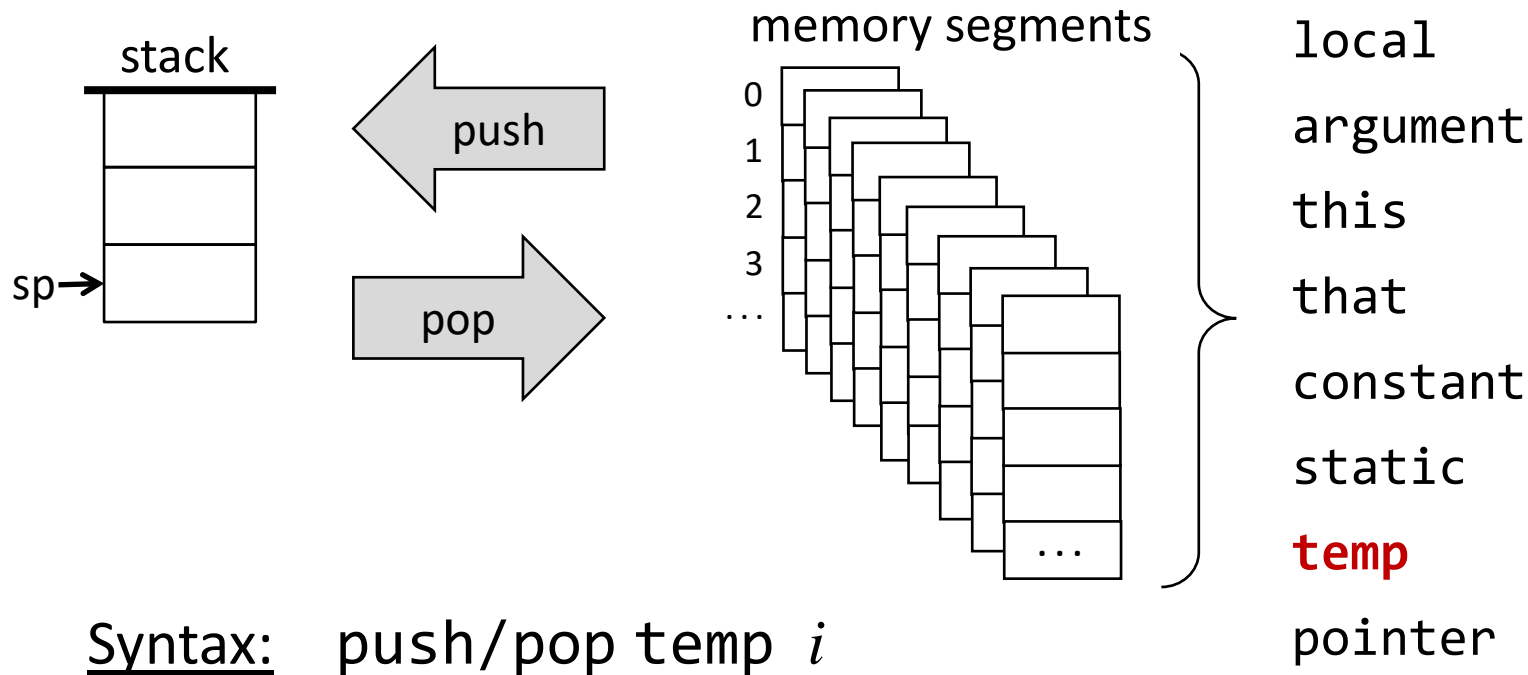
Store them in some “**global space**”:

- Have the VM translator translate each VM reference `static i` (in file `Foo.vm`) into an assembly reference `Foo.i`
- Following assembly, the Hack assembler will map these references onto `RAM[16]`, `RAM[17]`, ..., `RAM[255]`
- Therefore, the entries of the `static` segment will end up being mapped onto `RAM[16]`, `RAM[17]`, ..., `RAM[255]`, in the order in which they appear in the program.

Hack RAM

0		SP
1		LCL
2		ARG
3		THIS
4		THAT
5		
...		
12		
13		
14		
15		
16		} static variables
17		
...		
255		
256		
...		
2047		
...		

Implement `push/pop temp i`



Syntax: `push/pop temp i`

Why do we need the temp segment?

- So far, all the variable kinds that we discussed came from the source code.
- Sometimes, the **compiler** needs to use some working variables of its own.
- Our VM provides **8** such variables, stored in a segment named *temp*.

Implement push/pop temp i

VM code:

push temp i

pop temp i

VM Translator

Assembly psuedo code:

addr = $5 + i$, *SP = *addr, SP++

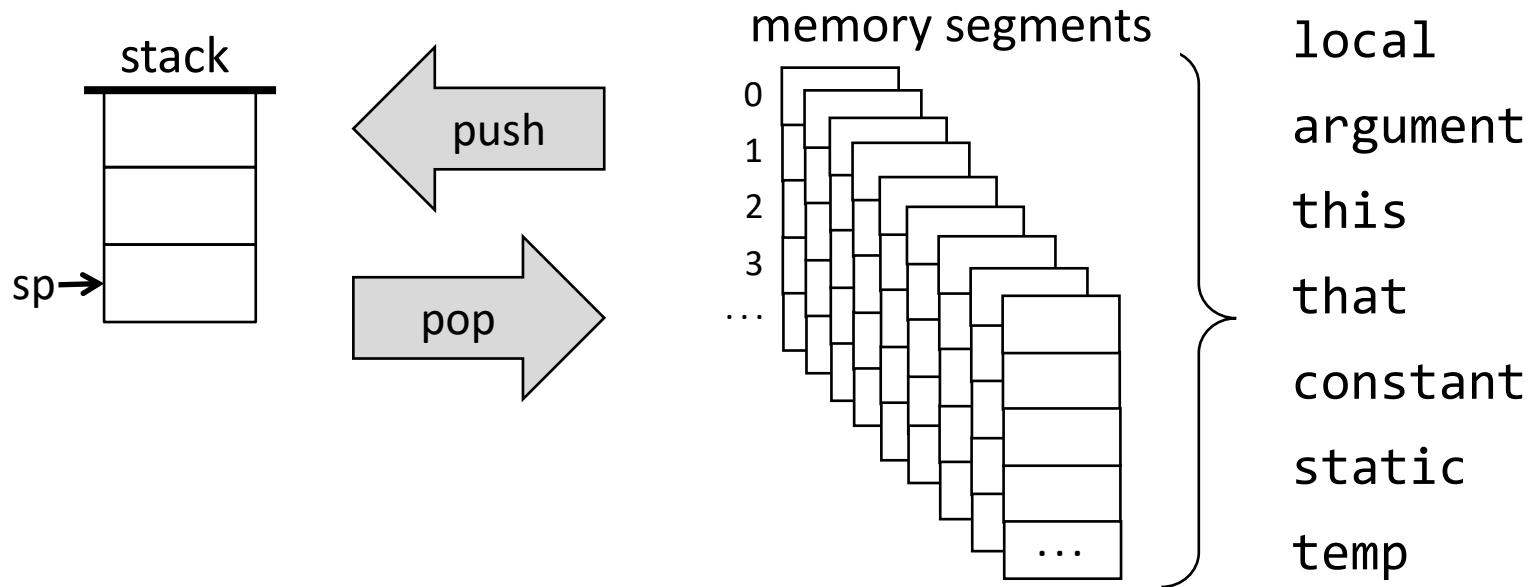
addr = $5 + i$, SP--, *addr = *SP

Hack RAM

0		SP
1		LCL
2		ARG
3		THIS
4		THAT
5		temp segment
...		
12		
13		
14		
15		
16		
...		
255		

A fixed, **8-place** memory segment, stored in RAM locations 5 to 12

Implement push/pop pointer 0/1



Syntax: push/pop pointer 0/1

Why do we need the *pointer* segment?

- We use it for storing the **base addresses** of the segments **this** and **that**.
- The need for this will become clear when writing the compiler.

Implement push/pop pointer 0/1

VM code:

push pointer 0/1

pop pointer 0/1

VM Translator

Assembly psuedo code:

*SP = THIS/THAT, SP++

SP--, THIS/THAT = *SP

A fixed, 2-place segment:

- accessing pointer 0 should result in accessing THIS
- accessing pointer 1 should result in accessing THAT

Implementation:

Supplies THIS or THAT // The base addresses of this and that.

// THIS and THAT: Built-in symbols.

Outlines

- Introduction to virtual machine
- VM abstraction
- VM implementation
 - Stack
 - Memory segment commands
 - Branching commands
 - Function commands
- VM translator

Branching

- *goto label*
 - jump to execute the command just after *label*
- *if-goto label*
 - *cond* = pop
 - if *cond* jump to execute the command just after *label*
- *label label*
 - label declaration command
- Implementation (VM translation):
 - The assembly language has similar branching commands.

Outlines

- Introduction to virtual machine
- VM abstraction
- VM implementation
 - Stack
 - Memory segment commands
 - Branching commands
 - Function commands
- VM translator

Functions in VM language: implementation

```
// 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

```
// Computes the product of two given arguments
```

```
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
```

```
//... computes the product into local 0
```

```
19 label END
```

```
20 push local 0
```

```
21 return
```

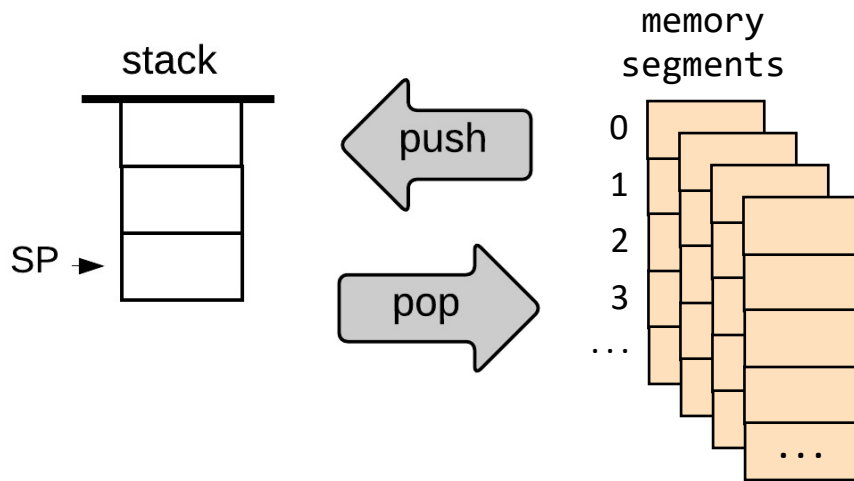
callee

Implementation

We can write low-level code to

- Handle the VM command `call`,
- Handle the VM command `function`,
- Handle the VM command `return`.

The function's state



During run-time:

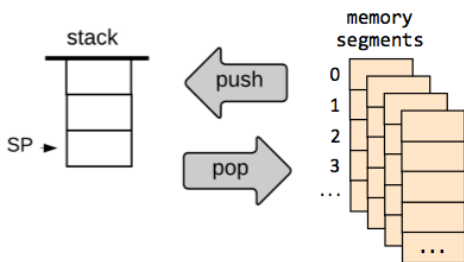
- Each function uses a **working stack + memory segments**
- The working stack and some of the segments should be:
 - **Created** when the function starts running,
 - **Maintained** as long as the function is executing,
 - **Recycled** when the function returns.

The function's state

function main 0

