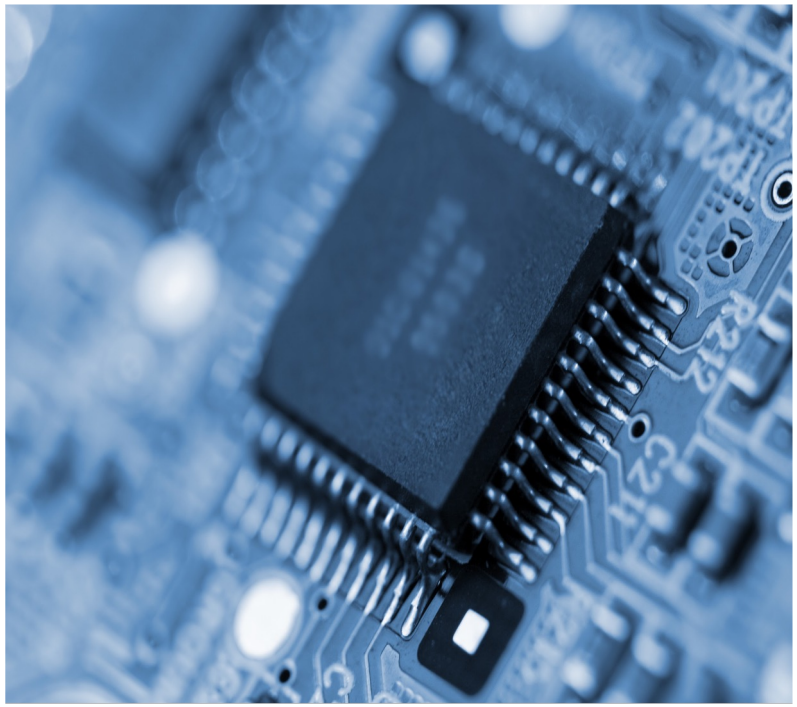
UNIVERSITY OF LEEDS

Computer Processors

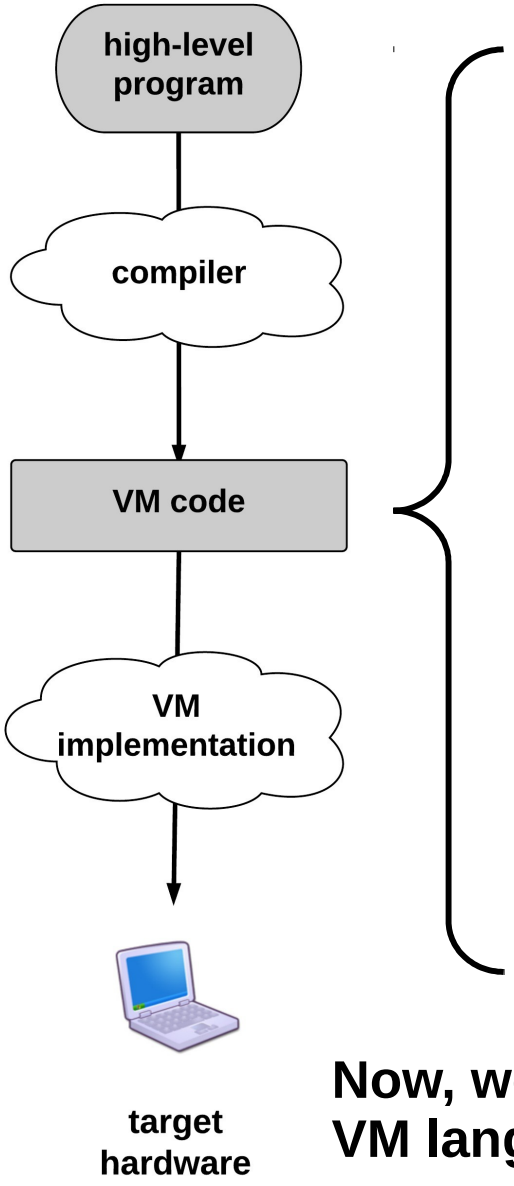
Virtual Machines:
Translation to Assembly Language



This lecture is based on the excellent course *Nand to Tetris* by Noam Nisam and Shimon Schocken, and we reuse here many of the slides provided at www.nand2tetris.org

VM

UNIVERSITY OF LEEDS



Stack machine, manipulated by:

- Arithmetic / logical commands
  → Definition
  → Implementation
- Memory segment commands
  → Definition
  → Implementation
- Branching commands
  → Definition
  → Implementation
- Function commands
  → Definition
  → Implementation

Now, we must translate VM-code (and even better understand VM language) to obtain the desired assembly language

VM

UNIVERSITY OF LEEDS

- We start by defining a “standard mapping” from VM elements and operations to the Hack hardware and machine language.
  In what follows, we will refer to this software using the terms VM implementation or VM translator interchangeably.

VM

UNIVERSITY OF LEEDS

- We start by defining a ‘‘standard mapping’’ from VM elements and operations to the Hack hardware and machine language.
  In what follows, we will refer to this software using the terms *VM implementation* or *VM translator* interchangeably.

- The VM specification given so far made no assumption about the architecture on which the VM can be implemented. When it comes to virtual machines, this platform independence is the whole point: You don’t want to commit to any one hardware platform, since you want your machine to potentially run on all of them, including those that were not built yet.

VM

UNIVERSITY OF LEEDS

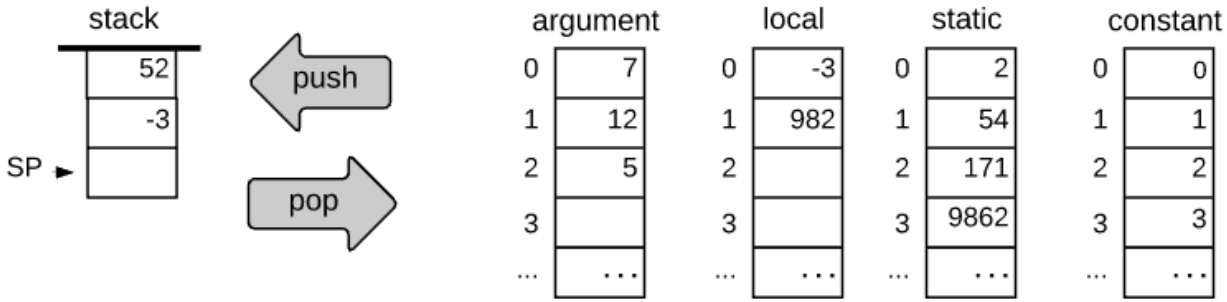
- We start by defining a “standard mapping” from VM elements and operations to the Hack hardware and machine language.
  In what follows, we will refer to this software using the terms VM implementation or VM translator interchangeably.

- The VM specification given so far made no assumption about the architecture on which the VM can be implemented. When it comes to virtual machines, this platform independence is the whole point: You don’t want to commit to any one hardware platform, since you want your machine to potentially run on all of them, including those that were not built yet.

- However, it is usually recommended that some guidelines are provided as to how the VM should map on the target platform, rather than leaving these decisions completely to the implementer’s discretion.
  These guidelines are called *standard mapping* and also used here

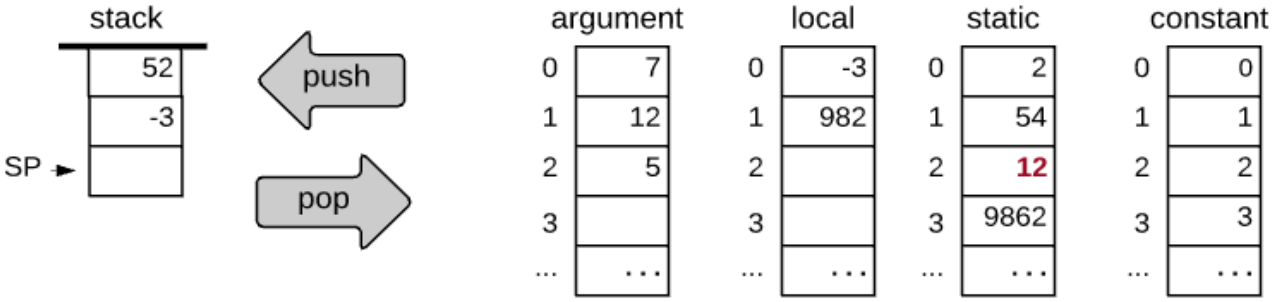
VM

UNIVERSITY OF LEEDS




```
let  static 2 = argument 1
```

```
push argument 1

pop static 2
```



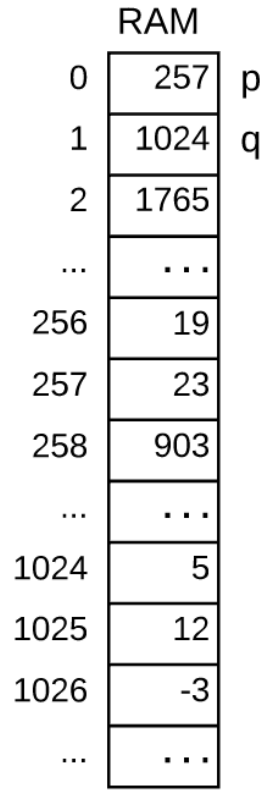
In this example four memory segments (in total there are eight),

- We have to somehow map these memory segments on the single RAM that is available for us
- We have to remember where each segment starts on the RAM and manage these locations.

Once we have this mapping, we have to translate the commands (e.g. `push` and `pop`) into a sequence of machine language instructions.

In a nutshell: pointer manipulation

UNIVERSITY OF LEEDS

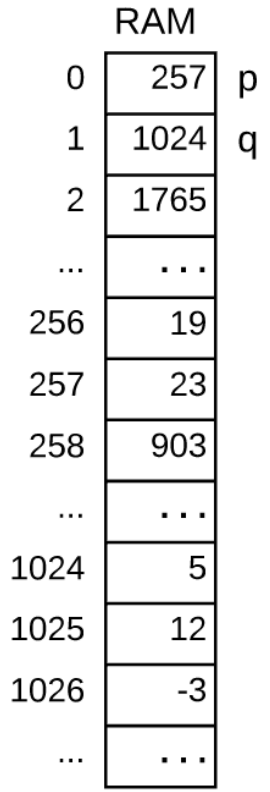


symbols p and q refer to RAM 0 and RAM 1, resp.

In a nutshell: pointer manipulation

UNIVERSITY OF LEEDS

```
D = *p              // D becomes 23
```



Notation:

*p // the memory location that p points at

If D = p, then D will get the value 257 (=content of RAM[0])
If D = *p then D gets value stored at address 257 (indirect addressing)

symbols p and q refer to RAM 0 and RAM 1, resp.

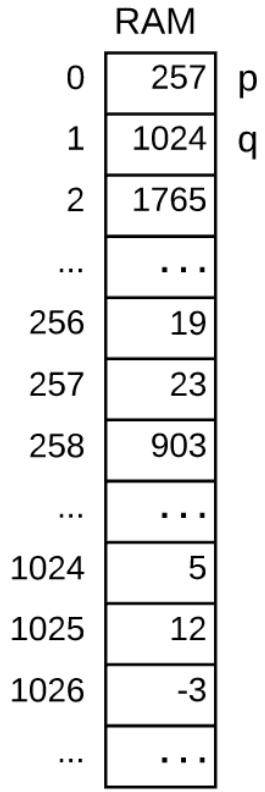
In a nutshell: pointer manipulation

UNIVERSITY OF LEEDS

```
D = *p              // D becomes 23
```

In Hack:

```
@ p
A=M
D=M
```



Notation:

*p // the memory location that p points at

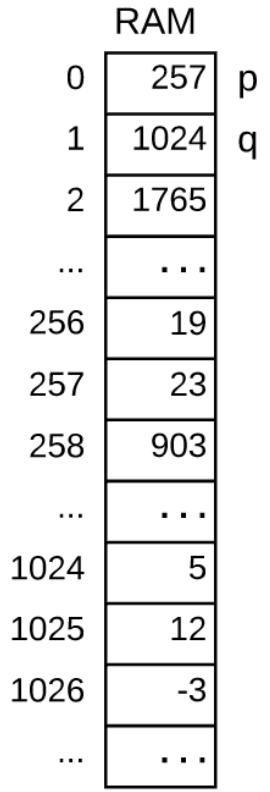
If D = p, then D will get the value 257 (=content of RAM[0])
If D = *p then D gets value stored at address 257 (indirect addressing)

symbols p and q refer to RAM 0 and RAM 1, resp.

In a nutshell: pointer manipulation

UNIVERSITY OF LEEDS

```
D = *p          // D becomes 23

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



symbols p and q refer to RAM 0 and RAM 1, resp.

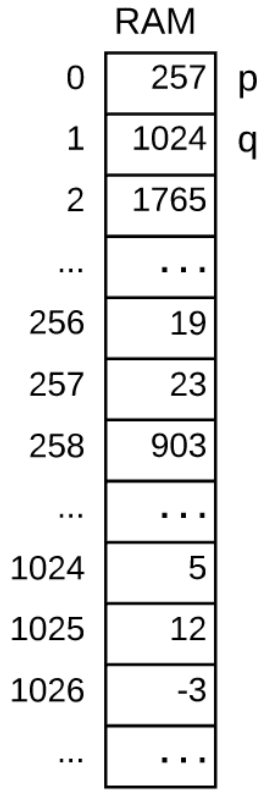
Notation:

```
*p          // the memory location that p points at

p --        // p is treated as normal variable and corresponds to value at
D =  *p     // RAM[0], thus, p=257 and p-- = p-1 results in 256, so RAM[0]
            // becomes 256
            // D =  *p means D gets value of RAM[256] and thus, 19

x --        //decrement x by 1

X++         //increment x by 1
```

In a nutshell: pointer manipulation

UNIVERSITY OF LEEDS

```
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
```



symbols p and q refer to RAM 0 and RAM 1, resp.

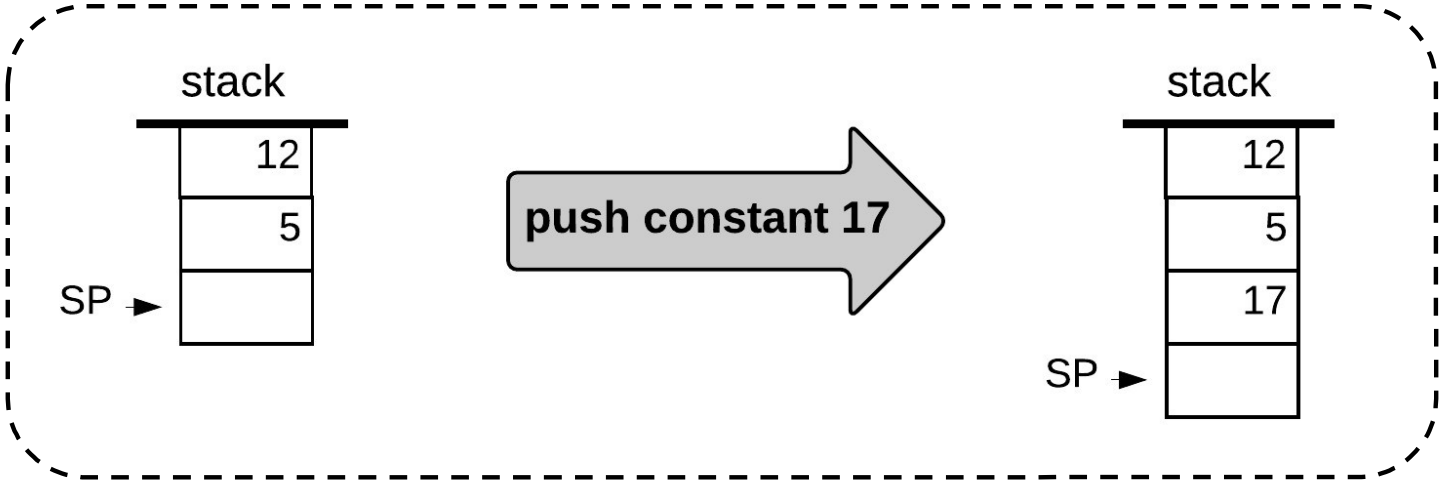
Notation:

```
*p          // the memory location that p points at
*q= 9       // q* points to adress at RAM[1] which is 1024 and
            // q*=9 means that RAM[1024] gets value 9
            // q++  means that RAM[1] gets value 1025

x --        //decrement x by 1

X++         //increment x by 1
```

Stack implementation

UNIVERSITY OF LEEDS

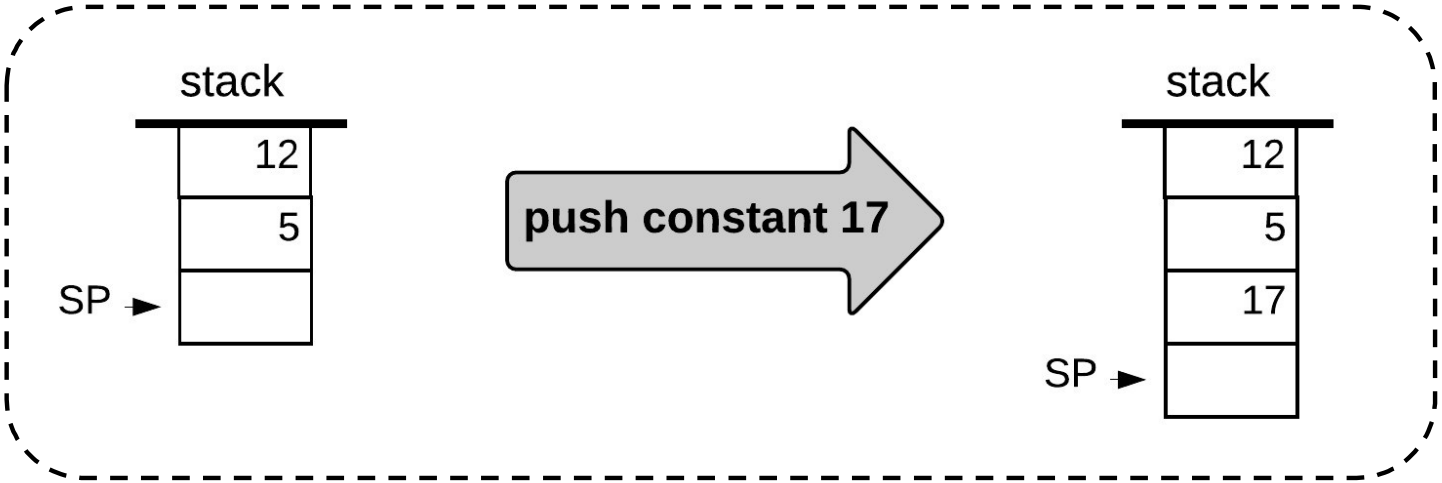
Abstraction:



Stack implementation

UNIVERSITY OF LEEDS

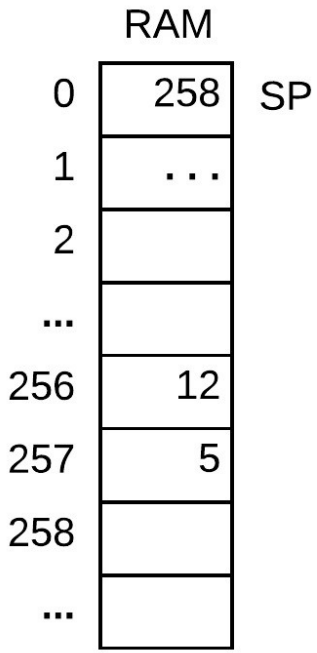
Abstraction:



Implementation:

Assumptions:

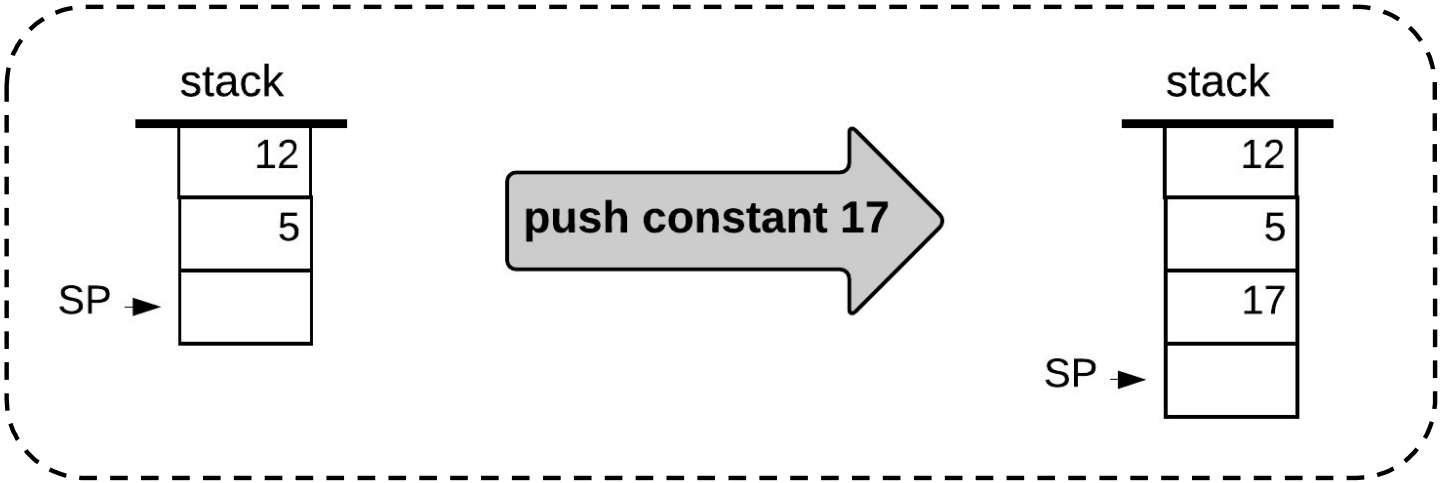
- `SP` stored in `RAM[0]`
- Stack base addr = `256`



Stack implementation

UNIVERSITY OF LEEDS

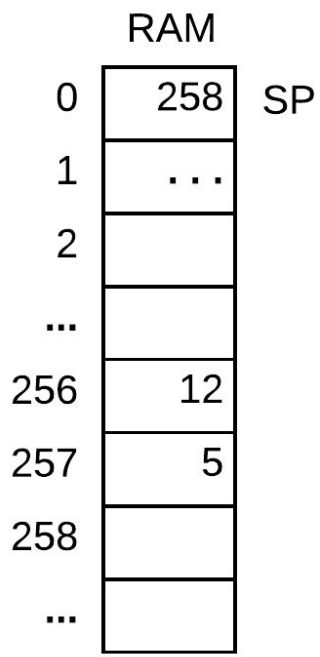
Abstraction:



Implementation:

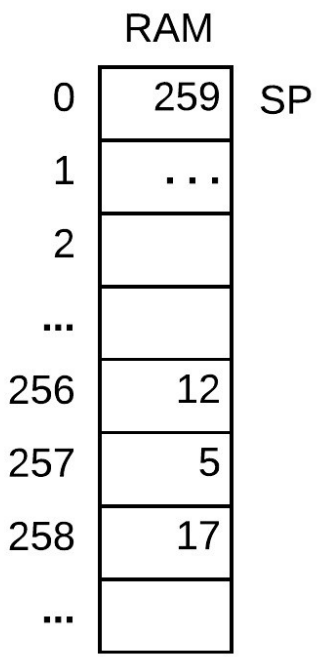
Assumptions:

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



Logic:

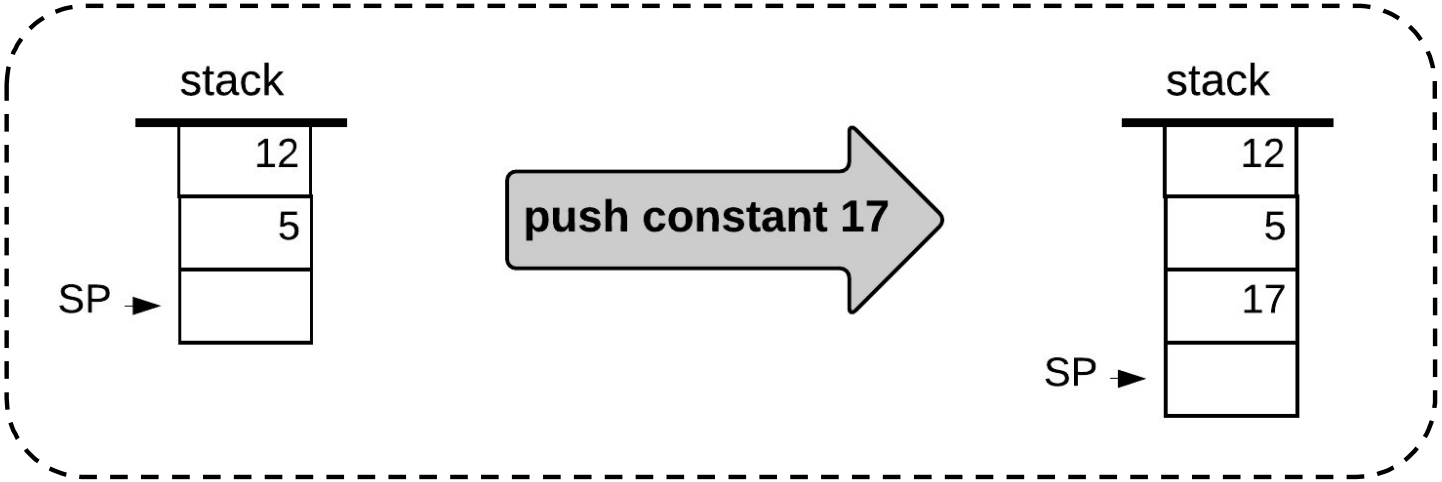
```
*SP = 17
SP++
```



Stack implementation

UNIVERSITY OF LEEDS

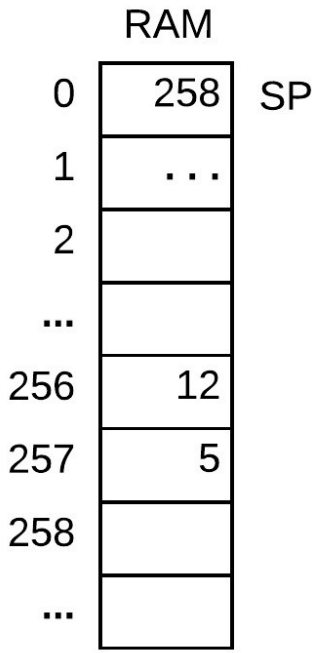
Abstraction:



Implementation:

Assumptions:

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

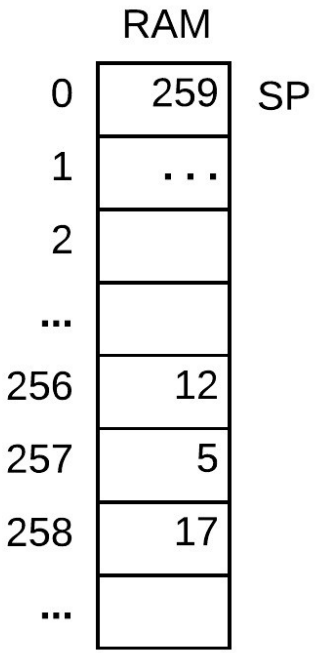


Logic:

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

Hack assembly:

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



Stack implementation

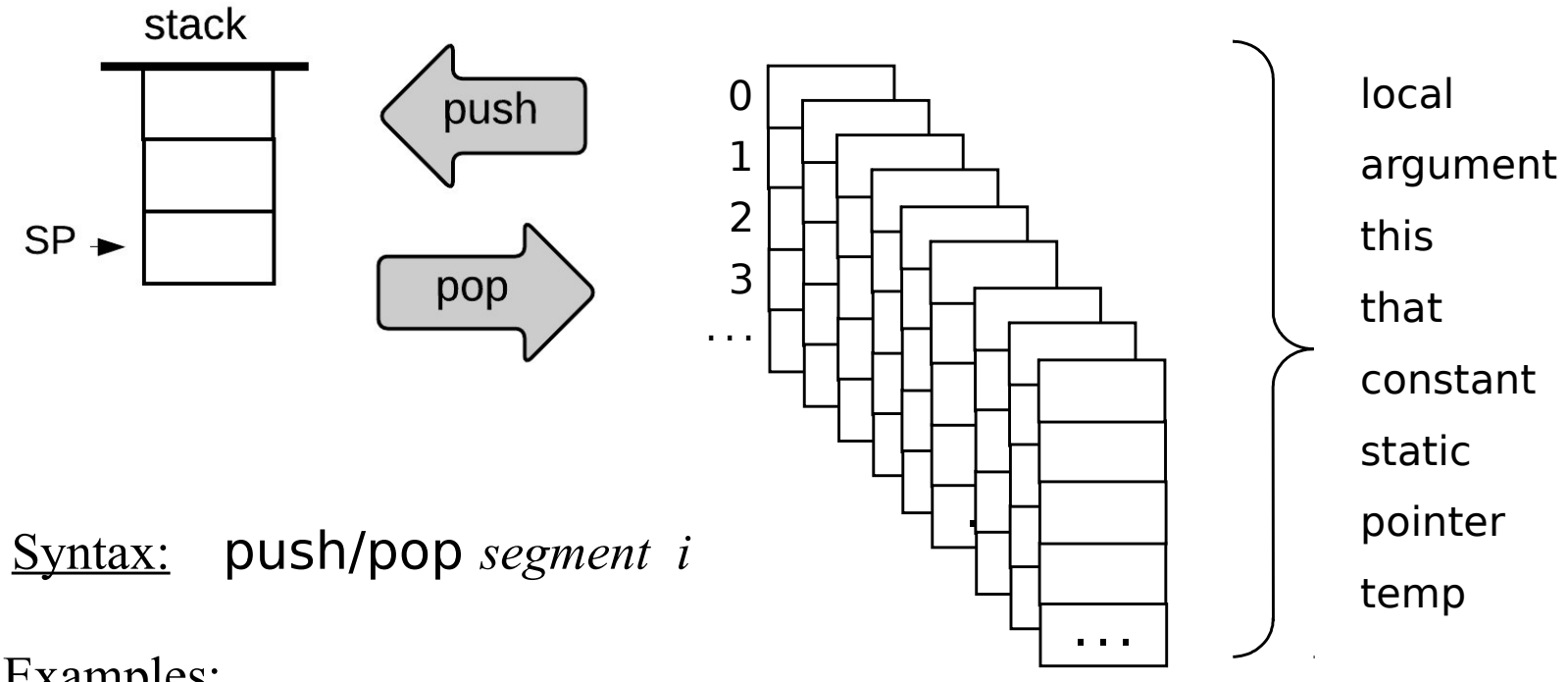
UNIVERSITY OF LEEDS

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 end up realizing the stack and the memory segments on the host platform.

Memory segments

UNIVERSITY OF LEEDS



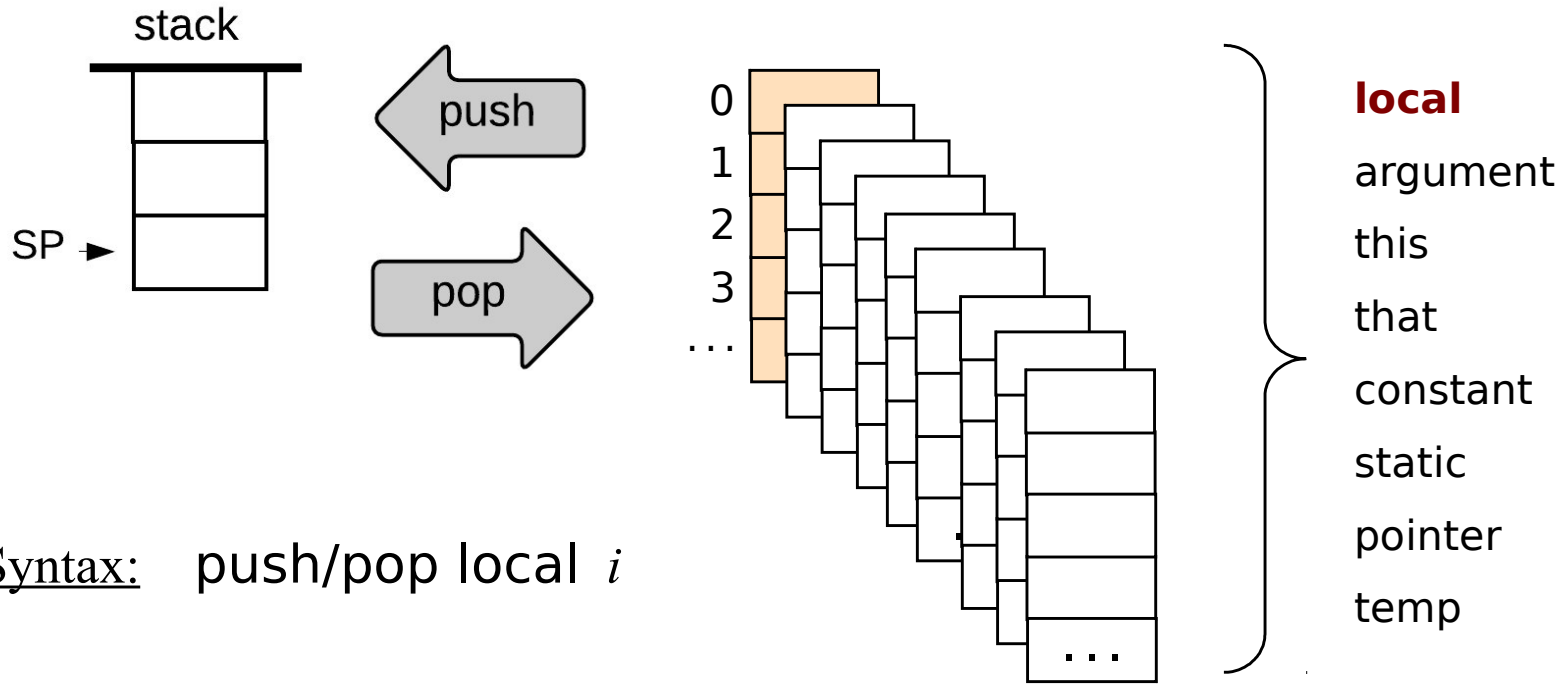
Syntax: push/pop *segment i*

Examples:

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

Implementing push/pop local *i*

UNIVERSITY OF LEEDS



Syntax: push/pop local *i*

Why do we need a local segment?

When the compiler translates high-level code into VM code...

- high-level operations on *local variables* are translated into VM operations on the entries of the segment local

Implementing pop local *i*

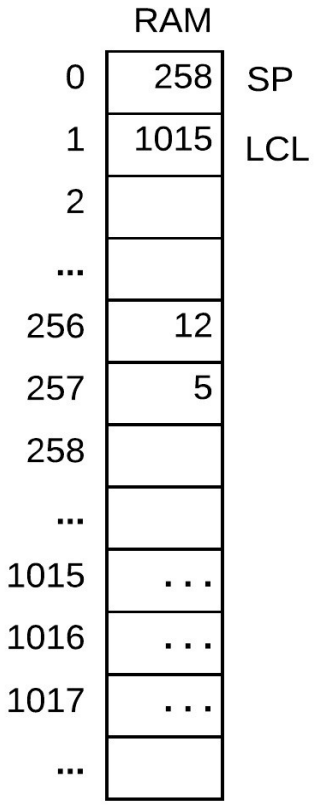
UNIVERSITY OF LEEDS

Abstraction:

Implementing pop local *i*

UNIVERSITY OF LEEDS

Abstraction:



Implementing pop local *i*

UNIVERSITY OF LEEDS

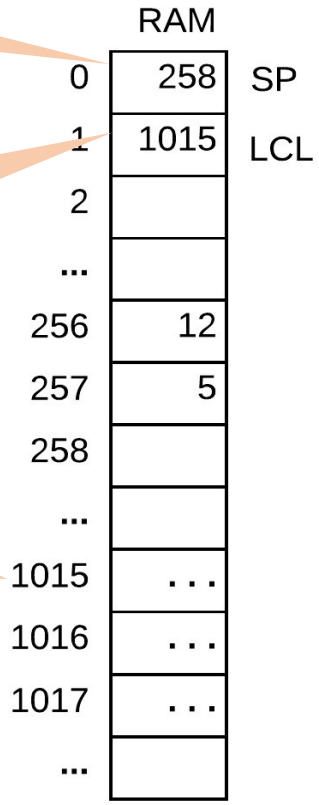
Abstraction:

Implementation:

stack pointer

base address of the local segment

the local segment is stored somewhere in the RAM



Implementing pop local *i*

UNIVERSITY OF LEEDS

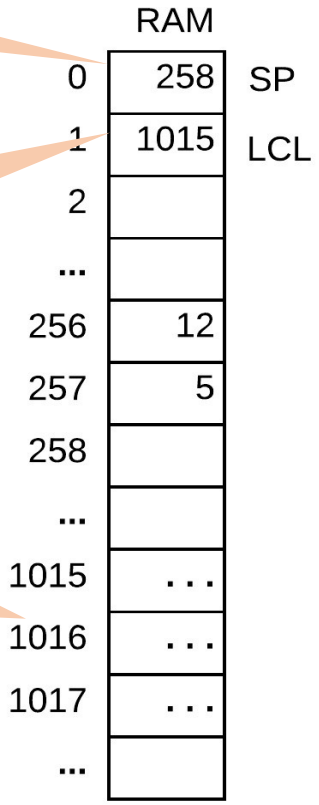
Abstraction:

Implementation:

stack pointer

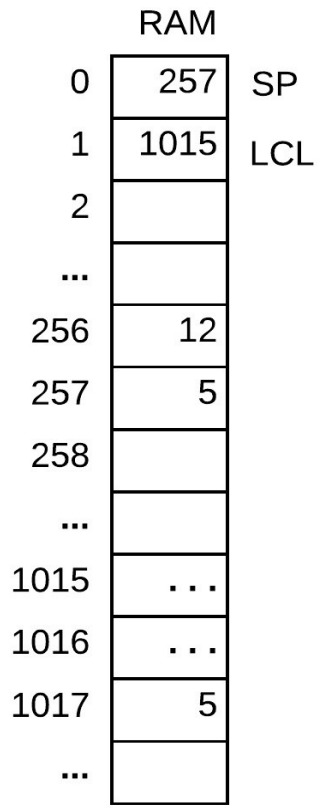
base address of the local segment

the local segment is stored some-where in the RAM



Implementation (here *i*=*2*):

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



Implementing pop local *i*

UNIVERSITY OF LEEDS

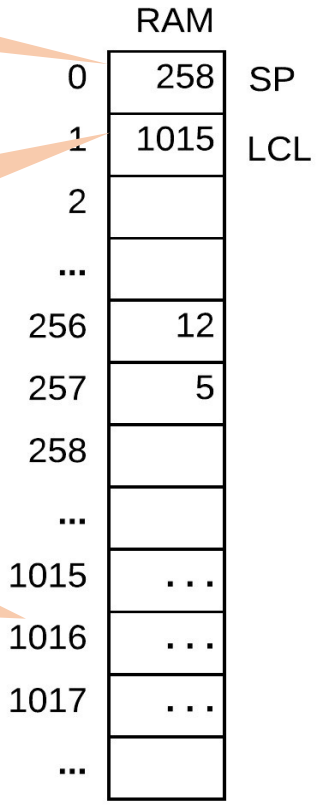
Abstraction:

Implementation:

stack pointer

base address of the local segment

the local segment is stored some-where in the RAM

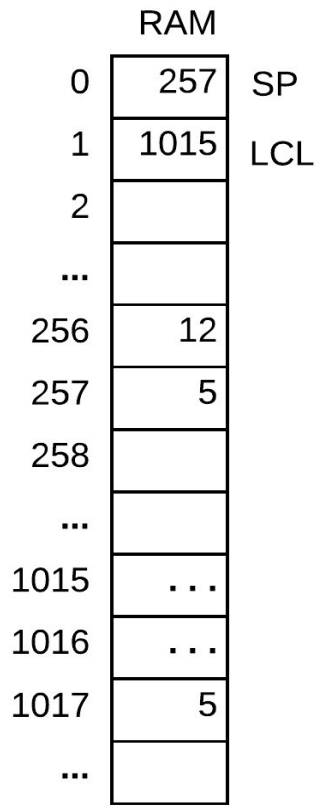


Implementation (here *i*=*2*):

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

Hack assembly:

Good Exercise!
(similar as before)