```
push constant 3
push constant 8
push constant 5
call mult 2
add
return
```

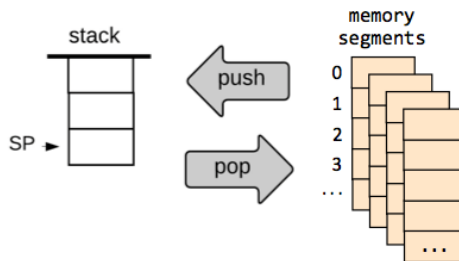
caller



function mult 2

```
push constant 0
pop local 0
...
label LOOP
push local 1
//... computes the product
label END
push local 0
return
```

callee

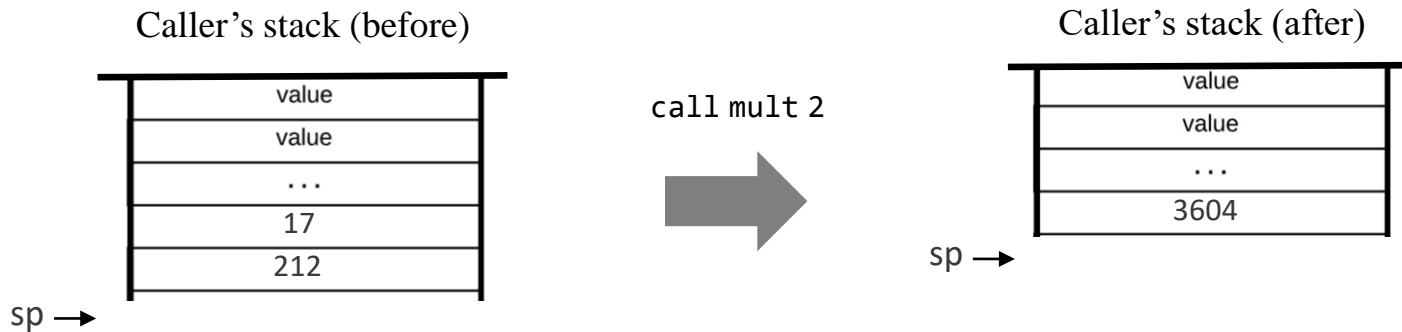


Challenge:

- Maintain the states of all the functions up the calling chain.
- Can be done by using a single *global stack*.

Function call and return: abstraction

Example: computing `mult(17,212)`

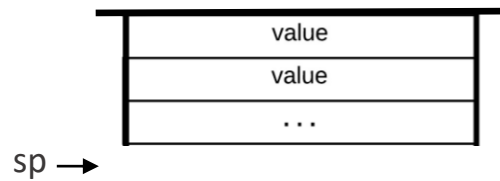


Net effect:

The function's arguments were replaced by the function's return value

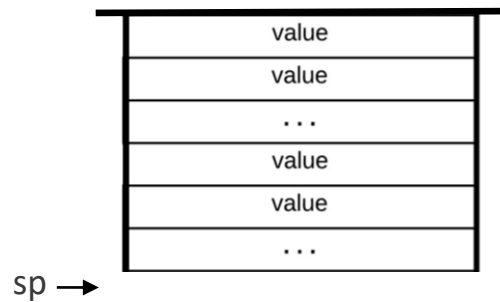
Function call and return: implementation

A function is running, and
doing something



Function call and return: implementation

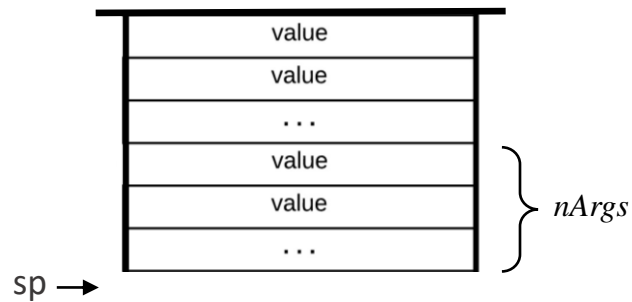
The function prepares
to call another function:



Function call and return: implementation

The function says:

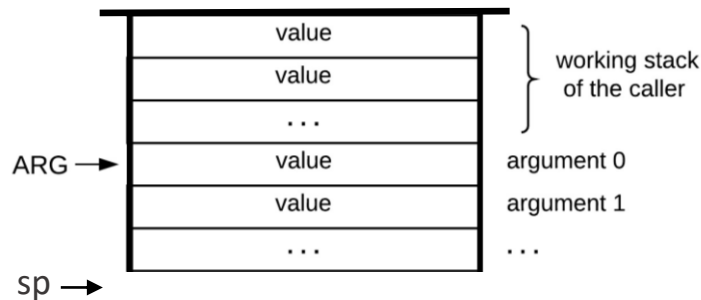
call foo nArgs



Function call and return: implementation

The function says:

call foo nArgs



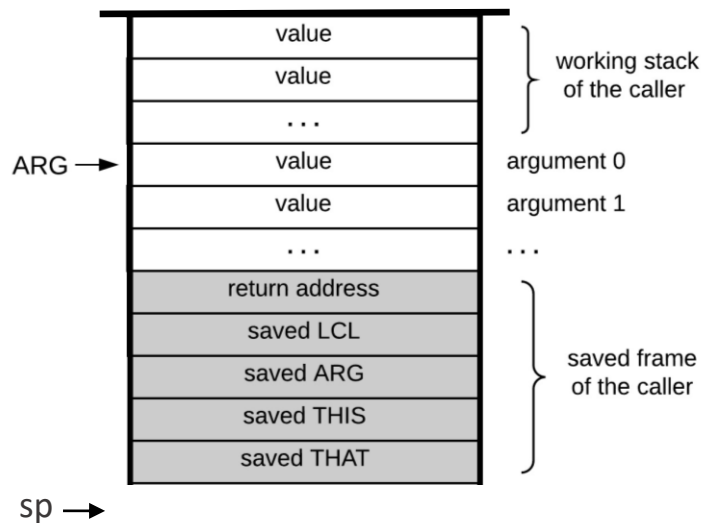
VM implementation (handling call):

1. Set ARG

Function call and return: implementation

The function says:

call foo nArgs



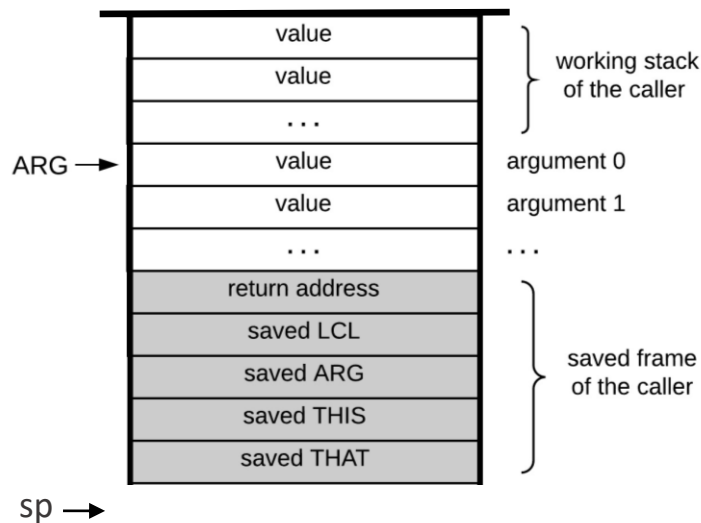
VM implementation (handling call):

1. Set ARG
2. Save the caller's frame

Function call and return: implementation

The called function is entered:

function *foo* *nVars*



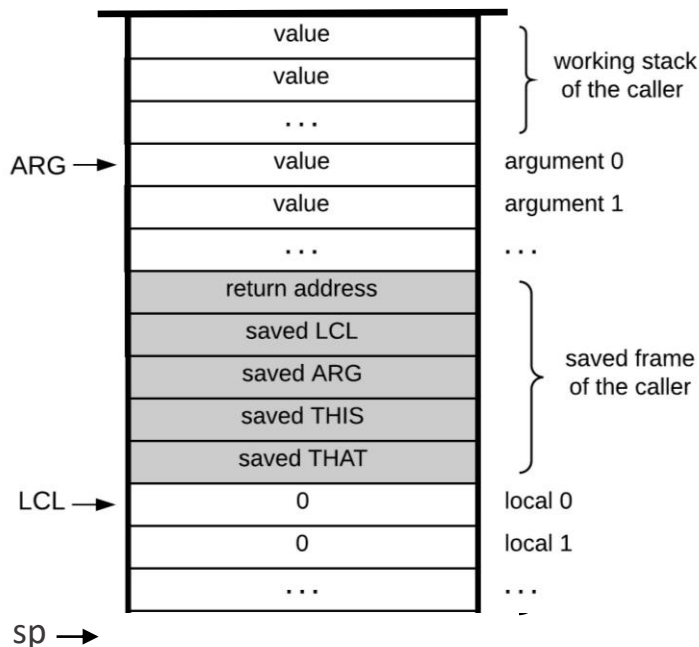
VM implementation (handling call):

1. Set ARG
2. Save the caller's frame
3. Jump to execute *foo*

Function call and return: implementation

The called function is entered:

function *foo* *nVars*



VM implementation (handling call):

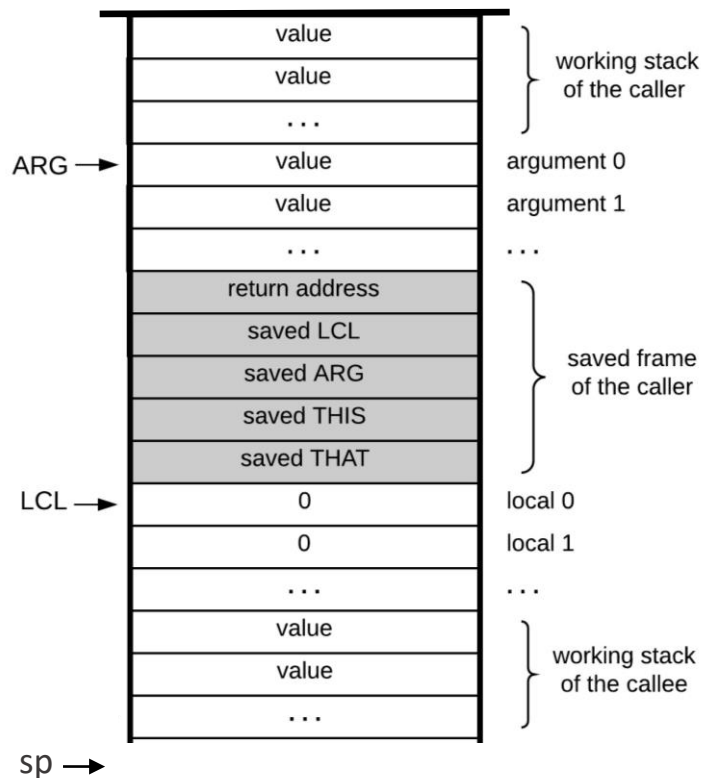
1. Set ARG
2. Save the caller's frame