Implementing pop/push local *i*

UNIVERSITY OF LEEDS

VM code:

pop local *i*

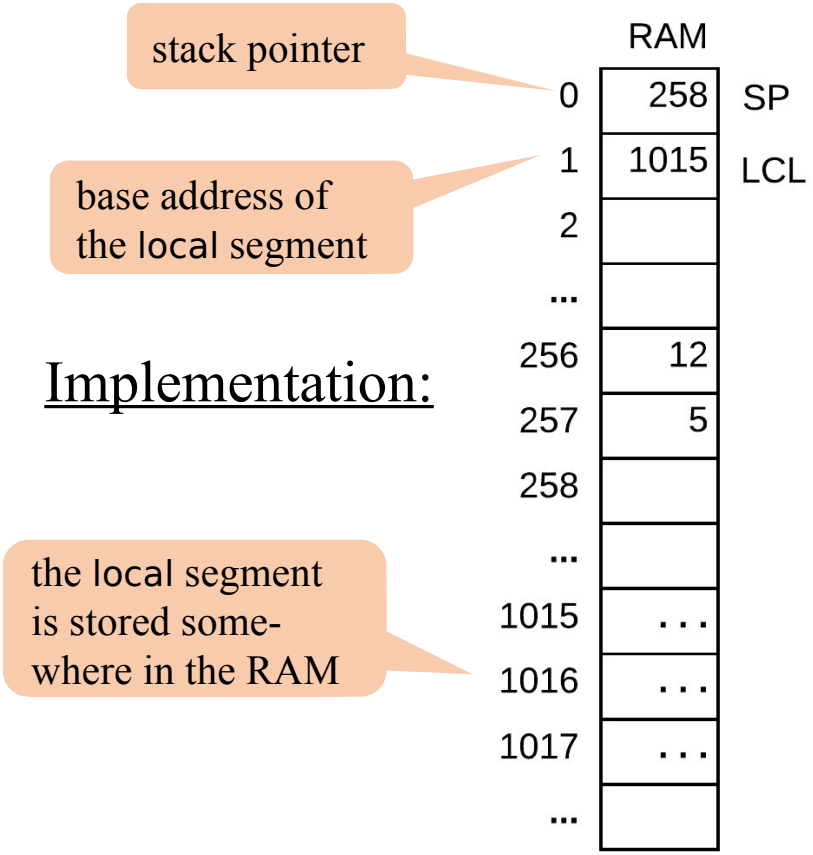
push local *i*

VM Translator

Assembly pseudo code:

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

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



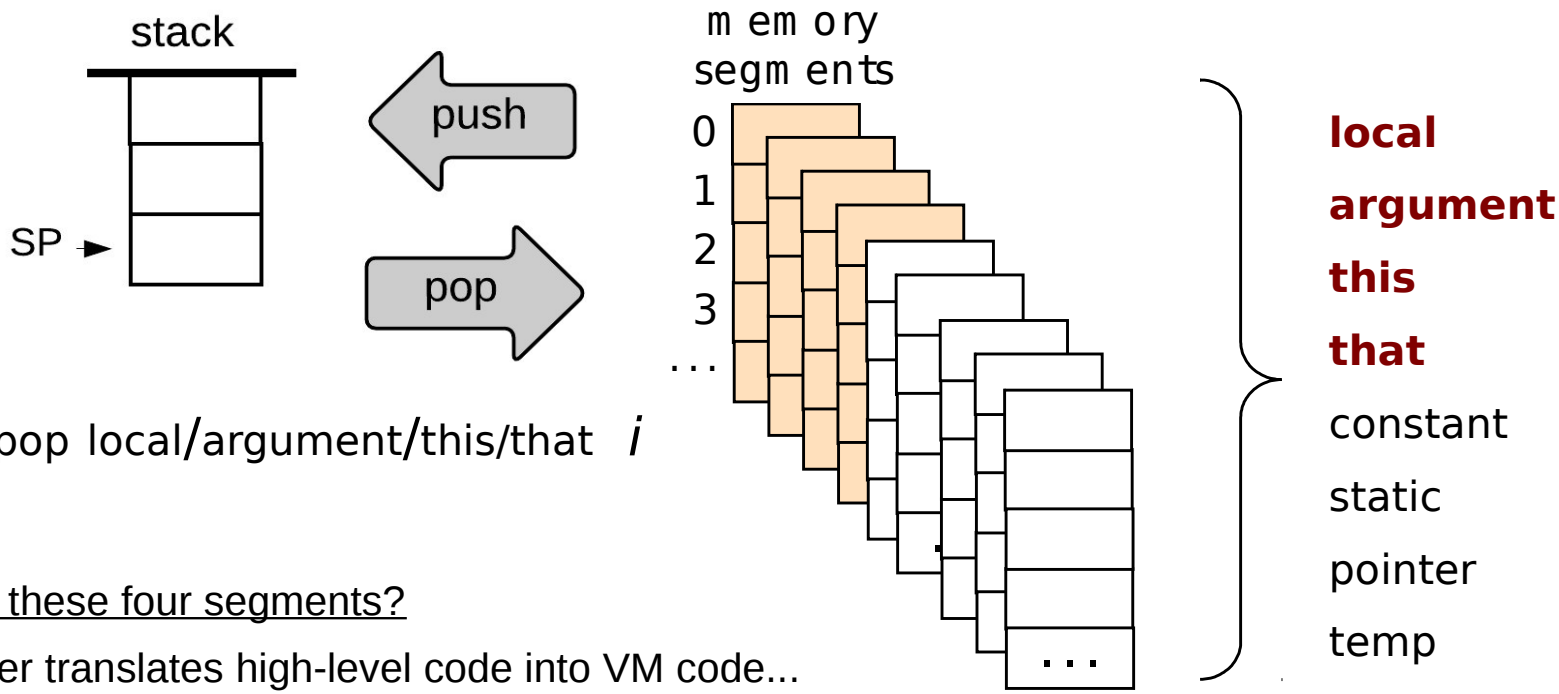
Implementation:

Hack assembly:

Good Exercise!
(similar as before)

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

UNIVERSITY OF LEEDS



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

Why do we need these four segments?

When the compiler translates high-level code into VM code...

- high-level operations on *local* variables are translated into VM operations on the entries of the segment local
- high-level operations on *argument* variables are translated into VM operations on the entries of the segment argument
- high-level operations on the *field* variables of the current object are translated into VM operations on the entries of the segment this
- high-level operations on *array entries* are translated into VM operations on the entries of the segment that

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

UNIVERSITY OF LEEDS

VM code:

`push` *segment i*

`pop` *segment i*

VM translator

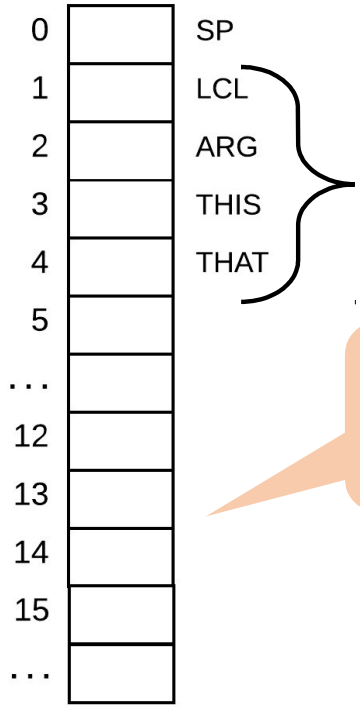
Assembly pseudo code:

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

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

segment = {`local, argument, this, that`}

Host RAM



base addresses of the four segments are stored in these pointers

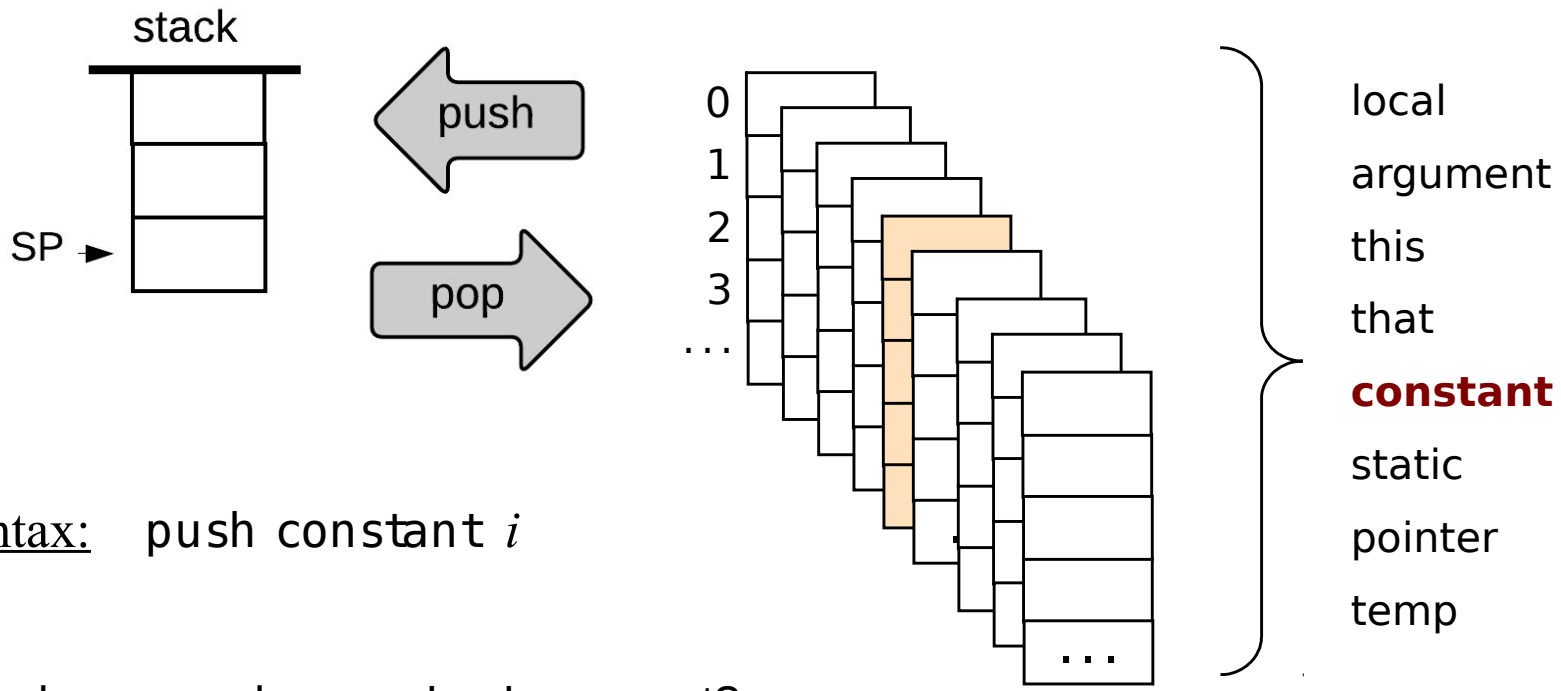
the four segments are stored somewhere in the RAM (*specified later*)

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

are implemented precisely the same way

Implementing push constant *i*

UNIVERSITY OF LEEDS



Syntax: `push constant i`

Why do we need a `constant` segment?

Because we need to represent constants somehow in the VM level.

When the compiler translates high-level code into VM code...

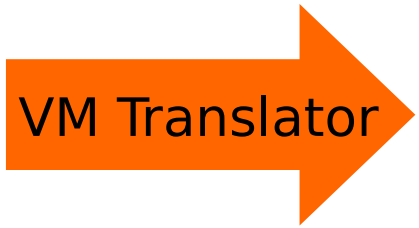
- high-level operations on the *constant i* are translated into VM operations on the segment entry `constant` *i*
- This syntactical convention will make sense when we write the compiler (not done in this course).

Implementing push constant *i*

UNIVERSITY OF LEEDS

VM code:

push constant *i*



Assembly psuedo code:

*SP = *i*, SP++

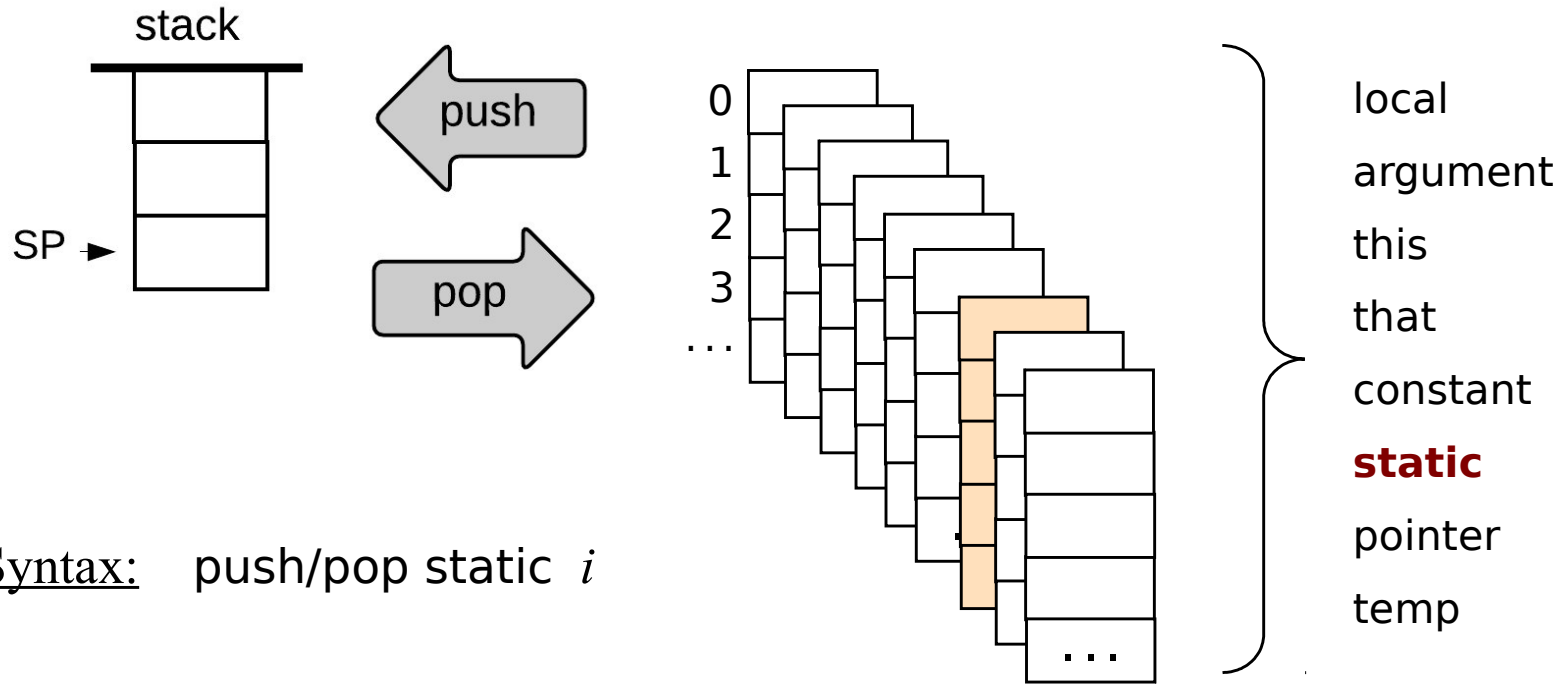
(no pop constant operation)

Implementation:

Supplies the specified constant

Implementing push/pop static *i*

UNIVERSITY OF LEEDS



Syntax: push/pop static *i*

Why do we need a static segment?

When translating high-level code into VM code, the compiler...

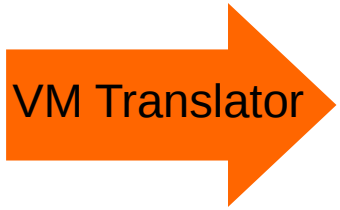
- high-level operations on *static* variables are translates into VM operations on entried of the segment static

Implementing push/pop static *i*

UNIVERSITY OF LEEDS

VM code:

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



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`

Implementing push/pop static *i*

UNIVERSITY OF LEEDS

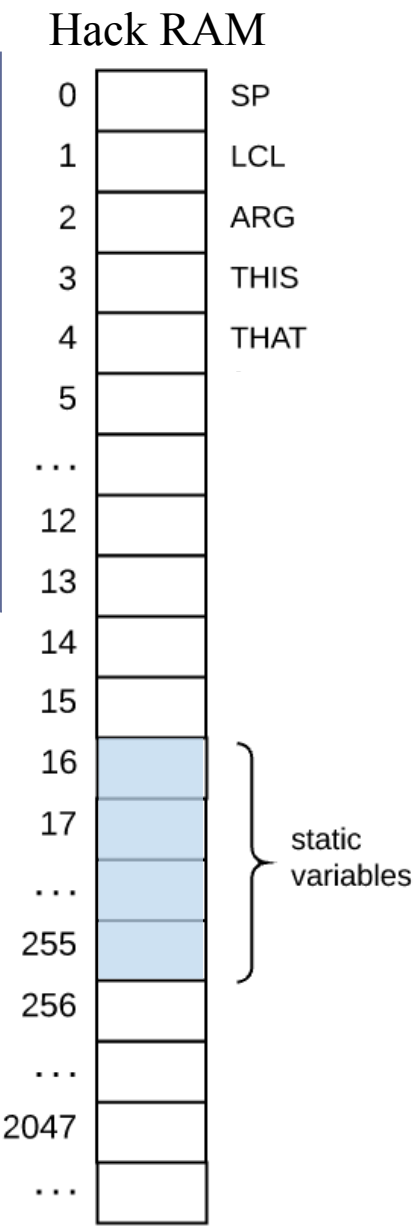
VM code:

```
// 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

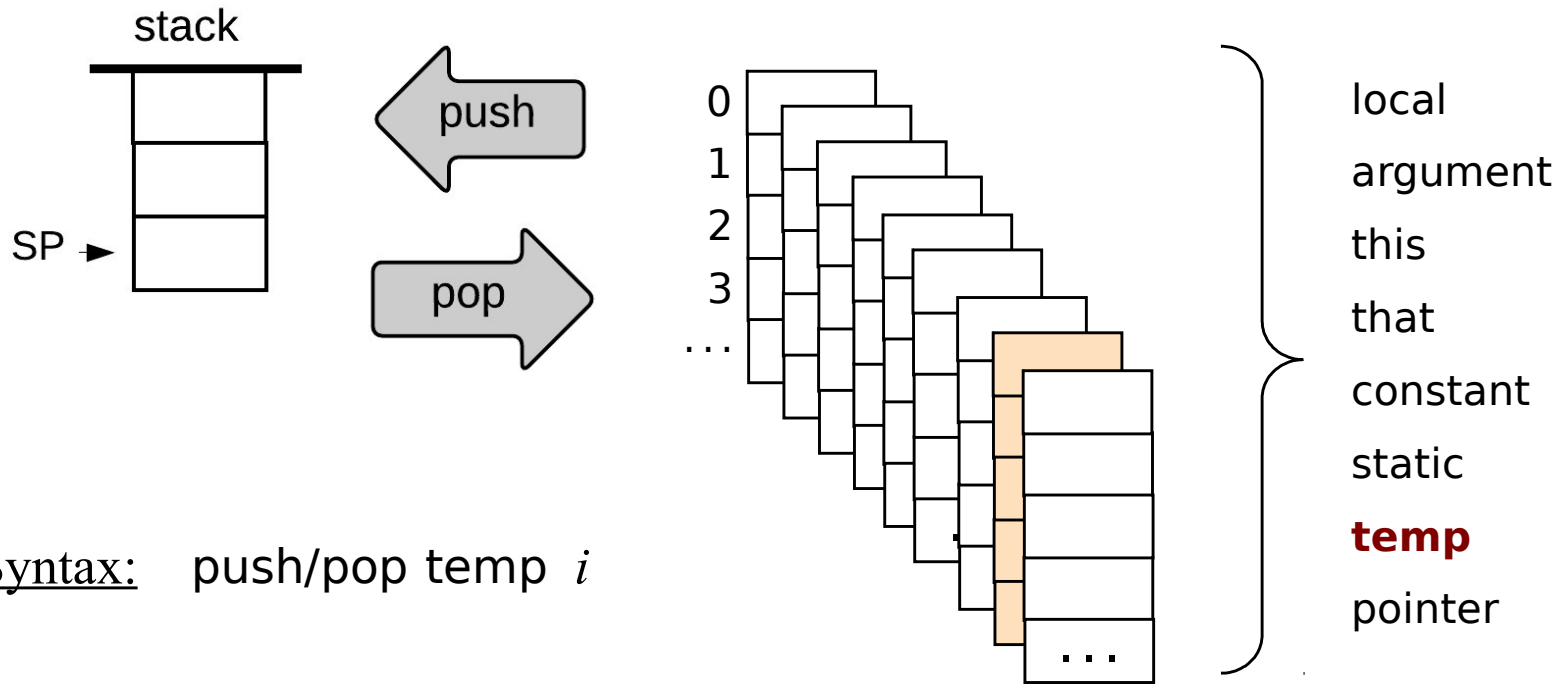
Solution:

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.

Implementing push/pop temp i

UNIVERSITY OF LEEDS



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.

Implementing push/pop temp *i*

UNIVERSITY OF LEEDS

VM code:

push temp *i*

pop temp *i*

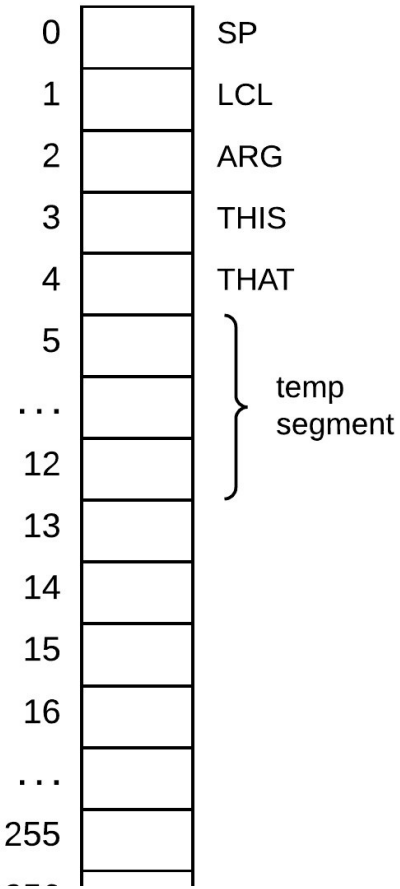
VM Translator

Assembly psuedo code:

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

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

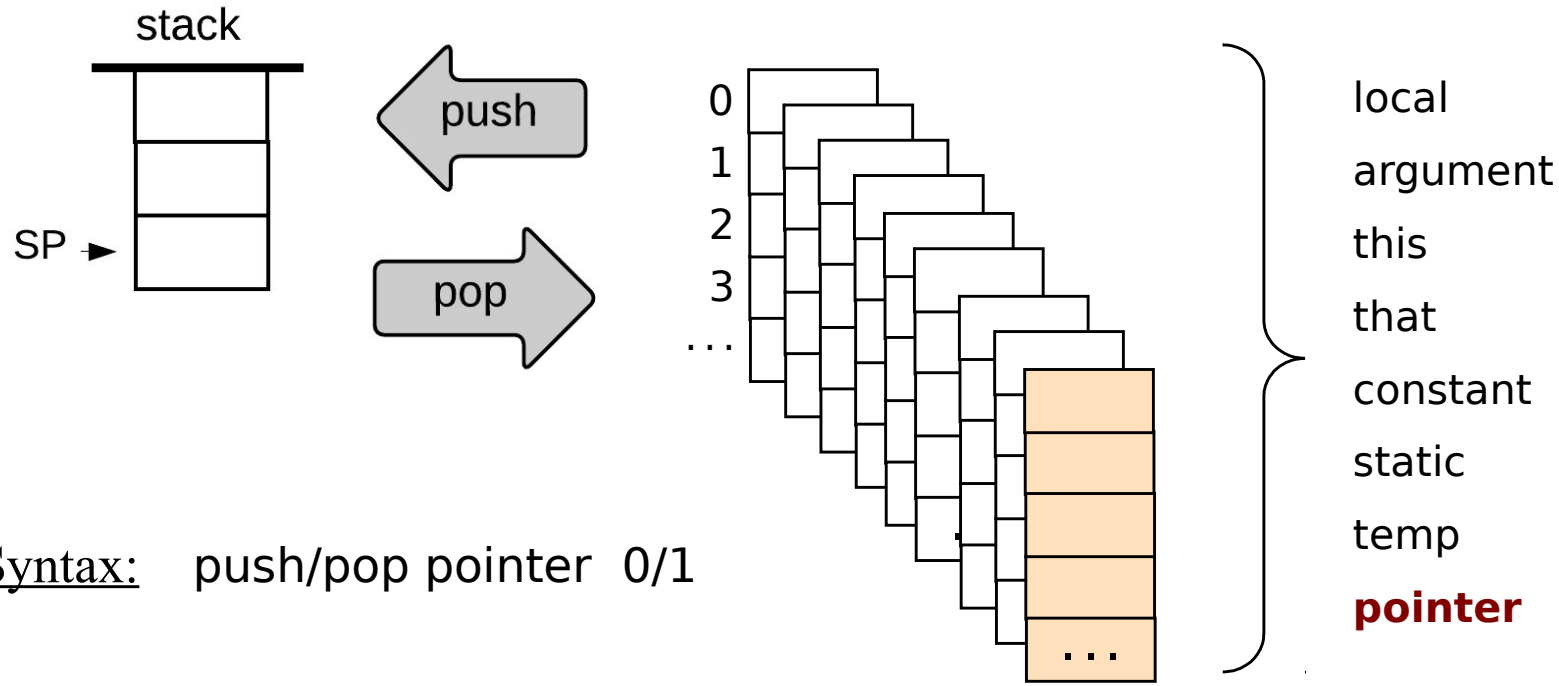
Hack RAM



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

Implementing push/pop pointer *0/1*

UNIVERSITY OF LEEDS



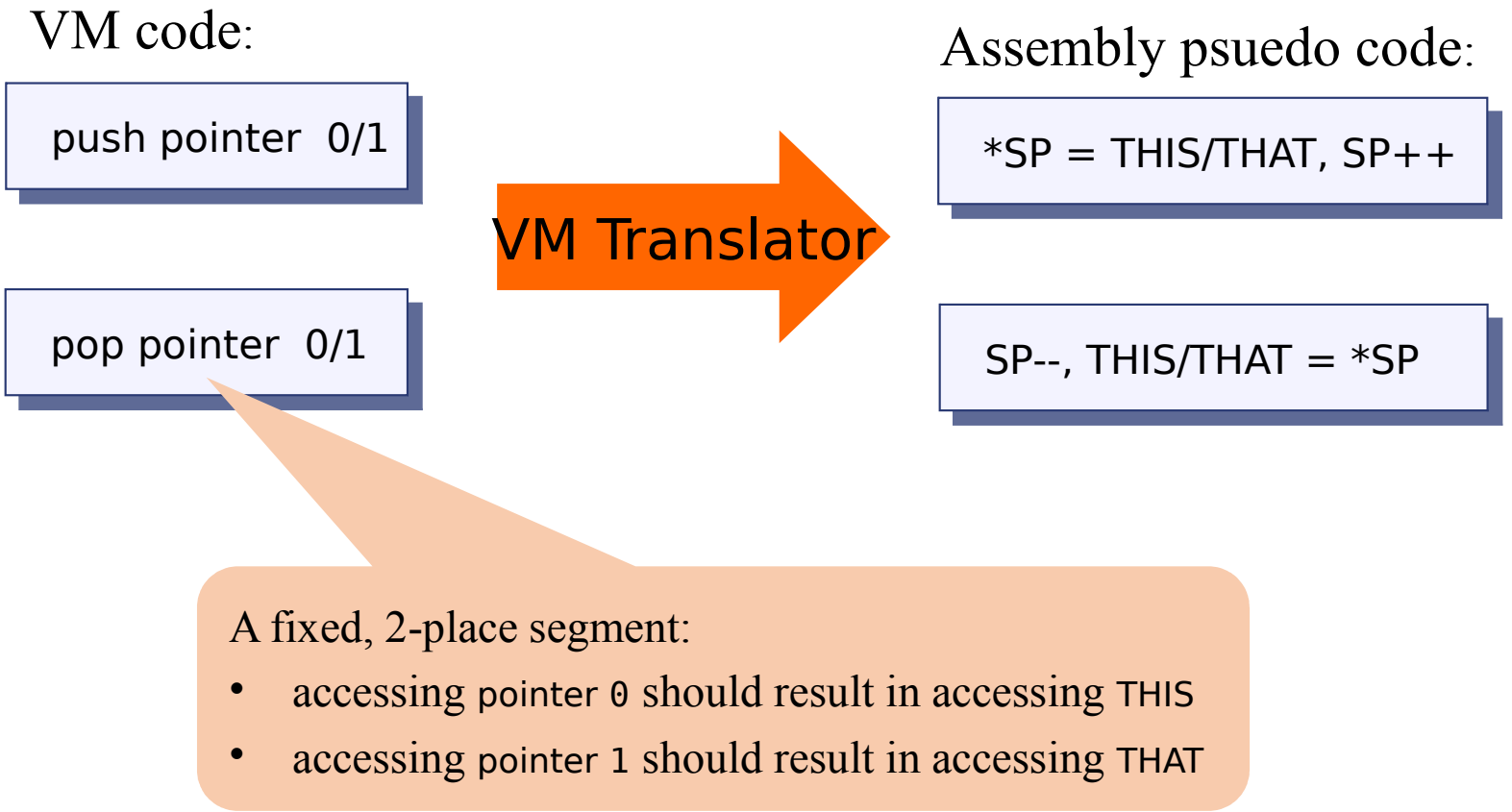
Syntax: push/pop pointer 0/1

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 would become clear when we would write the compiler (not done here).

Implementing push/pop pointer *0/1*

UNIVERSITY OF LEEDS



Implementation:

Supplies THIS or THAT // (the base addresses of this and that).

VM translator (so-far)

UNIVERSITY OF LEEDS

VM code

```
push constant 2
push local 0
sub
push local 1
push constant 5
add
sub
pop local 2
...
```

Each VM command is translated into several assembly commands

Assembly code

```
// push constant 2
@2
D=A
@SP
A=M
M=D
@SP
M=M+1
// push local 0
...
```

In order to write a VM translator, we must be familiar with:

- the source language
- the target language
- the VM mapping on the target platform.

Source: VM language

UNIVERSITY OF LEEDS

Arithmetic / Logical commands

add
sub
neg
eq
gt
lt
and
or
not

Memory access commands

pop *segment i*

push *segment i*

Branching commands

label *label*

goto *label*

if-goto *label*

Function commands

function *functionName nVars*

call *functionName nArgs*

return

Target: symbolic HACK code

UNIVERSITY OF LEEDS

A-instruction:

```
@ value     // A = value
```

where *value* is either a constant or a symbol referring to such a constant

C-instruction:

Syntax: `dest = comp ;jump` (both *dest* and *jump* are optional)

where:

comp =
```
0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D&A, D|A
M,  !M,  -M,  M+1,  M-1, D+M, D-M, M-D, D&M, D|M
```

dest = `null, M, D, MD, A, AM, AD, AMD` (M refers to RAM[A])

jump = `null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP`

Semantics:

- Computes the value of *comp*
- Stores the result in *dest*
- If the Boolean expression (*comp jump* `0`) is true, jumps to execute the instruction at ROM[A]

Standard VM mapping on the Hack platform

UNIVERSITY OF LEEDS

VM mapping decisions:

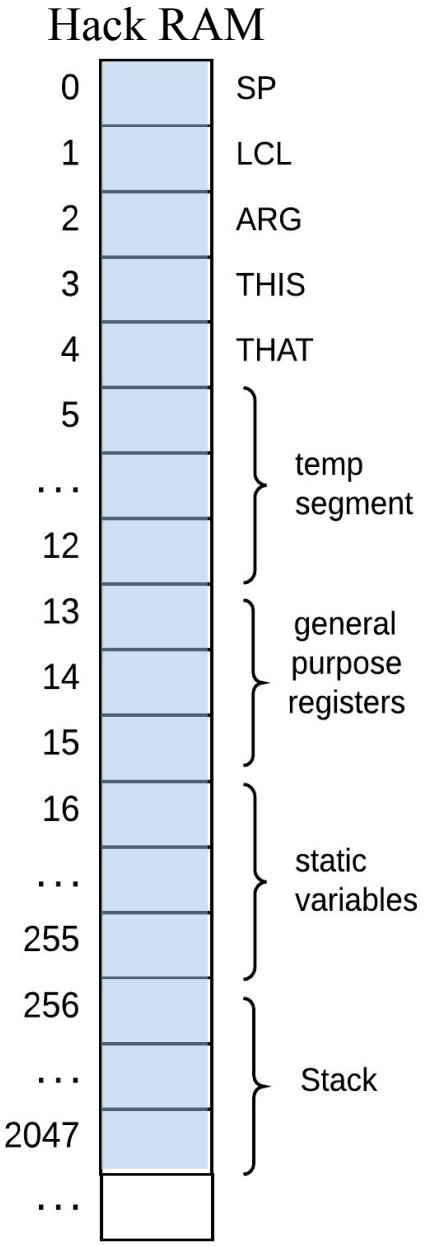
- How to map the VM's data structures using the host hardware platform
- How to express the VM's commands using the host machine language

Standard mapping:

- Specifies how to do the mapping in an agreed-upon way
- Benefits:
  - Compatibility with other software systems
  - Standard testing.