3. Jump to execute *foo*

VM implementation (handling function):

Set up the local segment
of the called function

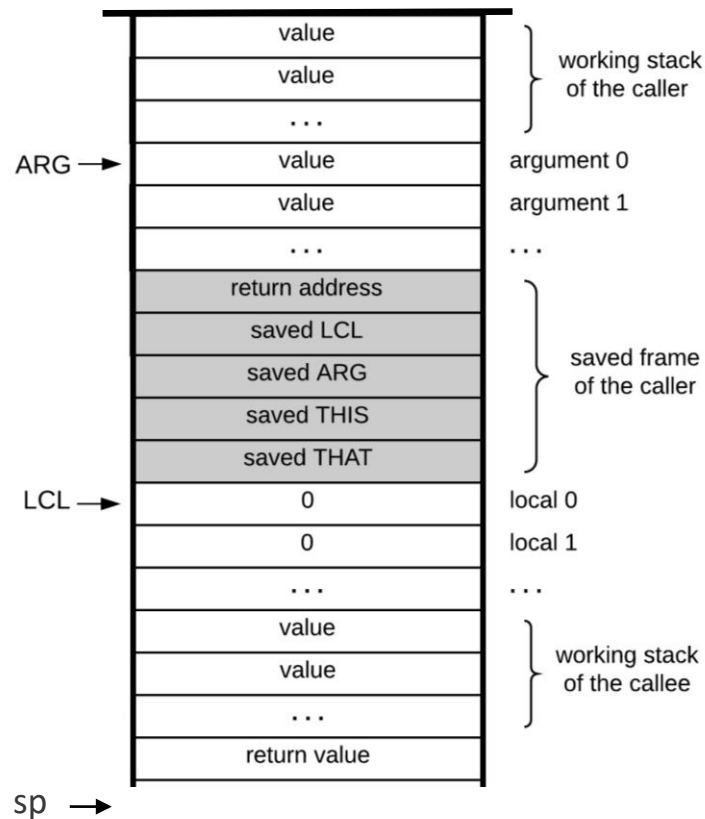
Function call and return: implementation

The called function is running,
doing something



Function call and return: implementation

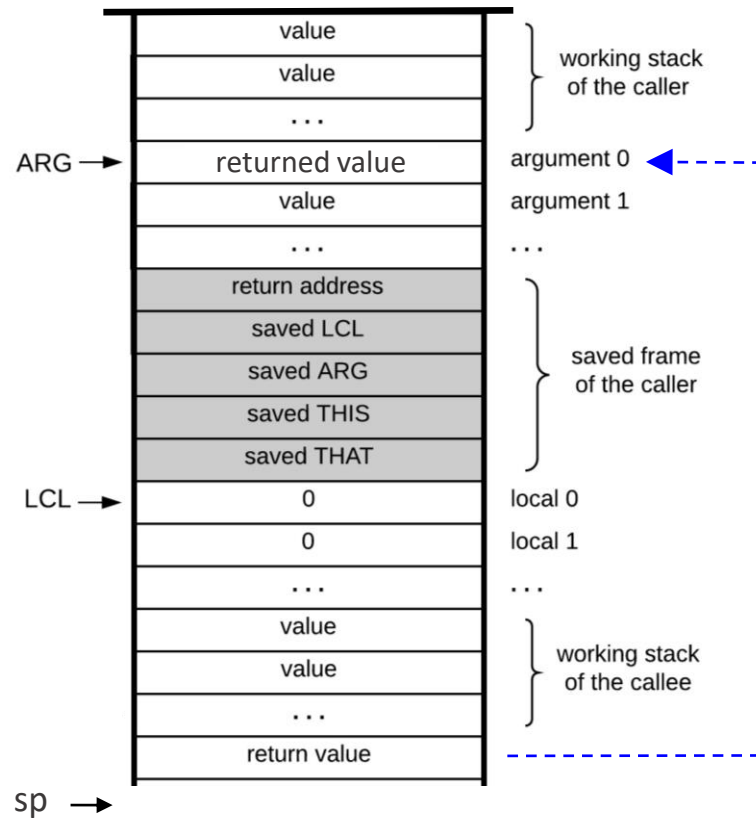
The called function prepares to return:
it pushes a *return value*, and says *return*.



Function call and return: implementation

The called function says:

return



VM implementation (handling call):

1. Set ARG
2. Save the caller's frame
3. Jump to execute *foo*

VM implementation (handling function):

Set up the local segment
of the called function

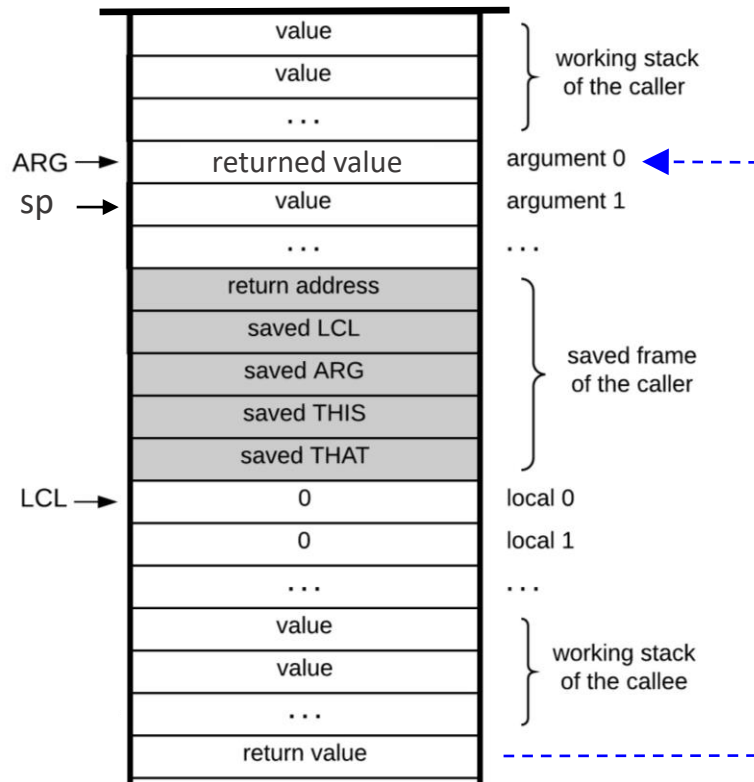
VM implementation (handling return):

1. Copy return value onto argument 0.

Function call and return: implementation

The called function says:

return



VM implementation (handling call):

1. Set ARG
2. Save the caller's frame
3. Jump to execute *foo*

VM implementation (handling function):

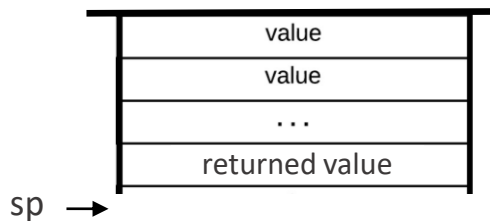
Set up the local segment
of the called function

VM implementation (handling return):

1. Copy return value onto argument 0.
2. Set SP for the caller.

Function call and return: implementation

The caller
resumes its execution



VM implementation (handling call):

1. Set ARG
2. Save the caller's frame
3. Jump to execute *foo*

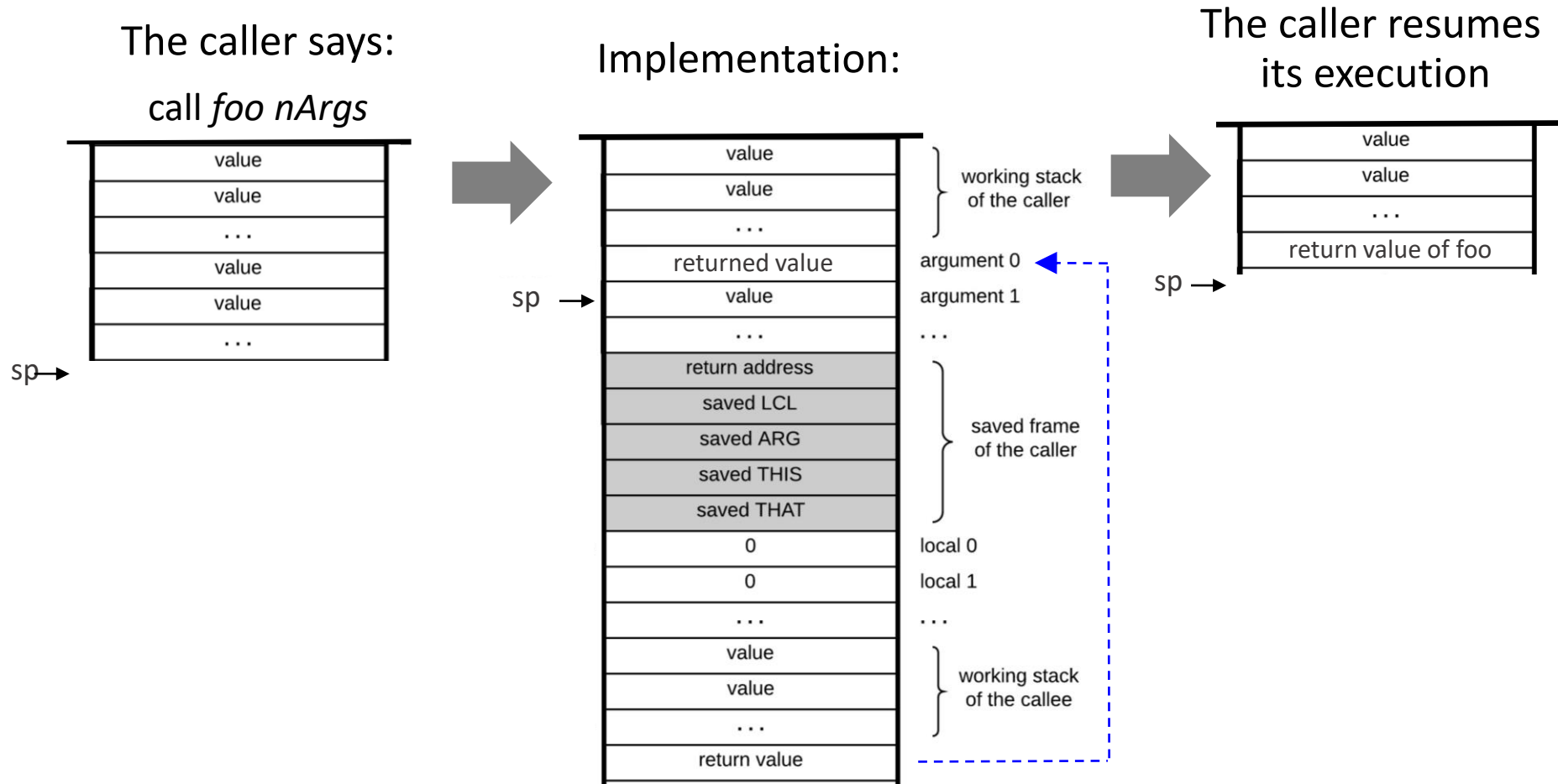
VM implementation (handling function):

Set up the local segment
of the called function

VM implementation (handling return):

1. Copy return value onto argument 0.
2. Set SP for the caller.
3. Restore segment pointers of the caller.
4. Jump to the return address within the caller's code. (note that the stack space below sp is recycled)

Recap: function call and return



Example: factorial

High-level program