Standard VM mapping on the Hack platform

UNIVERSITY OF LEEDS



local, argument, this, that:

allocated dynamically to the `RAM`. The base addresses of these allocations are kept in the segment pointers `LCL`, `ARG`, `THIS`, `THAT`

accessing *segment i* should result in accessing RAM[**segmentPointer* + *i*]

constant: accessing `constant` *i* should result in supplying the constant *i*

static: accessing `static` *i* within file `Foo.vm` should result in accessing the assembly variable `Foo.`*i*

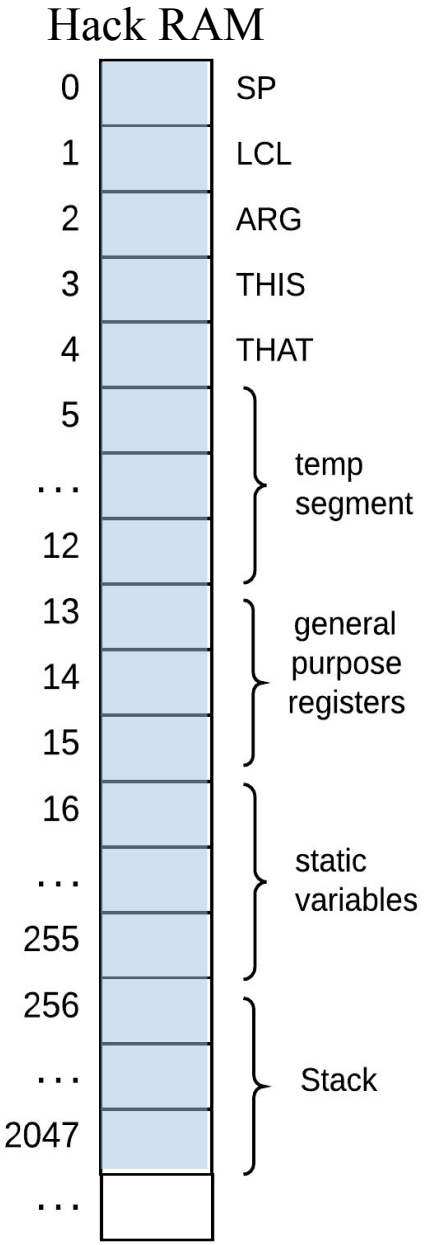
temp: fixed segment, mapped on `RAM` addresses 5-12.
accessing `temp` *i* should result in accessing `RAM`[5+*i*]

pointer: fixed segment, mapped on `RAM` addresses 3-4.

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

Standard VM mapping on the Hack platform

UNIVERSITY OF LEEDS



In order to realize this mapping, the VM translator should use some special variables / 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 addresses within the host RAM of the virtual segments `local`, `argument`, `this`, and `that` of the currently running VM function.
`R13`–`R15`	These predefined symbols can be used for any purpose.
`Xxx.`*i* symbols	The `static` segment is implemented as follows: each static variable *i* in file `Xxx.vm` is translated into the assembly symbol `Xxx.`*i*. In the subsequent assembly process, these symbolic variables will be allocated to the RAM by the Hack assembler.

<u>Implementation note:</u>

The standard mapping will be extended, when we'll complete the VM translator's implementation.

Source: VM language

UNIVERSITY OF LEEDS

Arithmetic / Logical commands

add
sub
neg
eq
gt
lt
and
or
not

Memory access commands

pop *segment i*
push *segment i*

Branching commands

label *label*
goto *label*
if-goto *label*

Function commands

function *functionName nVars*
call *functionName nArgs*
return

Branching: label *label*

UNIVERSITY OF LEEDS

- This command labels the current location in the function's code.
- Only labeled locations can be jumped to from other parts of the program.
- The scope of the label is the function in which it is defined.
- The label is an arbitrary string composed of any sequence of letters, digits, underscore (_), dot (.), and colon (:) that does not begin with a digit.

Branching: goto *label*

UNIVERSITY OF LEEDS

- This command effects an unconditional goto operation, causing execution to continue from the location marked by the label.
- The jump destination must be located in the same function.

Branching: if-goto *label*

UNIVERSITY OF LEEDS

- This command effects a conditional goto operation.
- The stack's topmost value is popped; if the value is not zero, execution continues from the location marked by the label; otherwise, execution continues from the next command in the program.
- The jump destination must be located in the same function.

Branching

UNIVERSITY OF LEEDS

Recap: VM branching commands:

- 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):

Translate each branching command into assembly instructions that effect the specified operation on the host machine

Simple implementation:

the assembly language has similar branching commands.

Source: VM language

UNIVERSITY OF LEEDS

Arithmetic / Logical commands

add
sub
neg
eq
gt
lt
and
or
not

Branching commands

label *label*
goto *label*
if-goto *label*

Function commands

function *functionName nVars*
call *functionName nArgs*
return

Memory access commands

pop *segment i*
push *segment i*

Function - Recap

UNIVERSITY OF LEEDS

The VM language features three function-related commands:

- **`function`** f n
  Here starts the code of a function named f that has n local variables;

- **`call`** f m
  Call function f, stating that m arguments have already been pushed onto the stack by the caller;

- **`return`**
  Return to the calling function.

Function execution

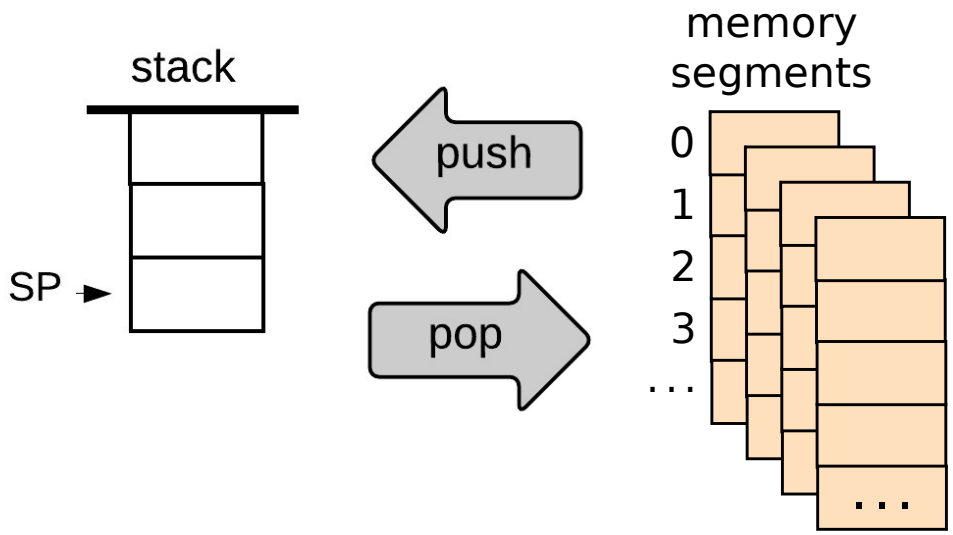
UNIVERSITY OF LEEDS

- A computer program typically consists of many functions
- At any given point of time, only a few functions are executing
- Calling chain: `foo > bar > sqrt > ...`

For each function in the calling chain during run-time, we must maintain the function's ***state = working stack and memory segments***

Function's state

UNIVERSITY OF LEEDS



<u>During run-time:</u>

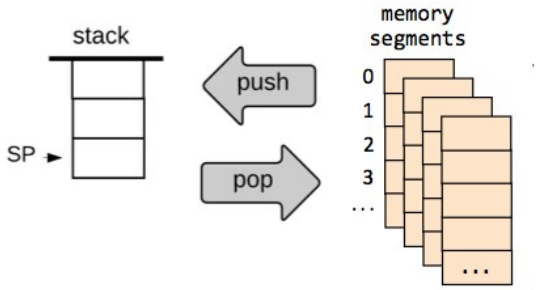
- Each function uses a working stack + memory segments
- The working stack and some of the segments should be:
  - Created when the function starts running,
  - Maintained as long as the function is executing,
  - Recycled when the function returns.

Function's state

UNIVERSITY OF LEEDS

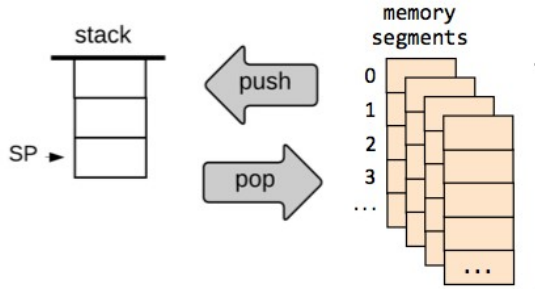
```
function main 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  return
```

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



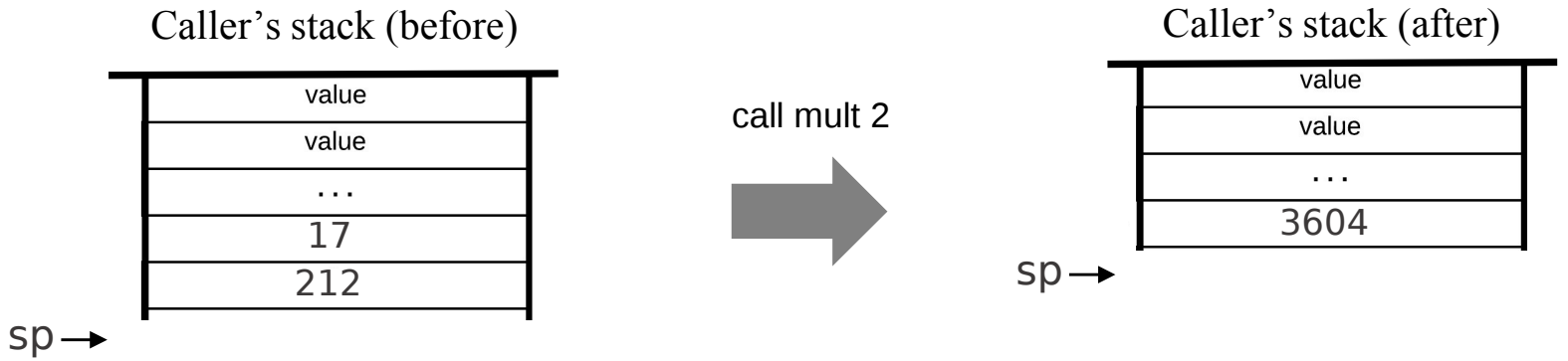
Challenge:

- 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

UNIVERSITY OF LEEDS

Example: computing `mult(17,212)`

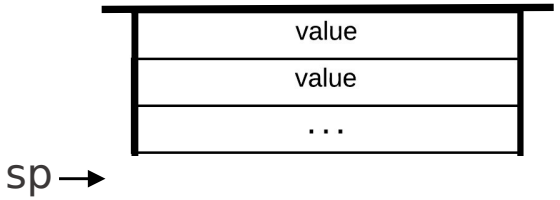


Net effect:

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

Function call and return: implementation

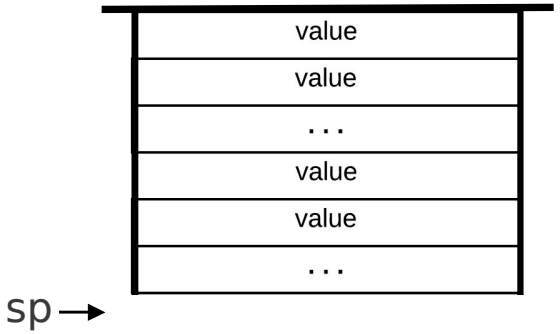
The function is running,
doing something



Function call and return: implementation

UNIVERSITY OF LEEDS

The function prepares
to call another function:

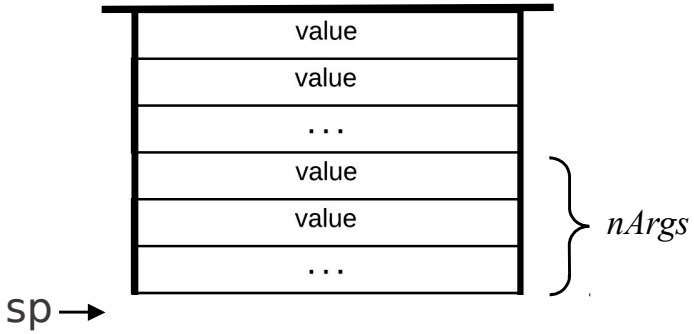


Function call and return: implementation

UNIVERSITY OF LEEDS

The function says:

call *foo nArgs*

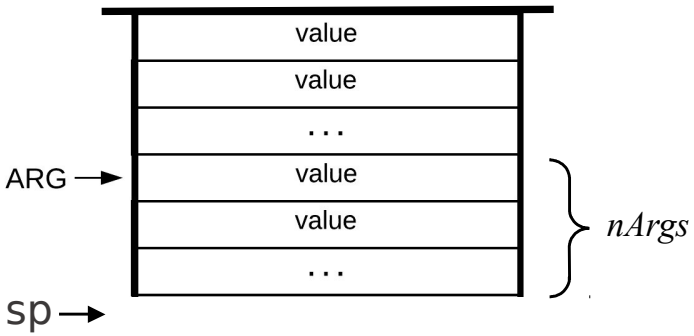


Function call and return: implementation

UNIVERSITY OF LEEDS

The function says:

call *foo nArgs*



VM implementation (handling call):

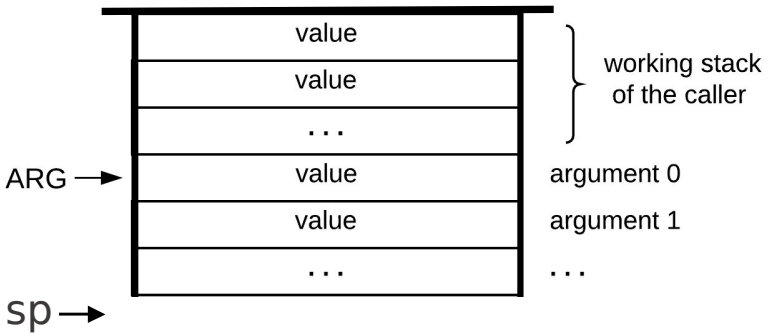
1. Sets ARG

Function call and return: implementation

UNIVERSITY OF LEEDS

The function says:

call *foo nArgs*



VM implementation (handling call):

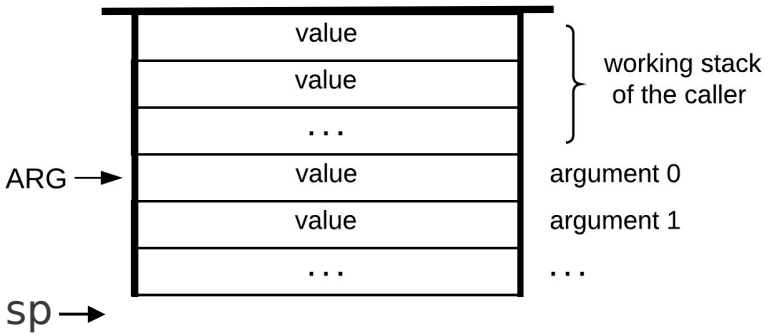
1. Sets ARG

Function call and return: implementation

UNIVERSITY OF LEEDS

The function says:

call *foo nArgs*



VM implementation (handling call):

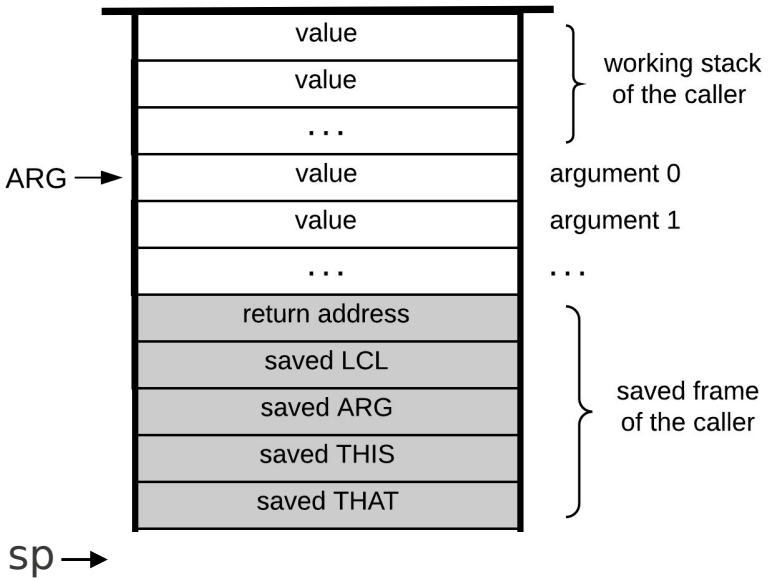
1. Sets ARG
2. Saves the caller's frame

Function call and return: implementation

UNIVERSITY OF LEEDS

The function says:

call *foo nArgs*



VM implementation (handling call):

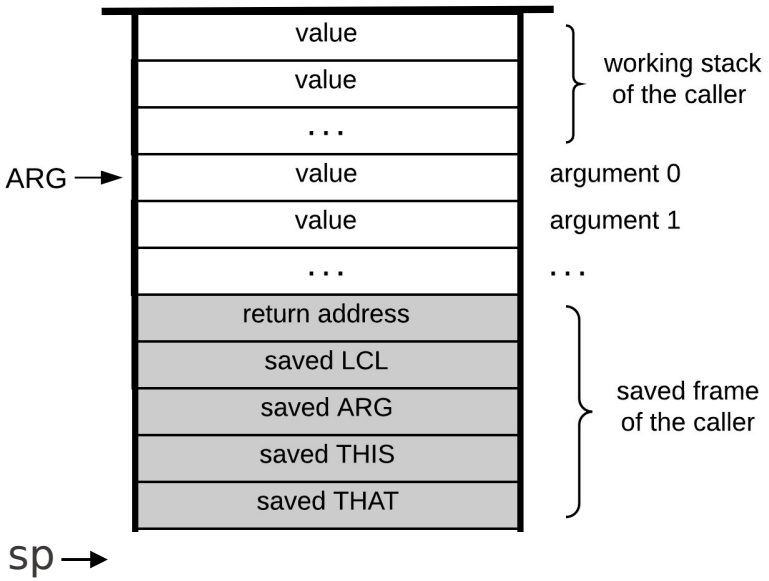
1. Sets ARG
2. Saves the caller's frame
3. Jumps to execute *foo*

Function call and return: implementation

UNIVERSITY OF LEEDS

The called function is entered:

function *foo nVars*



VM implementation (handling `call`):