```
// Tests the factorial function
int main() {
    return factorial(3);
}

// Returns n!
int factorial(int n) {
    if (n==1)
        return 1;
    else
        return n * factorial(n-1);
}
```



compiler

Pseudo VM code

```
function main
    push 3
    call factorial
    return

function factorial(n)
    push n
    push 1
    eq
    if-goto BASECASE

    push n
    push n
    push 1
    sub
    call factorial
    call mult
    return

label BASECASE
    push 1
    return

function mult(a,b)
    // Code omitted
```

VM program

```
function main 0
    push constant 3
    call factorial 1
    return

function factorial 1
    push argument 0
    push constant 1
    eq
    if-goto BASECASE

    push argument 0
    push argument 0
    push constant 1
    sub
    call factorial 1
    call mult 2
    return

label BASECASE
    push constant 1
    return

function mult 2
    // Code omitted
```

Run-time example

VM program

function main 0

```
push constant 3  
call factorial 1  
return
```

function factorial 1

```
push argument 0  
push constant 1  
eq  
if-goto BASECASE  
push argument 0  
push argument 0  
push constant 1  
sub  
call factorial 1  
call mult 2  
return  
label BASECASE  
push constant 1  
return
```

function mult 2

```
// Code omitted
```

global stack



Run-time example

VM program

function main 0

```
push constant 3  
call factorial 1  
return
```

function factorial 1

```
push argument 0  
push constant 1  
eq  
if-goto BASECASE  
push argument 0  
push argument 0  
push constant 1  
sub  
call factorial 1  
call mult 2  
return  
label BASECASE  
push constant 1  
return
```

function mult 2

```
// Code omitted
```

global stack

main:	3
-------	---

Run-time example

VM program

function main 0

push constant 3

call factorial 1

return

function factorial 1

push argument 0

push constant 1

eq

if-goto BASECASE

push argument 0

push argument 0

push constant 1

sub

call factorial 1

call mult 2

return

label BASECASE

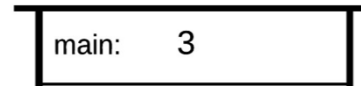
push constant 1

return

function mult 2

// Code omitted

global stack



argument 0

Run-time example

function main 0

push constant 3
call factorial 1
return

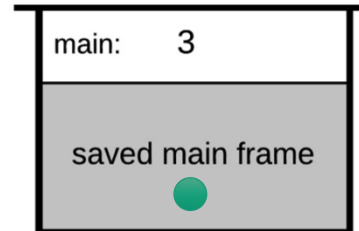
function factorial 1

push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return
label BASECASE
push constant 1
return

function mult 2

// Code omitted

global stack



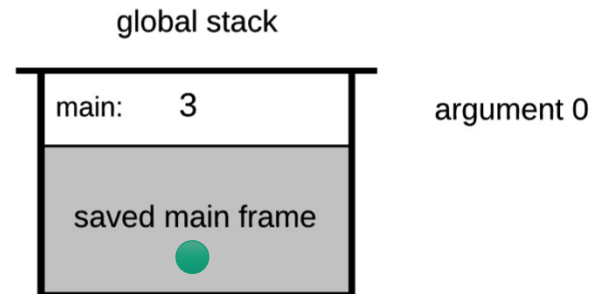
argument 0

Run-time example

function main 0
push constant 3
call factorial 1
return

function factorial 1
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return
label BASECASE
push constant 1
return

function mult 2
// Code omitted



impact on the
global stack
not shown

Run-time example

function main 0

push constant 3
call factorial 1
return

function factorial 1

push argument 0
push constant 1
eq
if-goto BASECASE

push argument 0
push argument 0
push constant 1
sub

call factorial 1
call mult 2
return

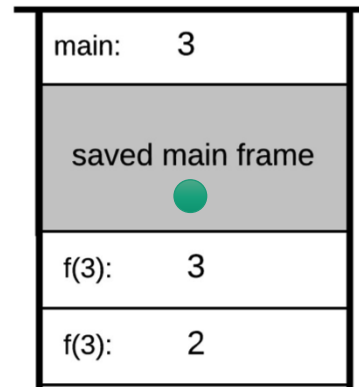
label BASECASE

push constant 1
return

function mult 2

// Code omitted

global stack



argument 0

Run-time example

function main 0

```
push constant 3
call factorial 1
return
```

function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
```

```
call factorial 1
```

```
call mult 2
```

```
return
```

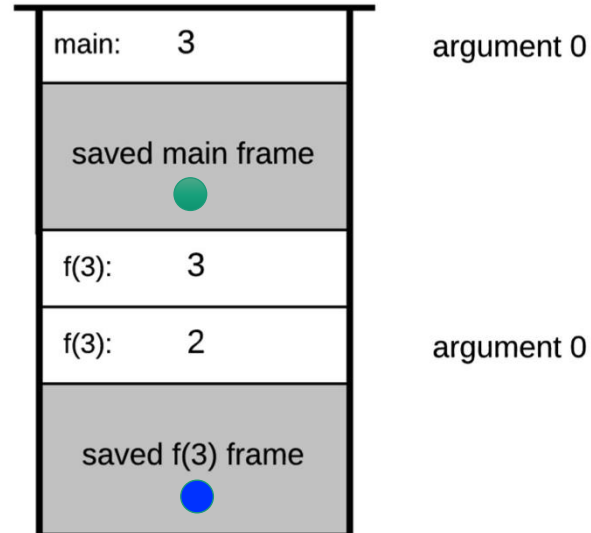
```
label BASECASE
```

```
push constant 1
return
```

function mult 2

```
// Code omitted
```

global stack



Run-time example

function main 0

```
push constant 3
call factorial 1
return
```

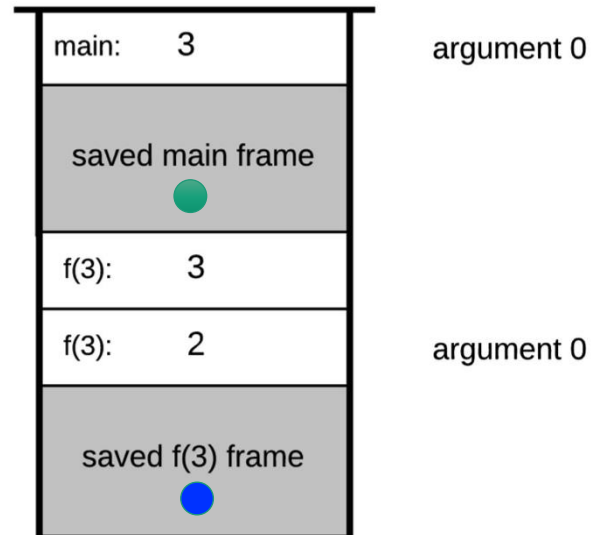
function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return
label BASECASE
push constant 1
return
```

function mult 2

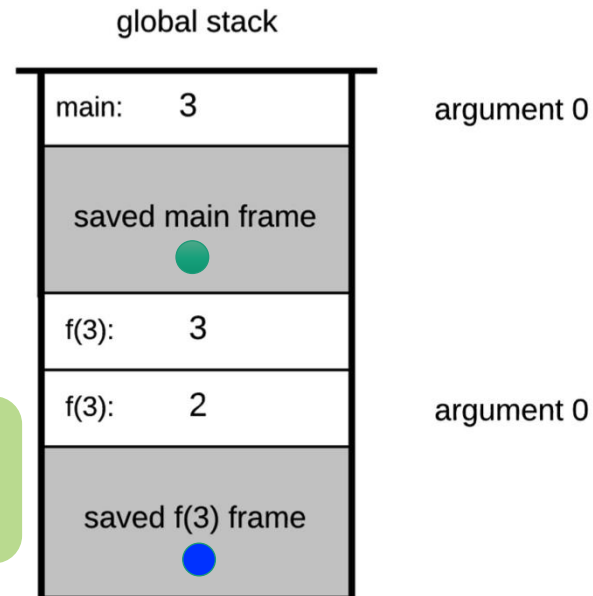
```
// Code omitted
```

global stack



Run-time example

```
function main 0  
  push constant 3  
  call factorial 1  
  return  
  
function factorial 1  
  push argument 0  
  push constant 1  
  eq  
  if-goto BASECASE  
  push argument 0  
  push argument 0  
  push constant 1  
  sub  
  call factorial 1  
  call mult 2  
  return  
label BASECASE  
  push constant 1  
  return  
  
function mult 2  
  // Code omitted
```



Run-time example

function main 0

```
push constant 3  
call factorial 1  
return
```

function factorial 1

```
push argument 0  
push constant 1  
eq  
if-goto BASECASE
```

```
push argument 0  
push argument 0  
push constant 1  
sub
```

```
call factorial 1  
call mult 2  
return
```

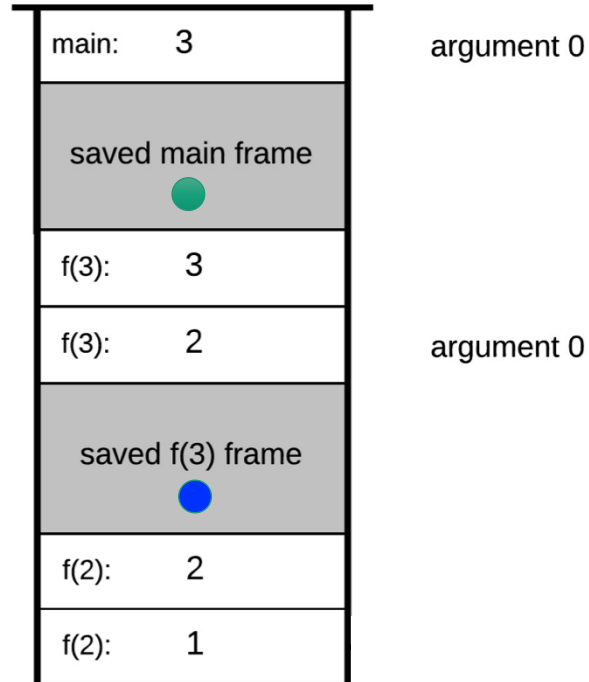
label BASECASE

```
push constant 1  
return
```

function mult 2

```
// Code omitted
```

global stack



Run-time example

function main 0

```
push constant 3
call factorial 1
return
```

function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
```

```
call factorial 1
```

```
call mult 2
```

```
return
```

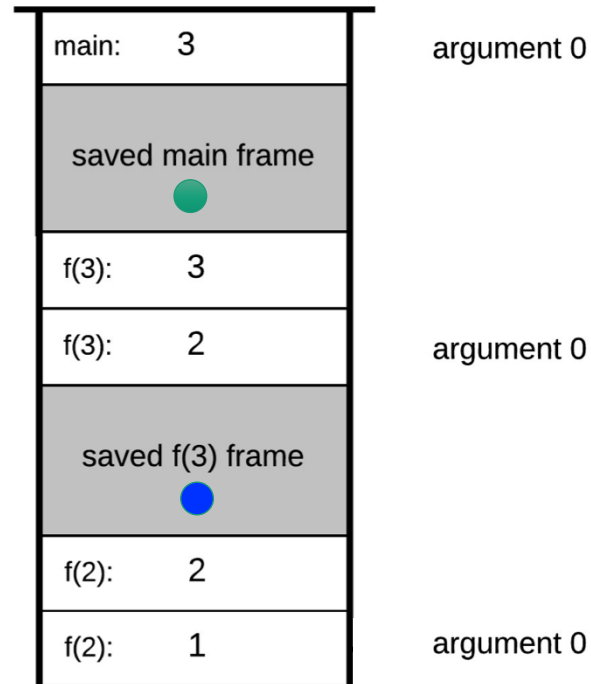
label BASECASE

```
push constant 1
return
```

function mult 2

```
// Code omitted
```

global stack



Run-time example

function main 0

push constant 3
call factorial 1
return

function factorial 1

push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub

call factorial 1

call mult 2

return

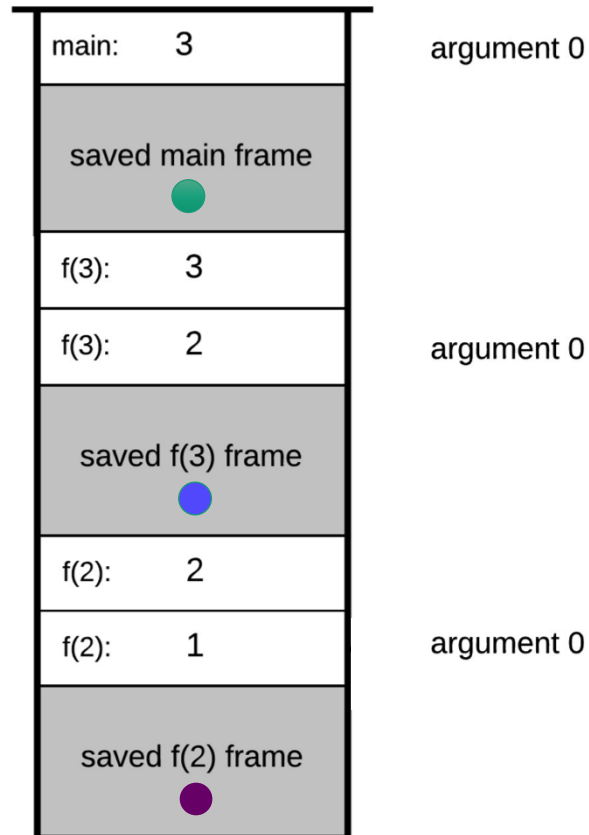
label BASECASE

push constant 1
return

function mult 2

// Code omitted

global stack



Run-time example

function main 0

```
push constant 3
call factorial 1
return
```

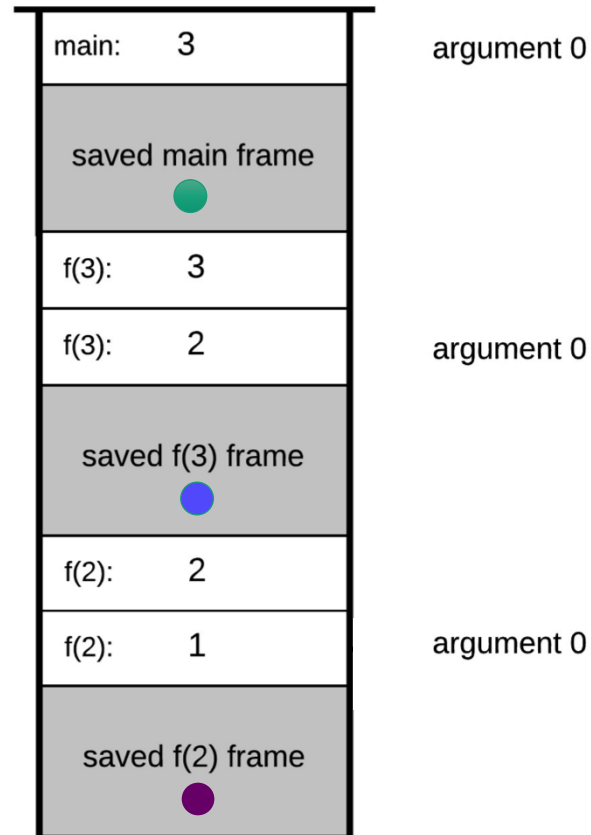
function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return
label BASECASE
push constant 1
return
```

function mult 2

```
// Code omitted
```

global stack



Run-time example

function main 0

```
push constant 3  
call factorial 1  
return
```

function factorial 1

```
push argument 0  
push constant 1  
eq  
if-goto BASECASE
```

```
push argument 0  
push argument 0  
push constant 1  
sub  
call factorial 1  
call mult 2  
return
```

label BASECASE

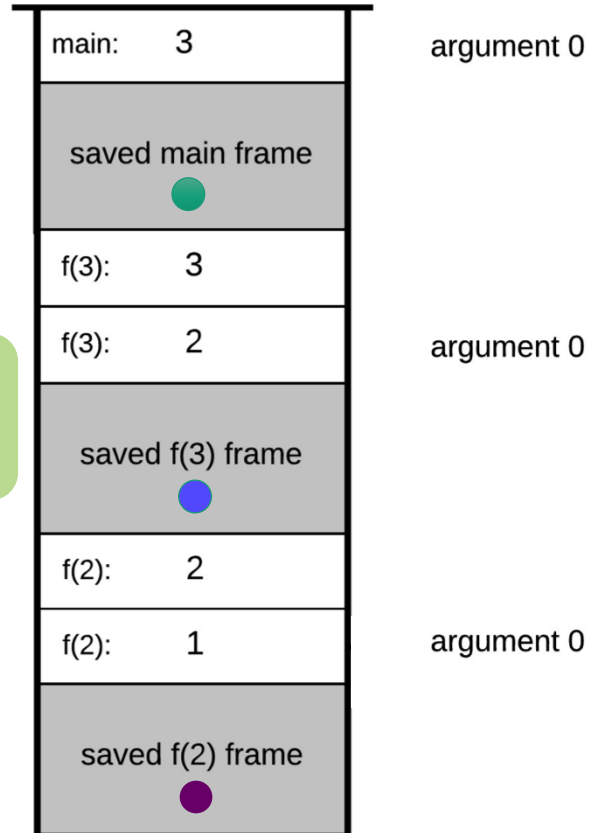
```
push constant 1  
return
```

function mult 2

```
// Code omitted
```

impact on the
global stack
not shown

global stack



Run-time example

function main 0

push constant 3
call factorial 1
return

function factorial 1

push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return

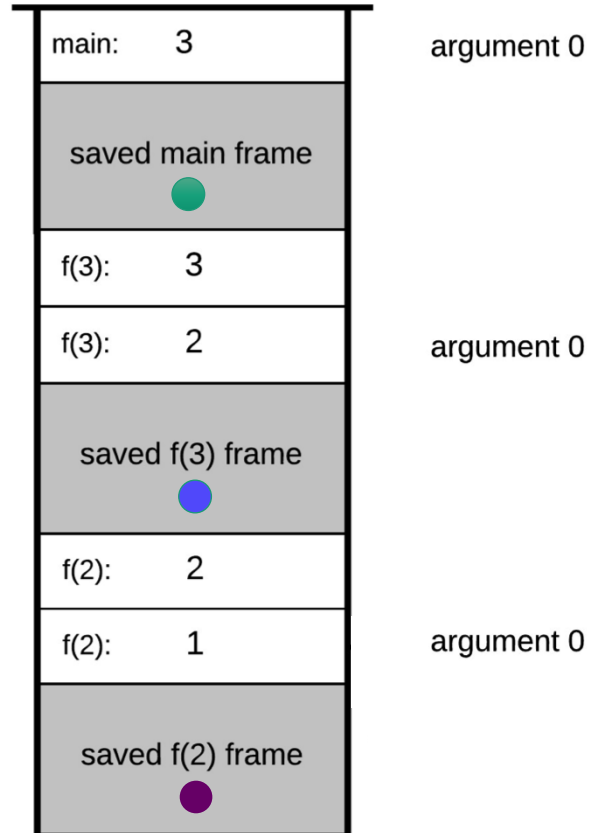
label BASECASE

push constant 1
return

function mult 2

// Code omitted

global stack



Run-time example

function main 0

push constant 3
call factorial 1
return

function factorial 1

push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return

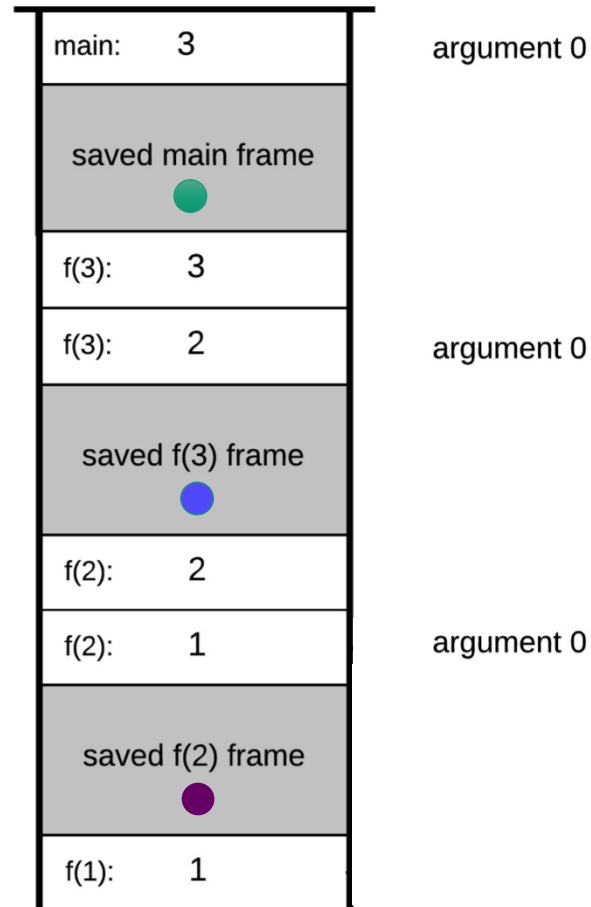
label BASECASE

push constant 1
return

function mult 2

// Code omitted

global stack



Run-time example

function main 0