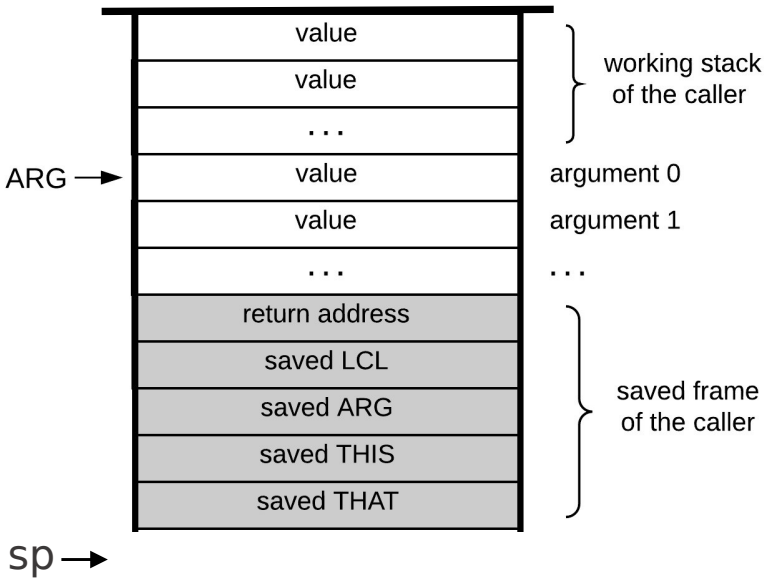
1. Sets ARG
2. Saves the caller's frame
3. Jumps to execute *foo*

Function call and return: implementation

UNIVERSITY OF LEEDS

The called function is entered:

function *foo nVars*



VM implementation (handling call):

1. Sets ARG
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling function):

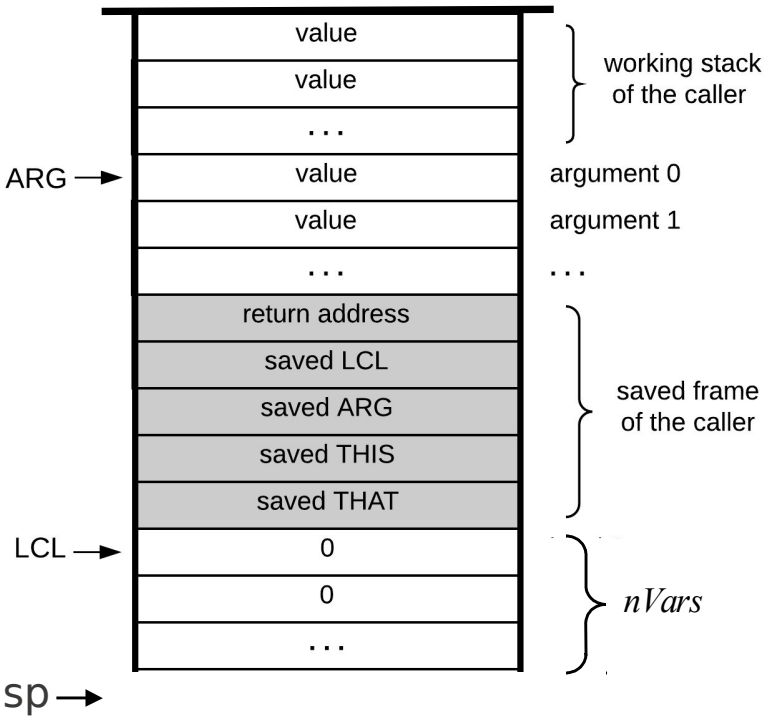
Sets up the local segment of the called function

Function call and return: implementation

UNIVERSITY OF LEEDS

The called function is entered:

function *foo nVars*



VM implementation (handling call):

1. Sets ARG
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling function):

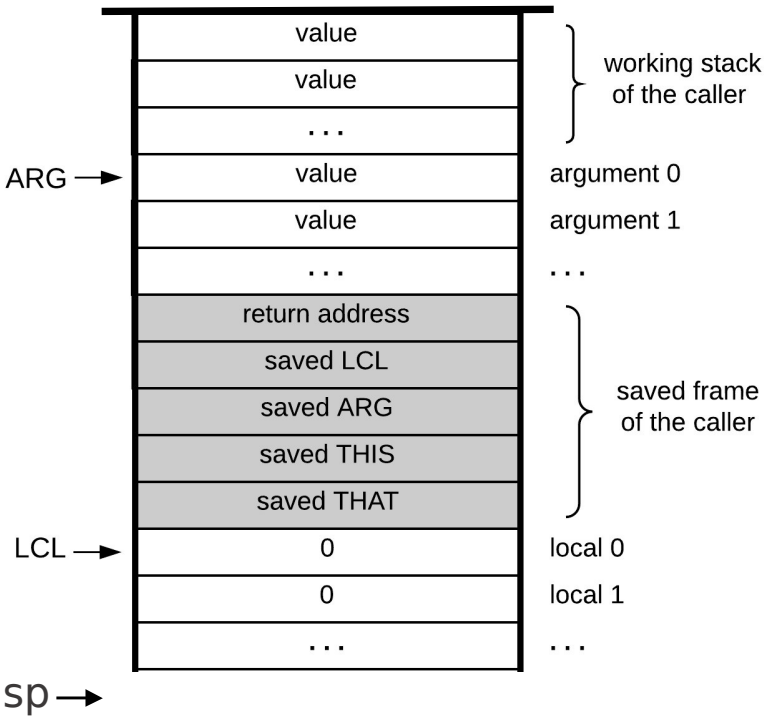
Sets up the local segment of the called function

Function call and return: implementation

UNIVERSITY OF LEEDS

The called function is entered:

function *foo nVars*



VM implementation (handling call):

1. Sets ARG
2. Saves the caller's frame
3. Jumps to execute *foo*

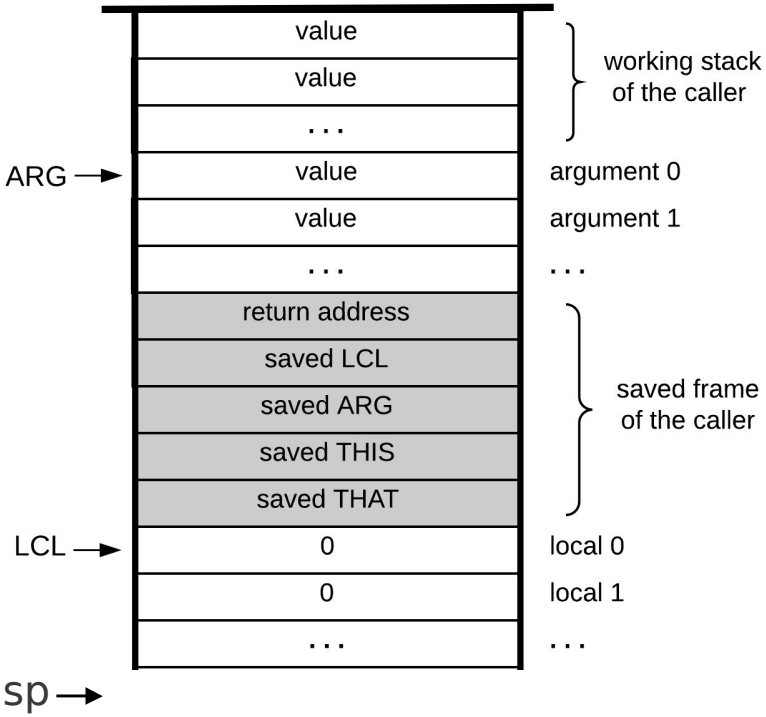
VM implementation (handling function):

Sets up the local segment of the called function

Function call and return: implementation

UNIVERSITY OF LEEDS

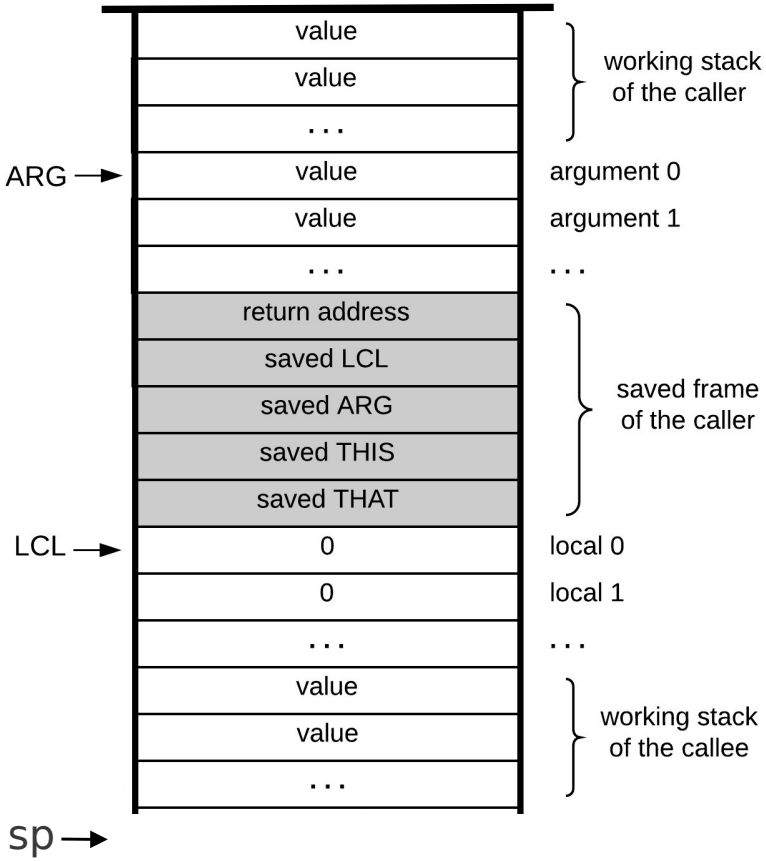
The called function is running, doing something



Function call and return: implementation

UNIVERSITY OF LEEDS

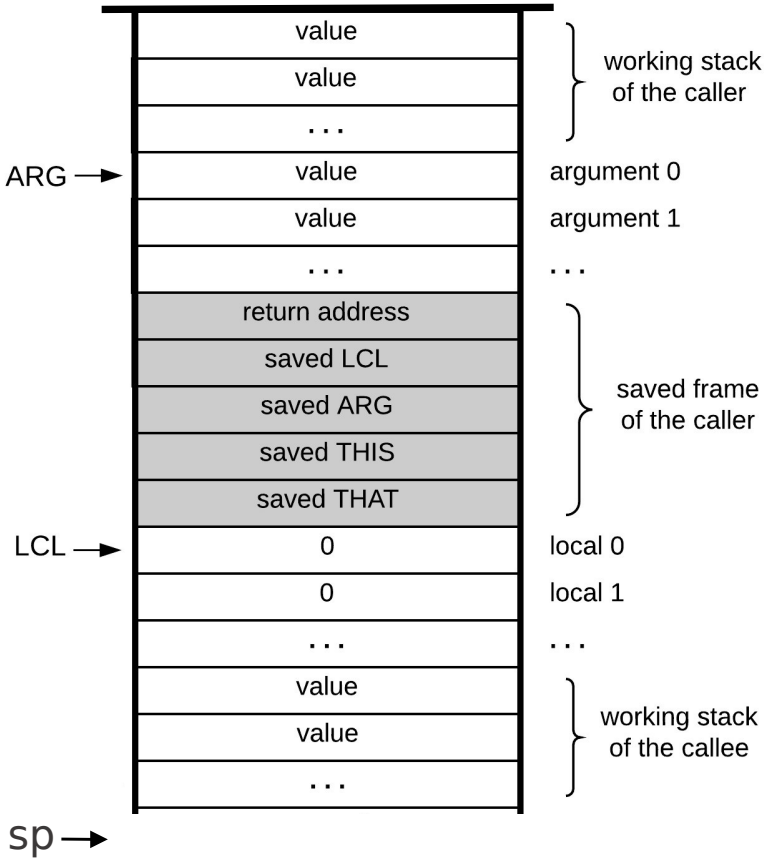
The called function is running,
doing something



Function call and return: implementation

UNIVERSITY OF LEEDS

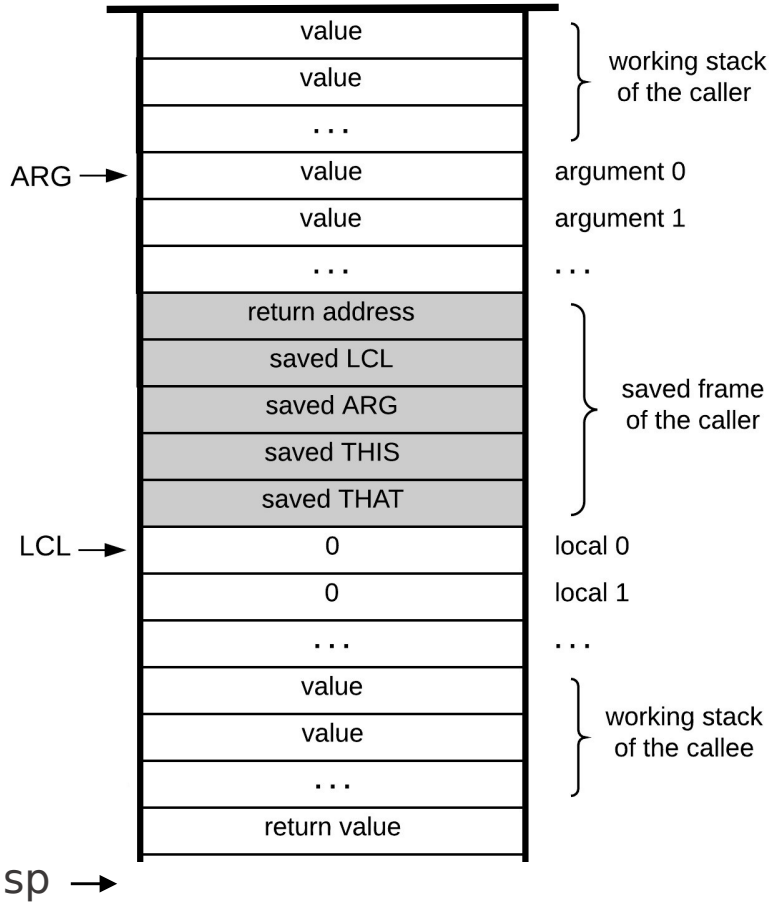
The called function prepares to return:
it pushes a *return value*



Function call and return: implementation

UNIVERSITY OF LEEDS

The called function prepares to return:
it pushes a *return value*

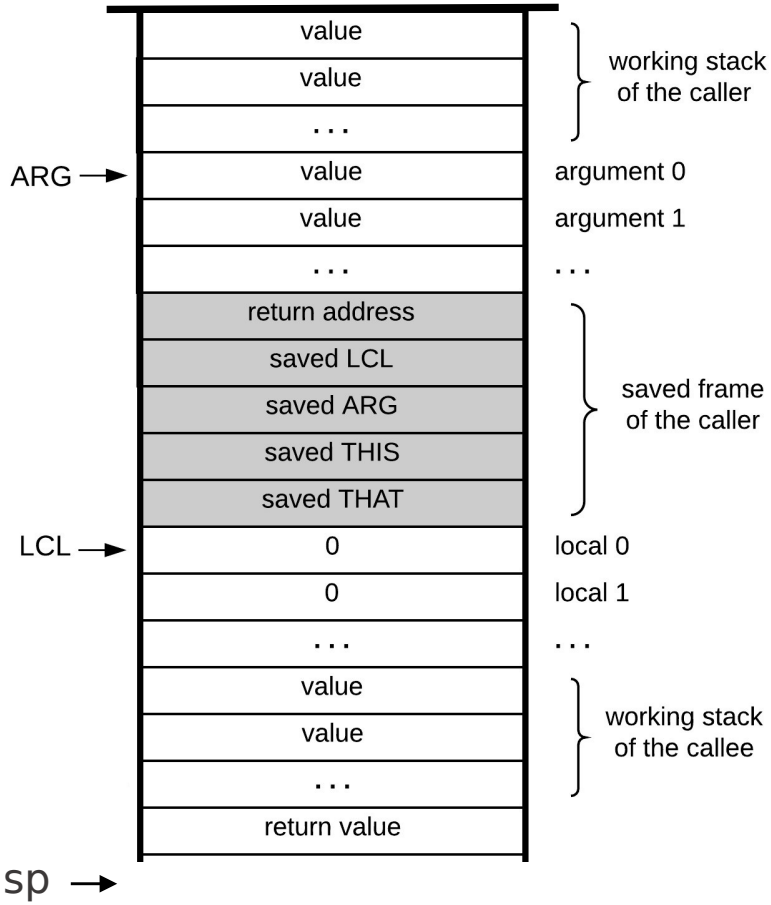


Function call and return: implementation

UNIVERSITY OF LEEDS

The called function says:

return

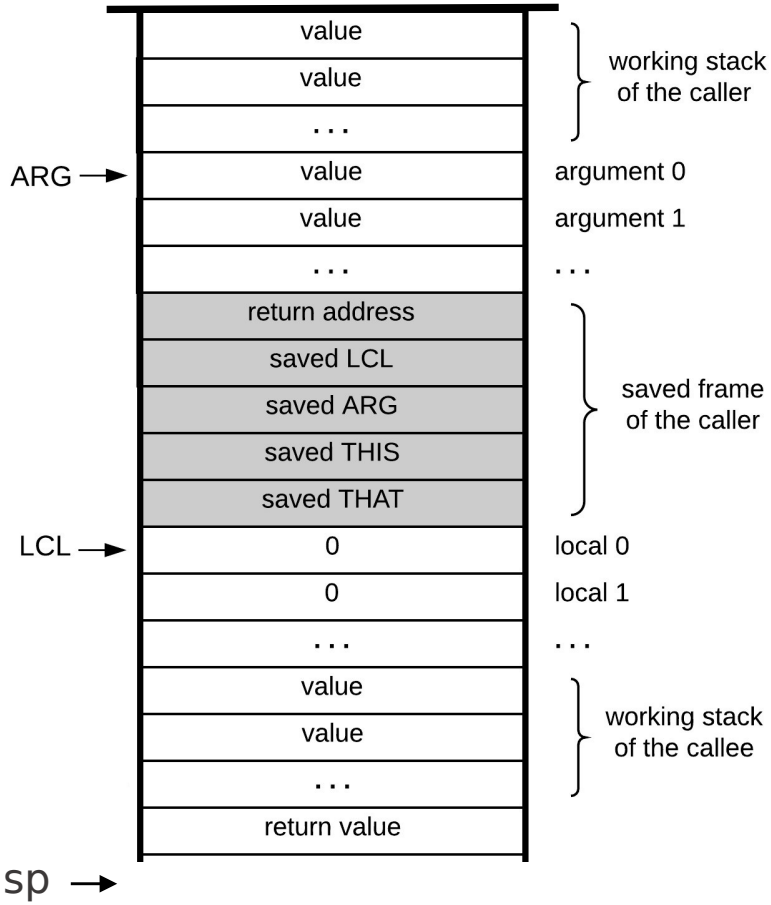


Function call and return: implementation

UNIVERSITY OF LEEDS

The called function says:

return



VM implementation (handling call):

1. Sets ARG
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling return):

Sets up the local segment of the called function

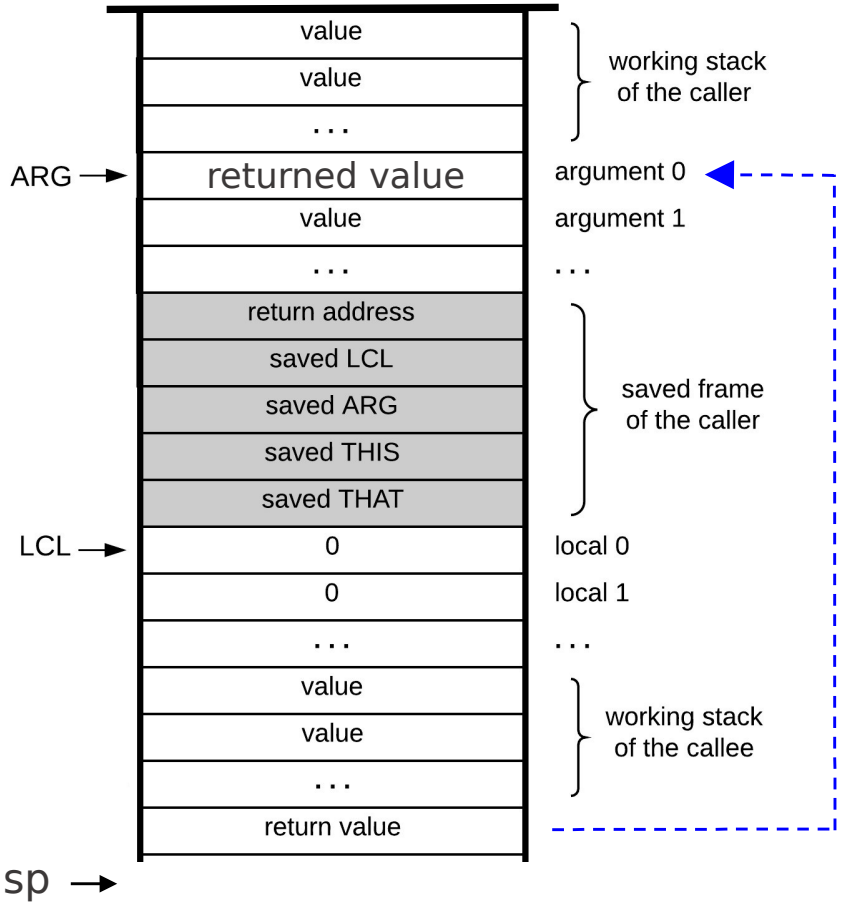
VM implementation (handling return):

Function call and return: implementation

UNIVERSITY OF LEEDS

The called function says:

return



VM implementation (handling call):

1. Sets ARG
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling return):

Sets up the local segment of the called function

VM implementation (handling return):

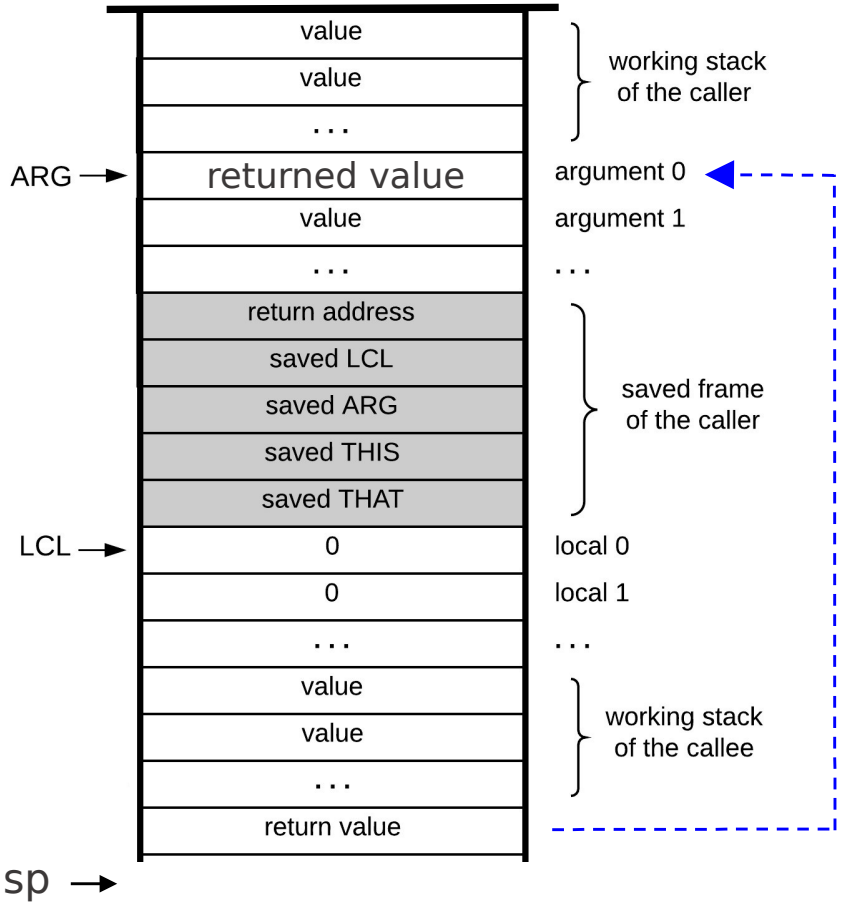
1. Copies the return value onto argument 0

Function call and return: implementation

UNIVERSITY OF LEEDS

The called function says:

return



VM implementation (handling call):

1. Sets ARG
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling return):

Sets up the local segment of the called function

VM implementation (handling return):

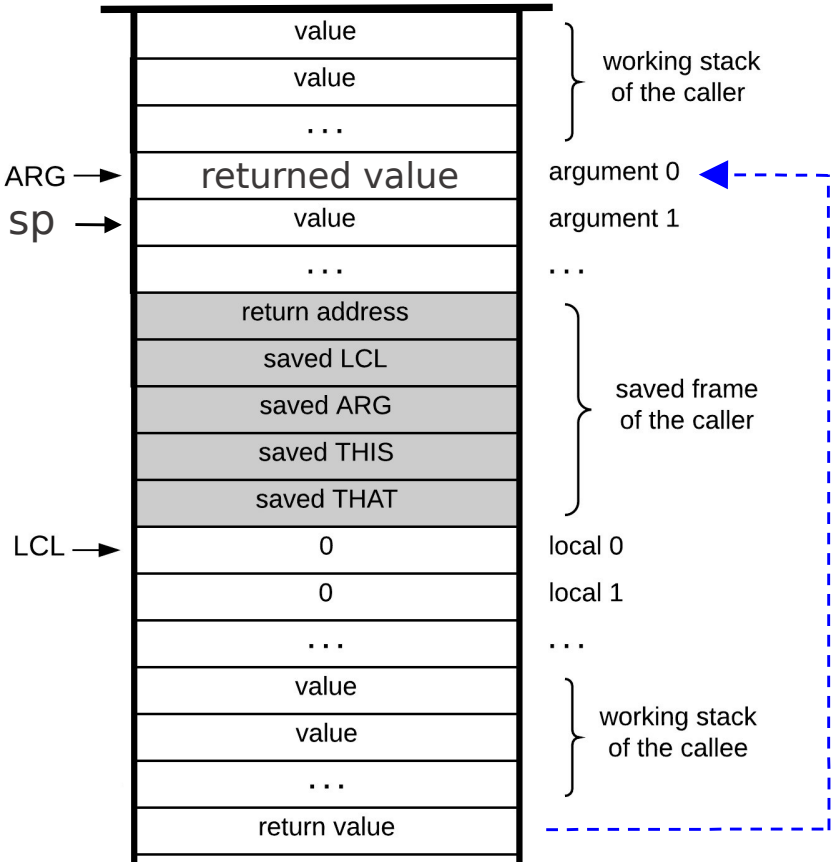
1. Copies the return value onto argument 0
2. Sets SP for the caller

Function call and return: implementation

UNIVERSITY OF LEEDS

The called function says:

return



VM implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling `return`):

Sets up the `local` segment of the called function

VM implementation (handling `return`):

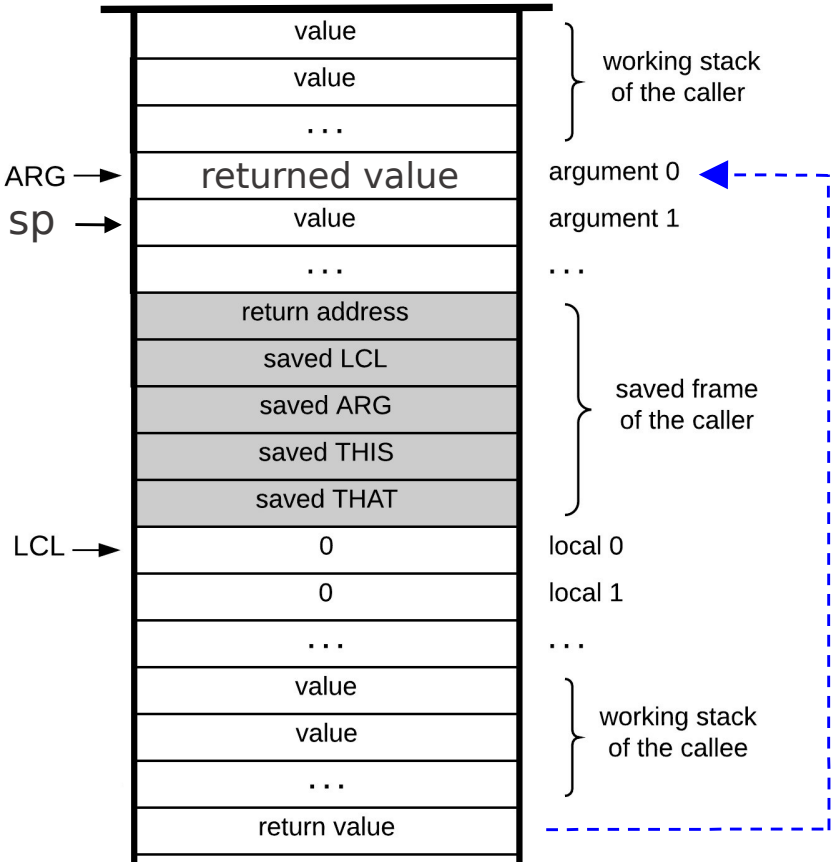
1. Copies the return value onto `argument 0`
2. Sets `SP` for the caller

Function call and return: implementation

UNIVERSITY OF LEEDS

The called function says:

return



VM implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling `return`):

Sets up the `local` segment of the called function

VM implementation (handling `return`):

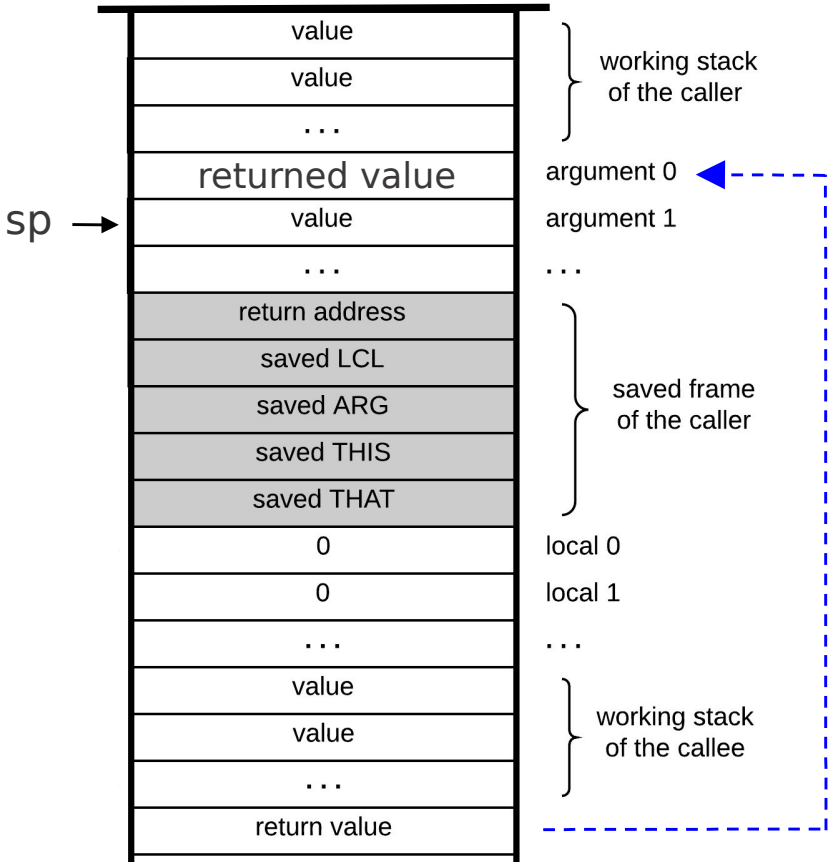
1. Copies the return value onto `argument 0`
2. Sets `SP` for the caller
3. Restores the segment pointers of the caller

Function call and return: implementation

UNIVERSITY OF LEEDS

The called function says:

return



VM implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling `return`):

Sets up the `local` segment of the called function

VM implementation (handling `return`):

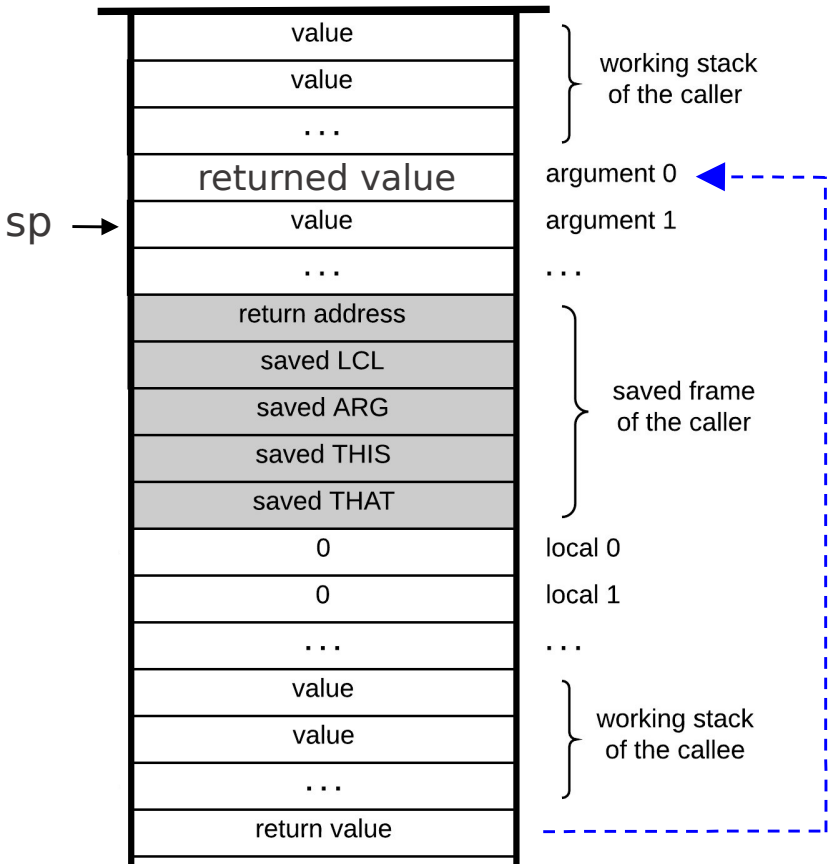
1. Copies the return value onto `argument 0`
2. Sets `SP` for the caller
3. Restores the segment pointers of the caller

Function call and return: implementation

UNIVERSITY OF LEEDS

The called function says:

return



VM implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling `return`):

Sets up the `local` segment of the called function

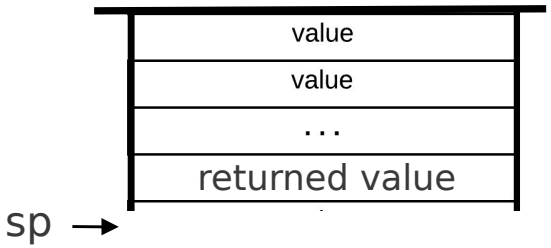
VM implementation (handling `return`):

1. Copies the return value onto `argument 0`
2. Sets `SP` for the caller
3. Restores the segment pointers of the caller
4. Jumps to the return address within the caller's code (note that the stack space below `SP` is recycled)

Function call and return: implementation

UNIVERSITY OF LEEDS

The caller
resumes its execution



VM implementation (handling call):

1. Sets ARG
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling return):

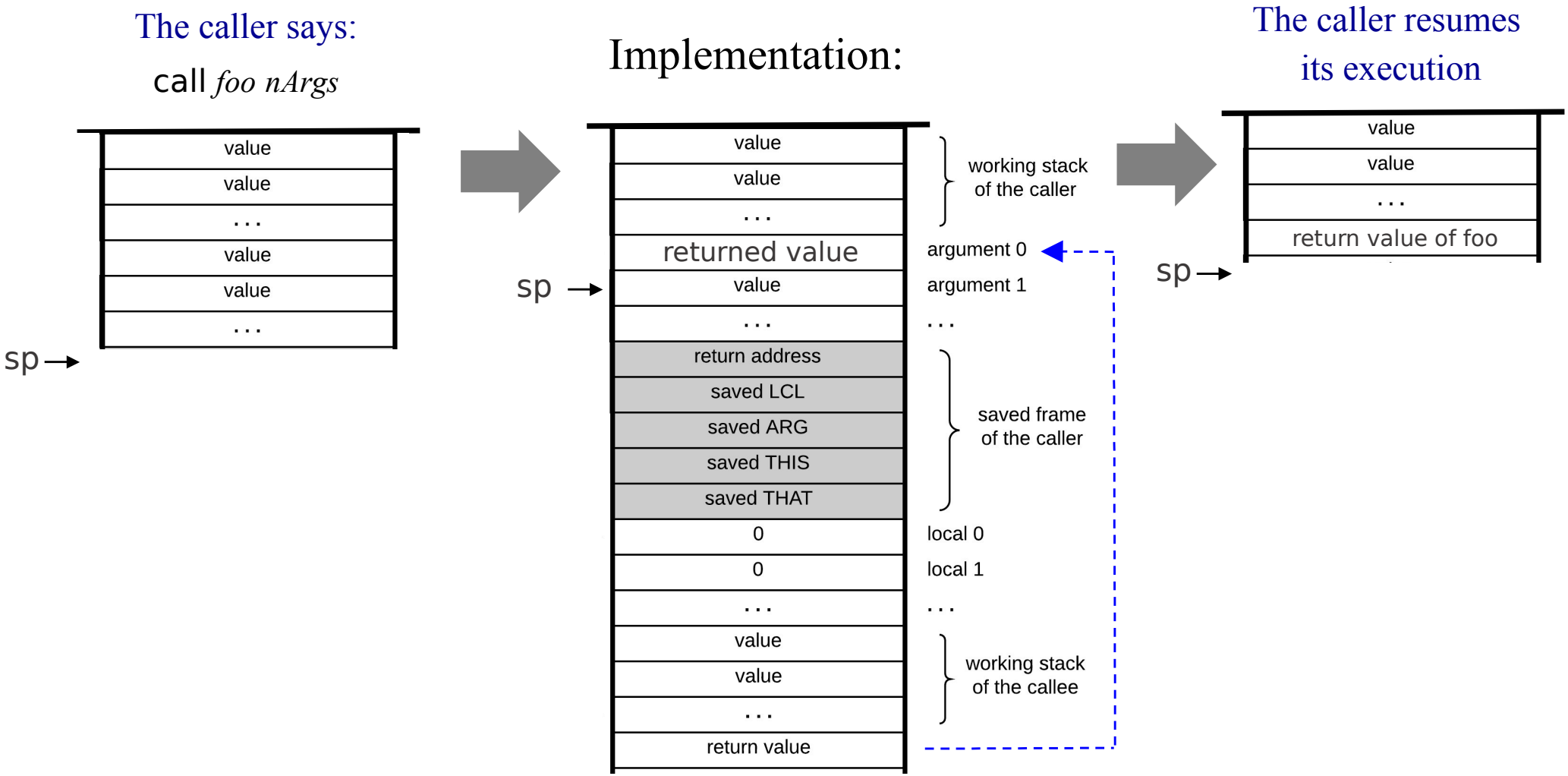
Sets up the local segment
of the called function