```
push constant 3
call factorial 1
return
```

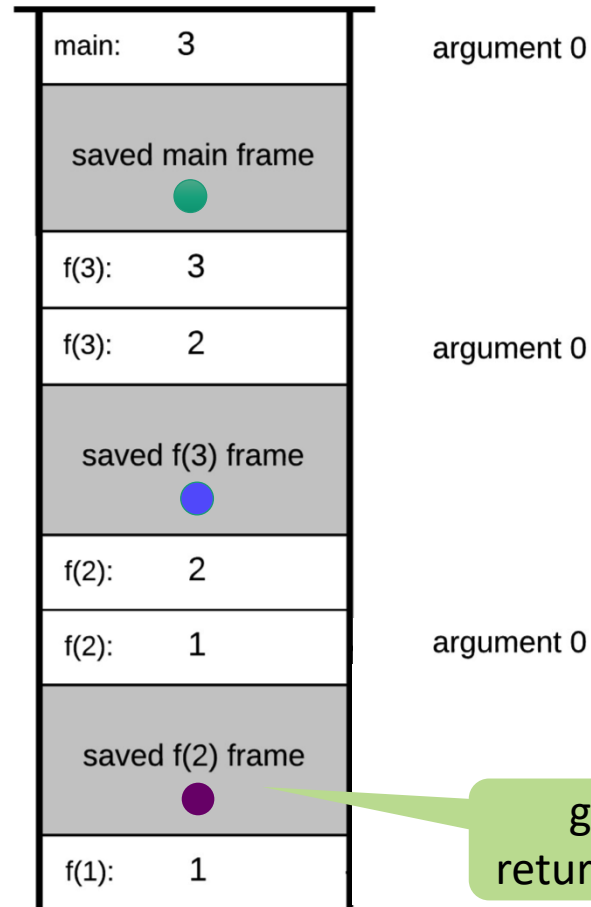
function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return
label BASECASE
push constant 1
return
```

function mult 2

```
// Code omitted
```

global stack



Run-time example

function main 0

```
push constant 3
call factorial 1
return
```

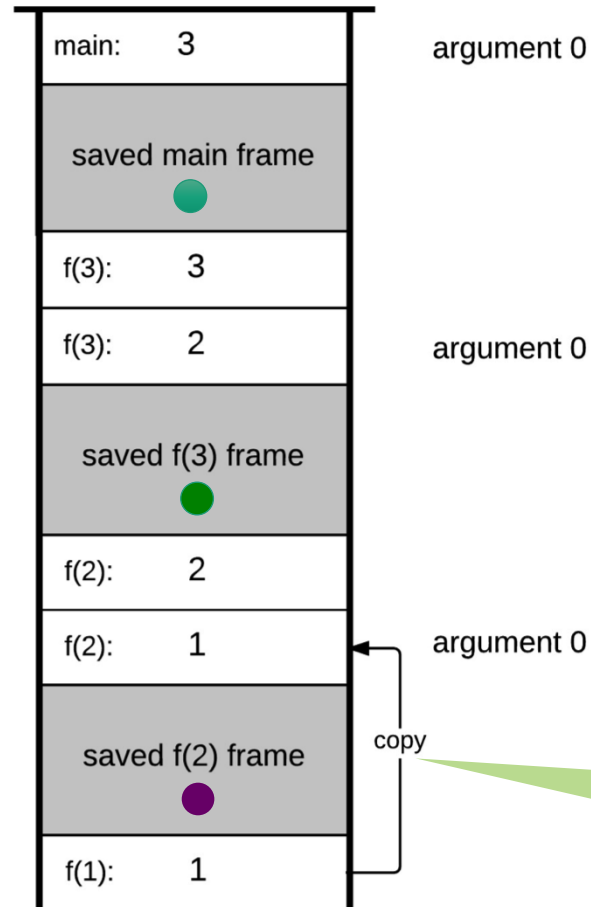
function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return
label BASECASE
push constant 1
return
```

function mult 2

```
// Code omitted
```

global stack



Run-time example

function main 0

```
push constant 3
call factorial 1
return
```

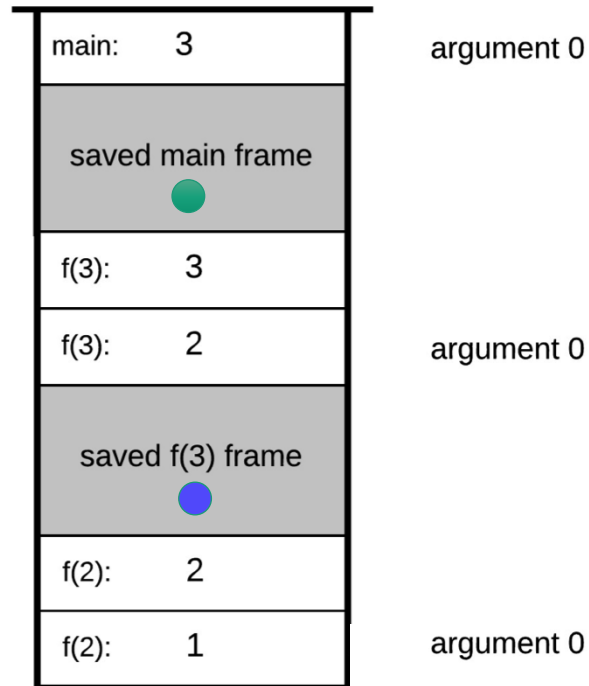
function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return
label BASECASE
push constant 1
return
```

function mult 2

```
// Code omitted
```

global stack



Run-time example

function main 0

```
push constant 3
call factorial 1
return
```

function factorial 1

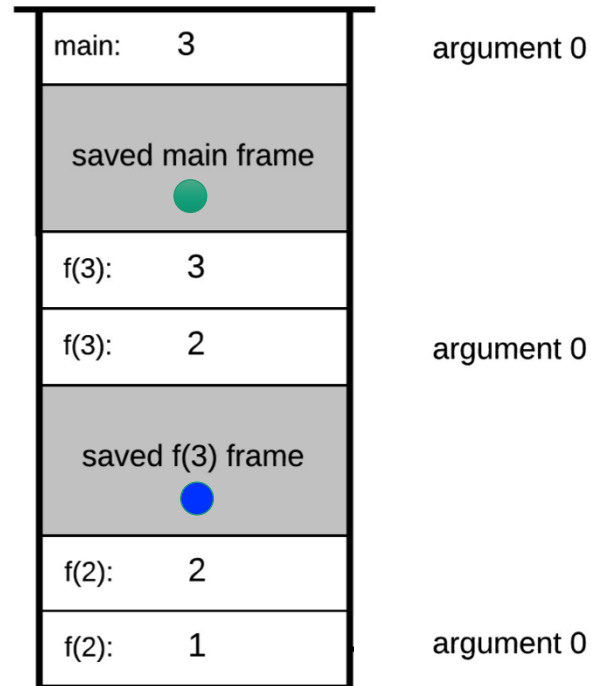
```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return
```

```
label BASECASE
push constant 1
return
```

function mult 2

```
// Code omitted
```

global stack



impact on the global
stack not shown
(except for end result)

Run-time example

function main 0

push constant 3
call factorial 1
return

function factorial 1

push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1

call mult 2

return

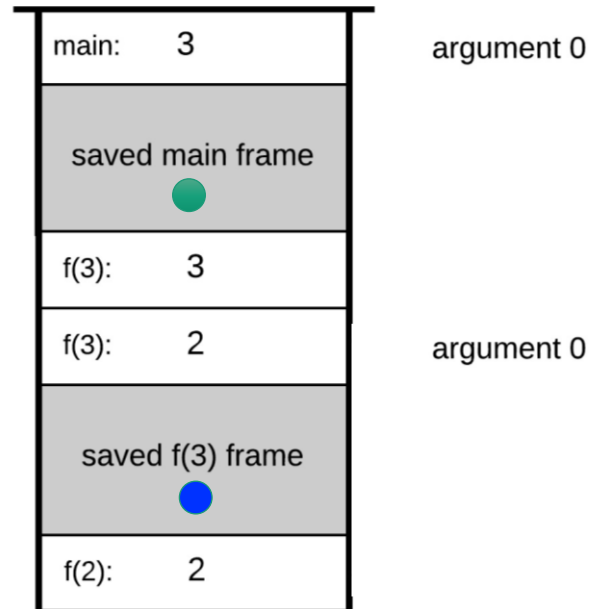
label BASECASE

push constant 1
return

function mult 2

// Code omitted

global stack



Run-time example

function main 0

```
push constant 3
call factorial 1
return
```

function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
```

```
return
```

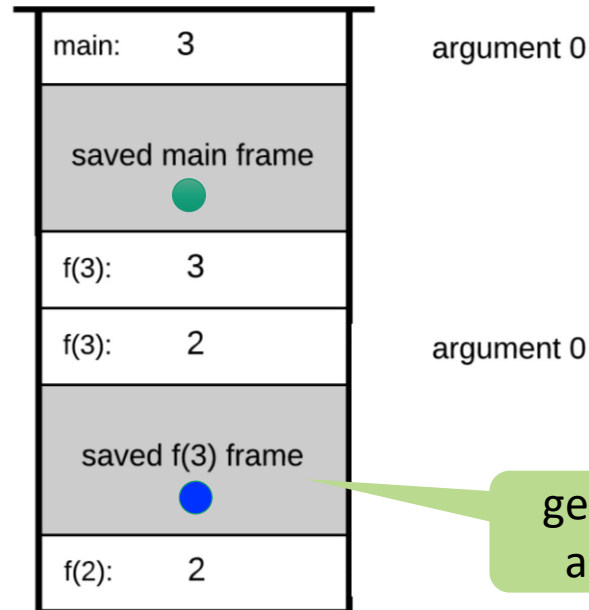
label BASECASE

```
push constant 1
return
```

function mult 2

```
// Code omitted
```

global stack



get return
address

Run-time example

function main 0

```
push constant 3
call factorial 1
return
```

function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
```

```
return
```

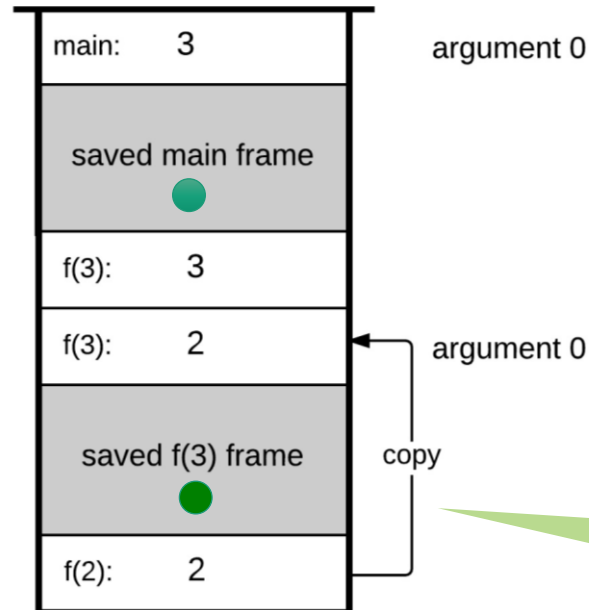
label BASECASE

```
push constant 1
return
```

function mult 2

```
// Code omitted
```

global stack



handle the
return value

Run-time example

function main 0

```
push constant 3  
call factorial 1  
return
```

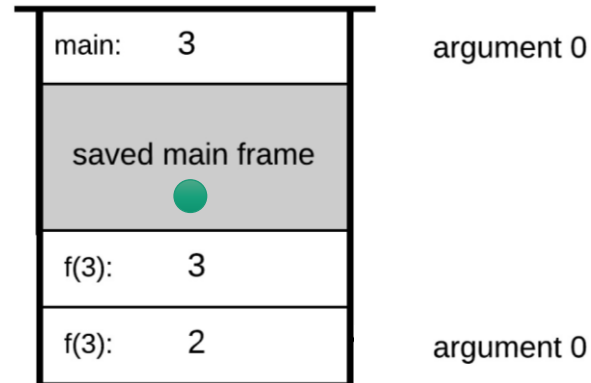
function factorial 1

```
push argument 0  
push constant 1  
eq  
if-goto BASECASE  
push argument 0  
push argument 0  
push constant 1  
sub  
call factorial 1  
call mult 2  
return  
label BASECASE  
push constant 1  
return
```

function mult 2

```
// Code omitted
```

global stack



impact on the global
stack not shown
(except for end result)

Run-time example

function main 0

push constant 3
call factorial 1
return

function factorial 1

push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1

call mult 2

return

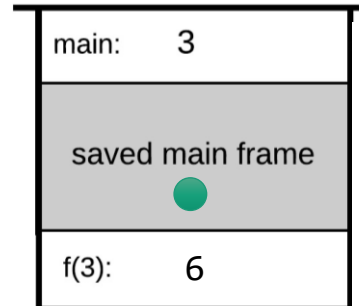
label BASECASE

push constant 1
return

function mult 2

// Code omitted

global stack



argument 0

Run-time example

function main 0

push constant 3
call factorial 1
return

function factorial 1

push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2

return

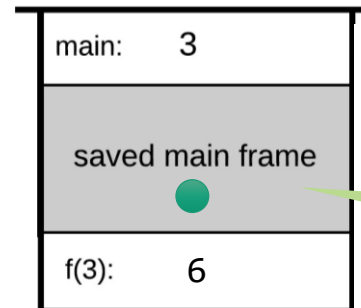
label BASECASE

push constant 1
return

function mult 2

// Code omitted

global stack



argument 0

get return
address

Run-time example

function main 0

push constant 3
call factorial 1
return

function factorial 1

push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2

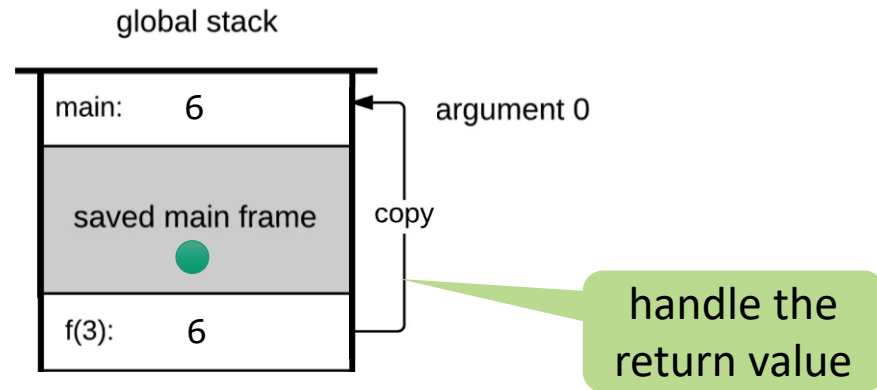
return

label BASECASE

push constant 1
return

function mult 2

// Code omitted



Run-time example

global stack

main:	6
-------	---

function main 0

push constant 3
call factorial 1
return

function factorial 1

push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2

return

label BASECASE

push constant 1
return

function mult 2

// Code omitted

Run-time example

global stack

main:	6
-------	---

function main 0

```
push constant 3
call factorial 1
return
```

function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return
label BASECASE
push constant 1
return
```

function mult 2

```
// Code omitted
```

Recap

function main 0

```
push constant 3
call factorial 1
return
```

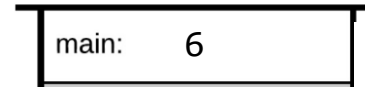
function factorial 1

```
push argument 0
push constant 1
eq
if-goto BASECASE
push argument 0
push argument 0
push constant 1
sub
call factorial 1
call mult 2
return
label BASECASE
push constant 1
return
```

function mult 2

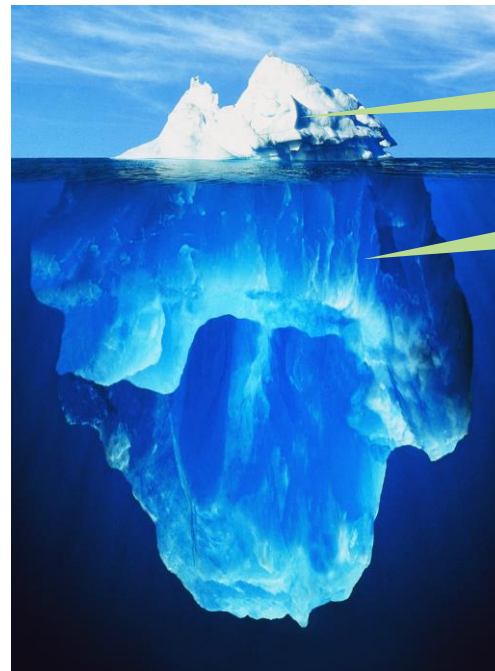
```
// Code omitted
```

global stack



The caller (main function) wanted to compute 3!

- it pushed 3, called factorial, and got 6
- from the caller's view, nothing exciting happened...



abstraction

implementation

Detailed implementation of function

VM code (arbitrary example)

caller

```
function Foo.main 4
...
// computes  $-(19 * (\text{local } 3))$ 
push constant 19
push local 3
call Bar.mult 2
neg
...
```

callee

```
function Bar.mult 2
// Computes the product of
// the first two
// arguments and puts the
// result in local 1
...
push local 1 // return value
return
```

We focus on VM function commands:

- call *functionName nArgs*
- function *functionName nVars*
- return

Contract: the caller's view

VM code (arbitrary example)

caller

```
function Foo.main 4
...
// computes -(19 * (local 3))
push constant 19
push local 3
call Bar.mult 2
neg
...

function Bar.mult 2
// Computes the product of
// the first two
// arguments and puts the
// result in local 1
...
push local 1 // return value
return
```

- Before calling another function, push as many **arguments** as the function expects to get.
- Next, **invoke** the function using
`call functionName nArgs`
- After the called **function returns**,
 - the **argument** values being pushed before the call disappear from the stack, and a *return value* (that always exists) appears at the top of the stack;
 - all my **memory segments** are exactly the same as they were before the call (except that temp is undefined and some values of my static segment may have changed).

Contract: the caller's view

VM code (arbitrary example)

caller

```
function Foo.main 4
...
// computes -(19 * (local 3))
push constant 19
push local 3
call Bar.mult 2
neg
...

function Bar.mult 2
// Computes the product of
// the first two
// arguments and puts the
// result in local 1
...
push local 1 // return value
return
```

- Before calling another function, push as many **arguments** as the function expects to get.
- Next, **invoke** the function using
`call functionName nArgs`
- After the called **function returns**,
 - the **argument** values being pushed before the call disappear from the stack, and a *return value* (that always exists) appears at the top of the stack;
 - all my **memory segments** are exactly the same as they were before the call (except that temp is undefined and some values of my static segment may have changed).

blue: must be handled by the VM implementation

Contract: the callee's view

VM code (arbitrary example)

```
function Foo.main 4
...
// computes  $-(19 * (\text{local } 3))$ 
push constant 19
push local 3
call Bar.mult 2
neg
...

function Bar.mult 2
// Computes the product of
// the first two
// arguments and puts the
// result in local 1
...
push local 1 // return value
return
```

callee

- Before start executing, my **argument** segment has been initialized with the argument values passed by the caller,
- My **local** variables segment has been allocated and initialized to zeros,
- My **static** segment has been set to the static segment of the VM file to which I belong, (memory segments this, that, pointer, and temp are undefined upon entry)
- My **stack** is empty,
- Before **returning**, I must push a value onto the stack.

Contract: the callee's view

VM code (arbitrary example)

```
function Foo.main 4
...
// computes  $-(19 * (\text{local } 3))$ 
push constant 19
push local 3
call Bar.mult 2
neg
...

function Bar.mult 2
// Computes the product of
// the first two
// arguments and puts the
// result in local 1
...
push local 1 // return value
return
```

callee

- Before start executing, my argument segment has been initialized with the argument values passed by the caller
- My local variables segment has been allocated and initialized to zeros
- My static segment has been set to the static segment of the VM file to which I belong (memory segments this, that, pointer, and temp are undefined upon entry)
- My stack is empty
- Before returning, I must push a value onto the stack.

blue: must be handled by the VM implementation

The VM implementation view

VM code

```
function Foo.main 4
...
// computes -(19 * (local 3))
push constant 19
push local 3
call Bar.mult 2
neg
...

function Bar.mult 2
// Computes the product of
// the first two
// arguments and puts the
// result in local 1
...
push local 1 // return value
return
```

VM translator



Generated assembly code

```
(Foo.main)           // created and plugged by the translator
// assembly code that handles the initialization of the
// function's execution
...
// assembly code that handles push constant 19
// assembly code that handles push local 3
// assembly code that saves the caller's state on the stack,
// sets up for the function call, and then:
goto Bar.mult        // (in assembly)
(Foo$ret.1)           // created and plugged by the translator
// assembly code that handles neg
...
(Bar.mult)            // created and plugged by the translator
// assembly code that handles the initialization of the
// function's execution
...
// assembly code that handles push local 1
// Assembly code that gets the return address (which happens
// to be Foo$ret.1) off the stack, copies the return value to
// the caller, reinstates the caller's state, and then:
goto Foo$ret.1        // (in assembly)
```

psuedocode!

Outlines

- Introduction to virtual machine
- VM abstraction
- VM implementation
- VM translator

Implementation

Proposed design:

- Parser: parses each VM command into its lexical elements,
- CodeWriter: writes the assembly code that implements the parsed command,
- Main: drives the process (VMTranslator).

Main (VMTranslator)

- Input: *fileName.vm*,
- Output: *fileName.asm*.

Main logic:

- Construct a Parser to handle the input file,
- Construct a CodeWriter to handle the output file,
- March through the input file, parsing each line and generating code from it.

Parser

- Handle the parsing of a single .vm file,
- Read a VM command, parse the command into its lexical components, and provide convenient access to these components,
- Ignore all white space and comments.

Routine	Arguments	Returns	Function
Constructor	Input file / stream	—	Opens the input file/stream and gets ready to parse it.
hasMoreCommands	—	Boolean	Are there more commands in the input?
advance	—	—	Reads the next command from the input and makes it the <i>current command</i> . Should be called only if hasMoreCommands() is true. Initially there is no current command.

Parser

- Handle the parsing of a single .vm file,
- Read a VM command, parse the command into its lexical components, and provide convenient access to these components,
- Ignore all white space and comments.

Routine	Arguments	Returns	Function
commandType	—	C_ARITHMETIC, C_PUSH, C_POP, C_LABEL, C_GOTO, C_IF, C_FUNCTION, C_RETURN, C_CALL	Returns a constant representing the type of the current command. C_ARITHMETIC is returned for all the arithmetic/logical commands.
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.

CodeWriter

Generates assembly code from the parsed VM command:

Routine	Arguments	Returns	Function
Constructor	Output file / stream	—	Opens the output file / stream and gets ready to write into it.
writeArithmetic	command (string)	—	Writes to the output file the assembly code that implements the given arithmetic command.
WritePushPop	command (C_PUSH or C_POP), segment (string), index (int)	—	Writes to the output file the assembly code that implements the given command, where command is either C_PUSH or C_POP.
Close	—	—	Closes the output file.

Booting

VM program convention

- One file in any **VM program** is expected to be named `Main.vm`;
- One **VM function** in this file is expected to be named `main`.

VM implementation conventions

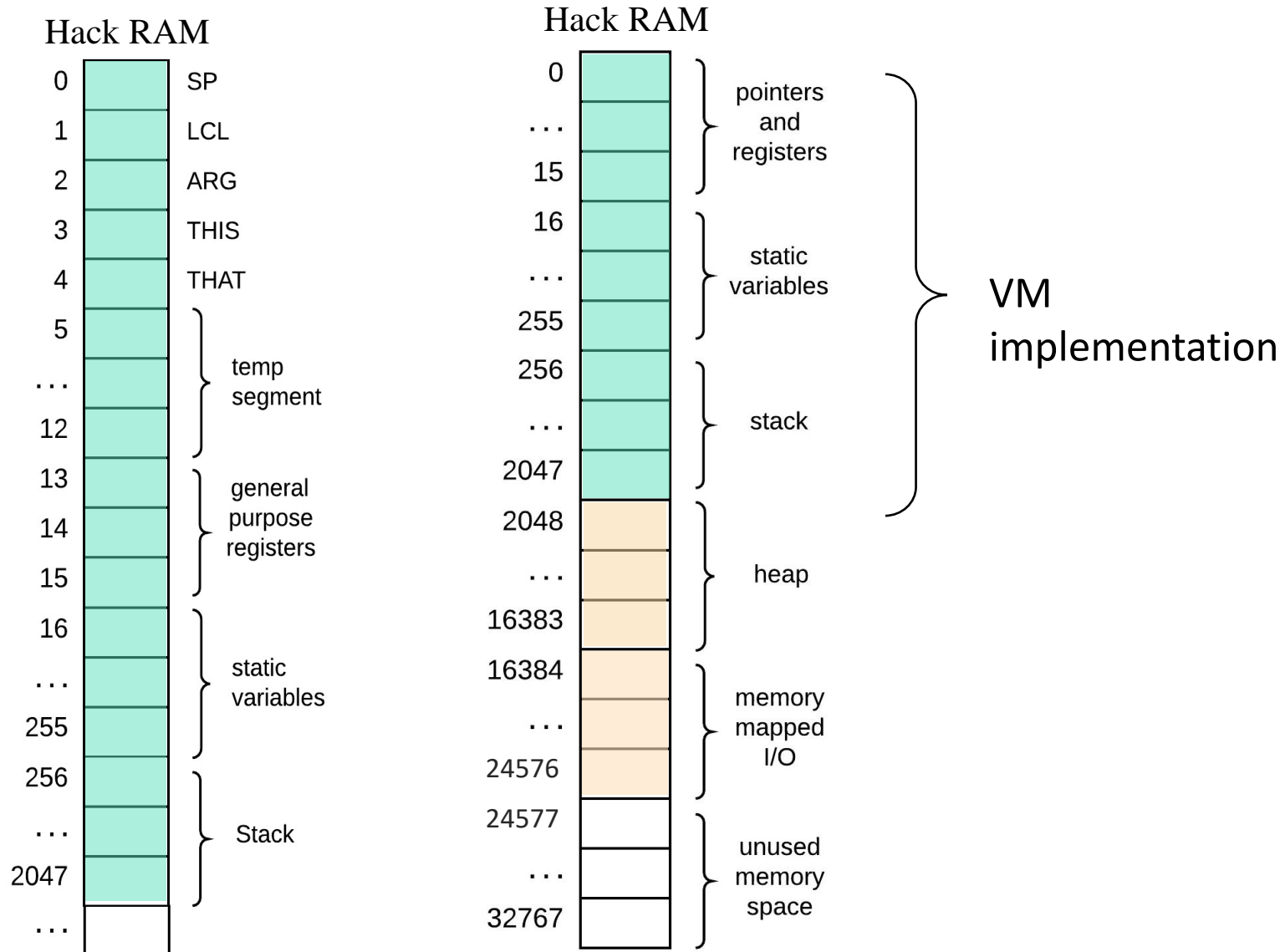
- The stack starts in address 256 in the host RAM,
- When the VM implementation starts running, or is reset, it starts executing an argument-less OS function named **Sys.init**,
- **Sys.init** is designed to call **Main.main**, and then enter an infinite loop.

These conventions are realized by the following code:

```
// Bootstrap code (should be  
// written in assembly)  
SP = 256  
call Sys.init
```

In the Hack platform, this code should be put in the ROM, starting at address 0

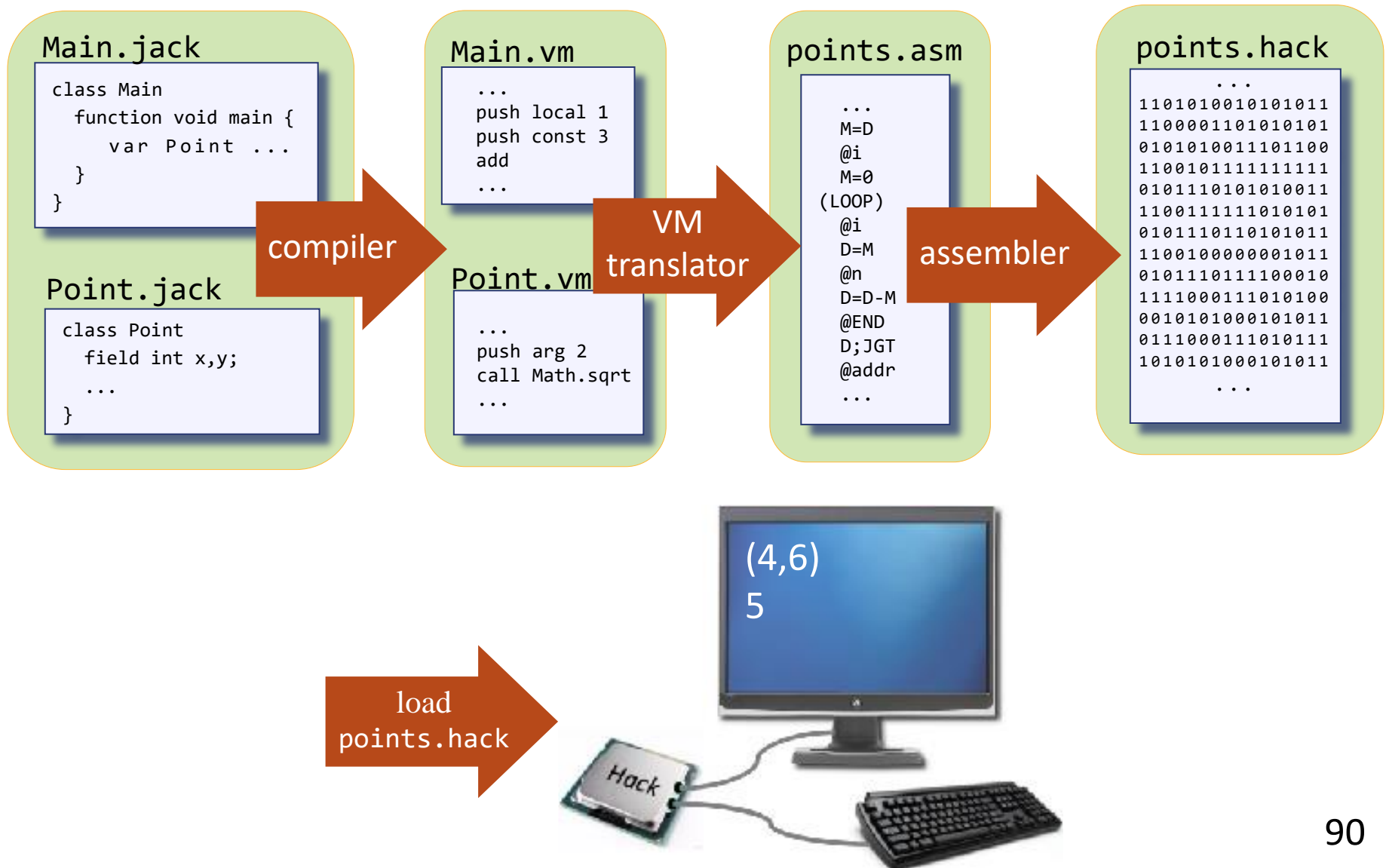
Standard mapping of the VM on the Hack platform



Special symbols in translated VM programs

<i>Symbol</i>	<i>Usage</i>
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 addresses within the host RAM of the virtual segments <code>local</code> , <code>argument</code> , <code>this</code> , and <code>that</code> of the currently running VM function.
R13–R15	These predefined symbols can be used for any purpose.
Xxx.i symbols	Each static variable <i>i</i> in file Xxx.vm is translated into the assembly symbol Xxx.j., where <i>j</i> is incremented each time a new static variable is encountered in the file Xxx.vm. In the subsequent assembly process, these symbolic variables will be allocated to the RAM by the Hack assembler.
functionName\$label	<p>Let foo be a function within a VM file Xxx. Each <code>label bar</code> command within foo should generate and insert into the assembly code stream a symbol Xxx.foo\$bar.</p> <p>When translating <code>goto bar</code> and <code>if-goto bar</code> commands (within foo) into assembly, the full label specification Xxx.foo\$bar must be used instead of bar.</p>
functionName	Each <code>function foo</code> command within a VM file Xxx should generate and insert 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 will translate this symbol into the physical memory address where the function code starts.
functionName\$ret.i	<p>Let foo be a function within a VM file Xxx.</p> <p>Within foo, each <code>function call</code> command should generate and insert into the assembly code stream a symbol Xxx.foo\$ret.i, where <i>i</i> is a running integer (one such symbol should be generated for each <code>call</code> command within foo).</p> <p>This symbol serves as the return address to the calling function. In the subsequent assembly process, the assembler will translate this symbol into the physical memory address of the command immediately after the function call command.</p>

Big picture



Summary

- VM bridges the high-level programming language and the machine code.
- VM is implemented using stack
 - Arithmetic/logic operation
 - memory segment
 - branching
 - function
- VM translator: VM code to assembly code
 - Stack
 - memory segment
 - Branching
 - function

Acknowledgement

- This set of lecture notes are based on the lecture notes provided by Noam Nisam / Shimon Schocken.
- You may find more information on:
www.nand2tetris.org.