VM implementation (handling return):

1. Copies the return value onto argument 0
2. Sets SP for the caller
3. Restores the segment pointers of the caller
4. Jumps to the return address within the caller's code
   (note that the stack space below SP is recycled)

Function call and return: implementation

UNIVERSITY OF LEEDS

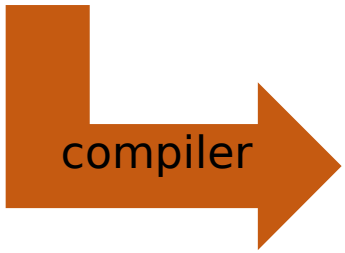


Function call and return: example factorial

UNIVERSITY OF LEEDS

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);
}
```

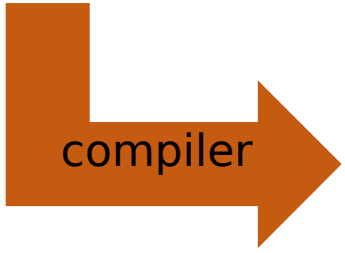


Function call and return: example factorial

UNIVERSITY OF LEEDS

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);
}
```



Pseudo VM code

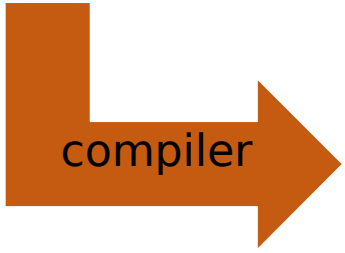
```
function main
   push 3
   call factorial
   return
```

Function call and return: example factorial

UNIVERSITY OF LEEDS

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);
}
```



Pseudo VM code

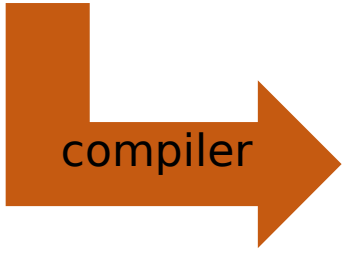
```
function main
  push 3
  call factorial
  return

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

Function call and return: example factorial

UNIVERSITY OF LEEDS

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);
}
```



Pseudo VM code

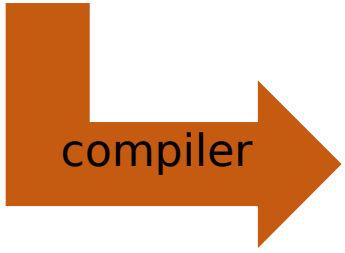
```
function main
  push 3
  call factorial
  return

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

  push n  //else-part
  push n
  push 1
  sub
  call factorial
  call mult
  return
```

Function call and return: example factorial

UNIVERSITY OF LEEDS

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);
}
```



Pseudo VM code

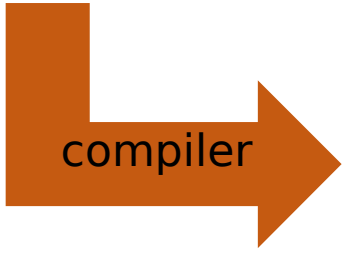
```
function main
  push 3
  call factorial
  return

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

  push n  //else-part
  push n
  push 1
  sub
  call factorial
  call mult
  return

label BASECASE
  push 1
  return

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

Function call and return: example factorial

UNIVERSITY OF LEEDS

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);
}
```



Pseudo VM code

```
function main
   push 3
   call factorial
   return

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

   push n  //else-part
   push n
   push 1
   sub
   call factorial
   call mult
   return

label BASECASE
   push 1
   return

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

VM program

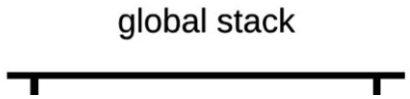
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   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
```

Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



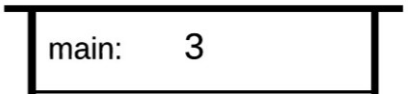
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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



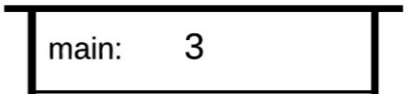
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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

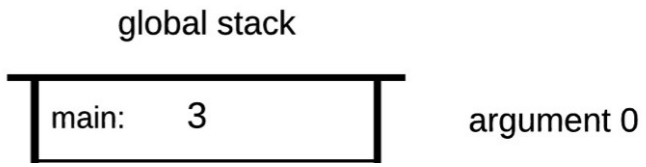


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



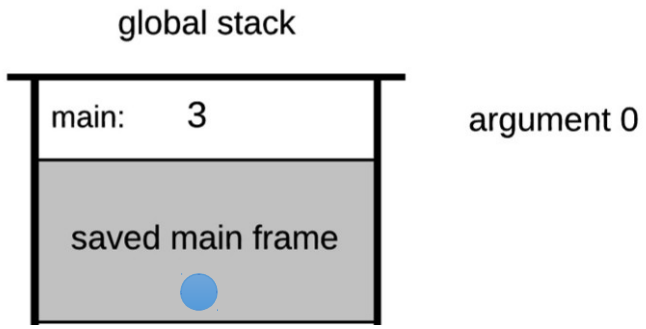
call factorial 1: only the first element on top of stack serves as argument (= argument 0)

Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



call factorial 1: only the first element on top of stack serves as argument (= argument 0)

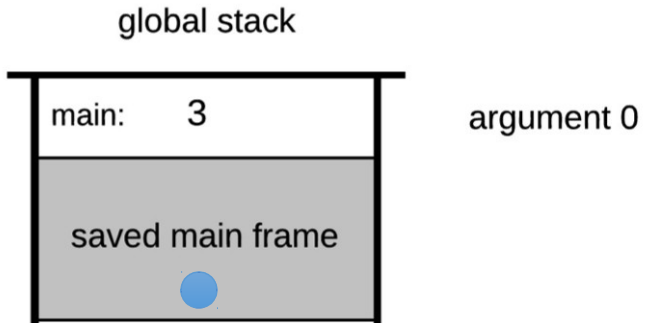
In addition, we have to save frame of `main` onto the stack (incl. the return address = which is the next command after the current call of factorial)

Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



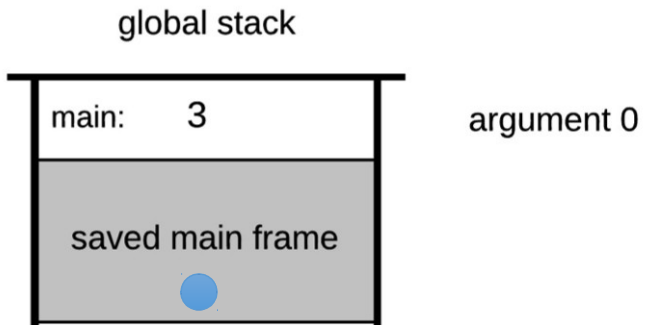
function factorial 0: no local variables (everything we need is argument0 which is already in the stack)

Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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

Q: Is 3=1?
A: No and thus, don't goto BASECASE

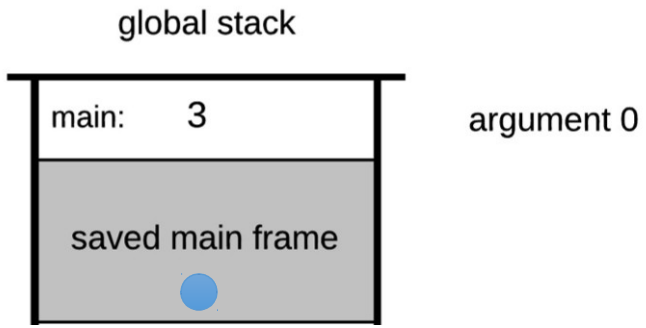
Note, ***after*** these four commands have been executed the stack remained “unchanged”

Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```

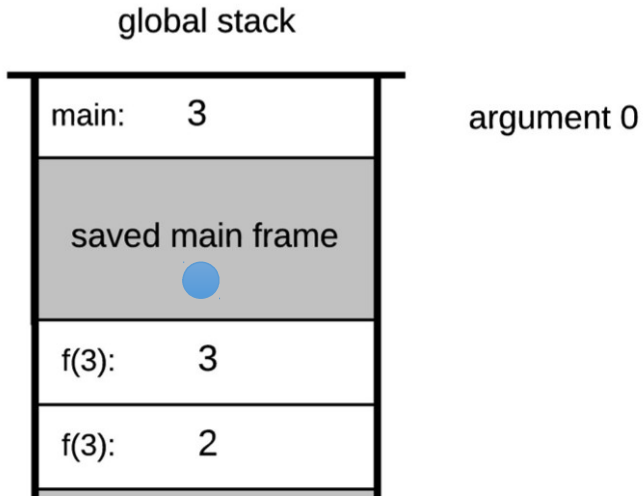


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```

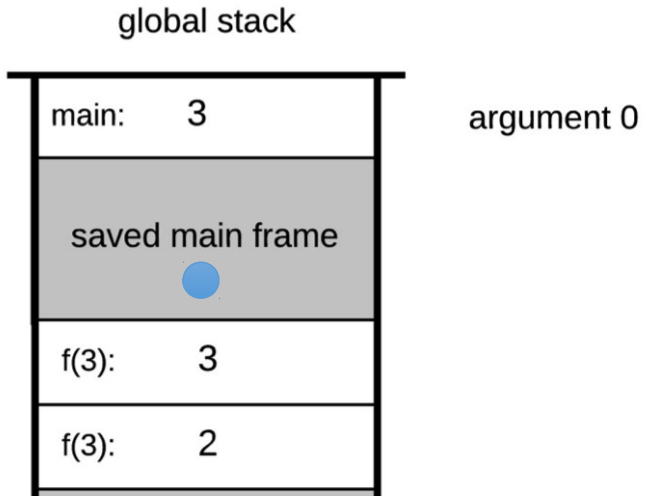


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```

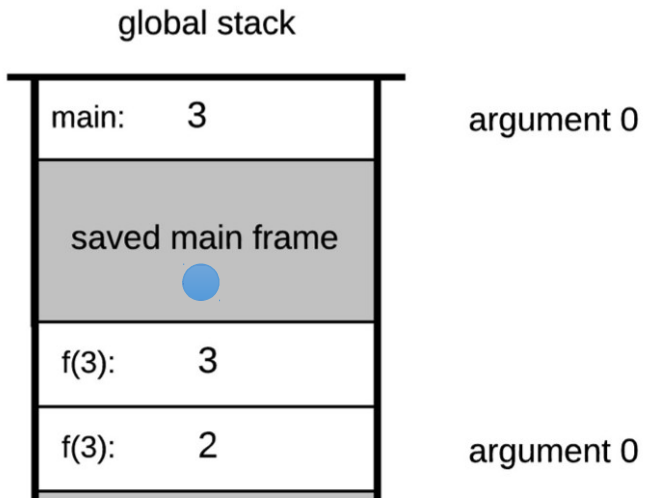


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



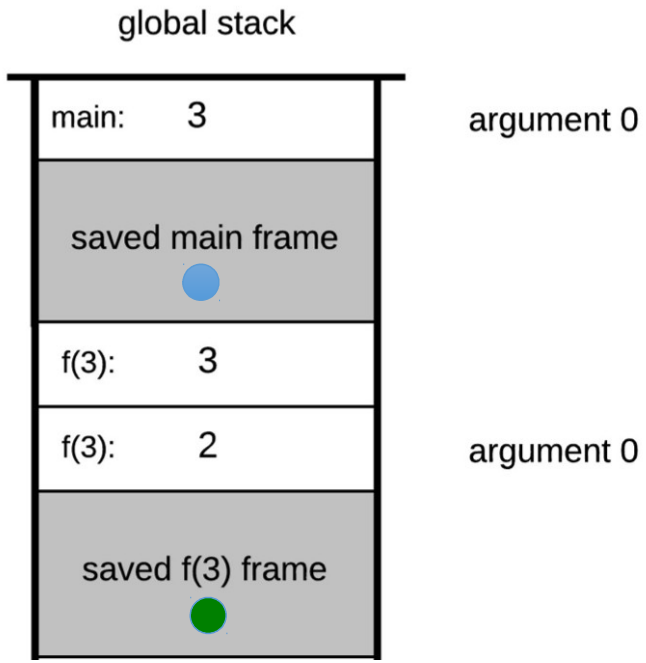
call factorial 1: only the first element on top of stack serves as argument (= argument 0)

Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



call factorial 1: only the first element on top of stack serves as argument (= argument 0)

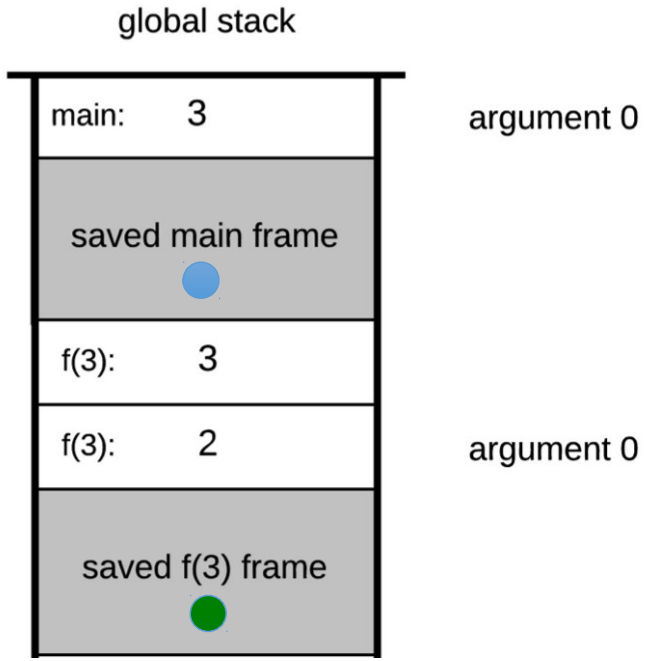
In addition, we have to save the frame of current function "f(3)" onto the stack
(incl. the return address = ● which is the next command after the current call of factorial)

Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



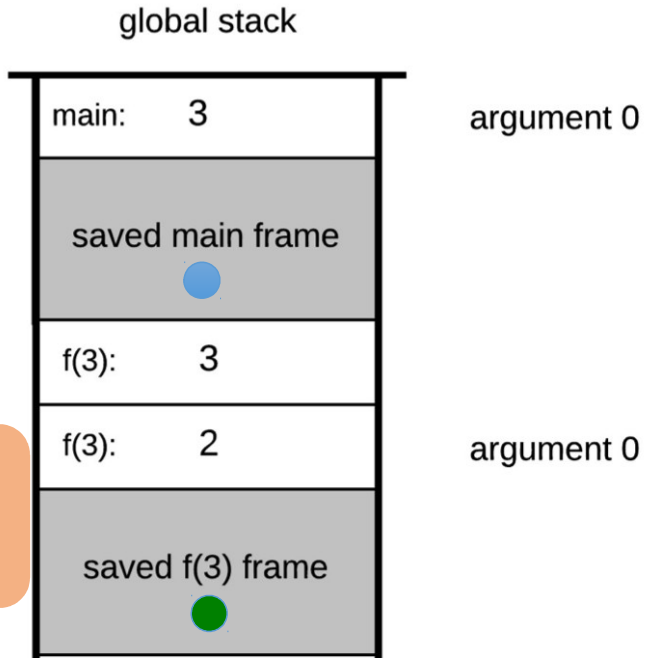
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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



Q: Is 2=1?
A: No and thus, don't goto BASECASE

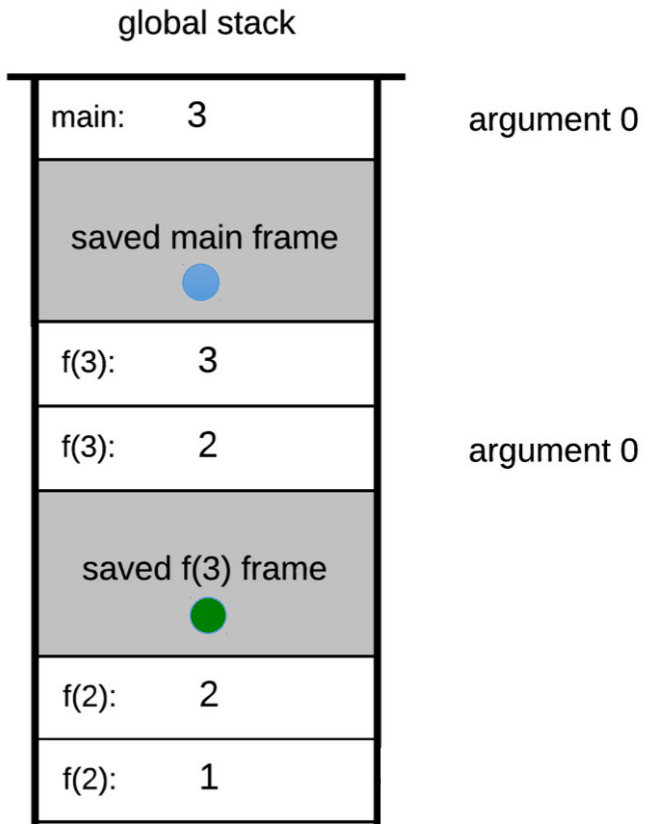
Note, ***after*** these four commands have been executed the stack remained “unchanged”

Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```

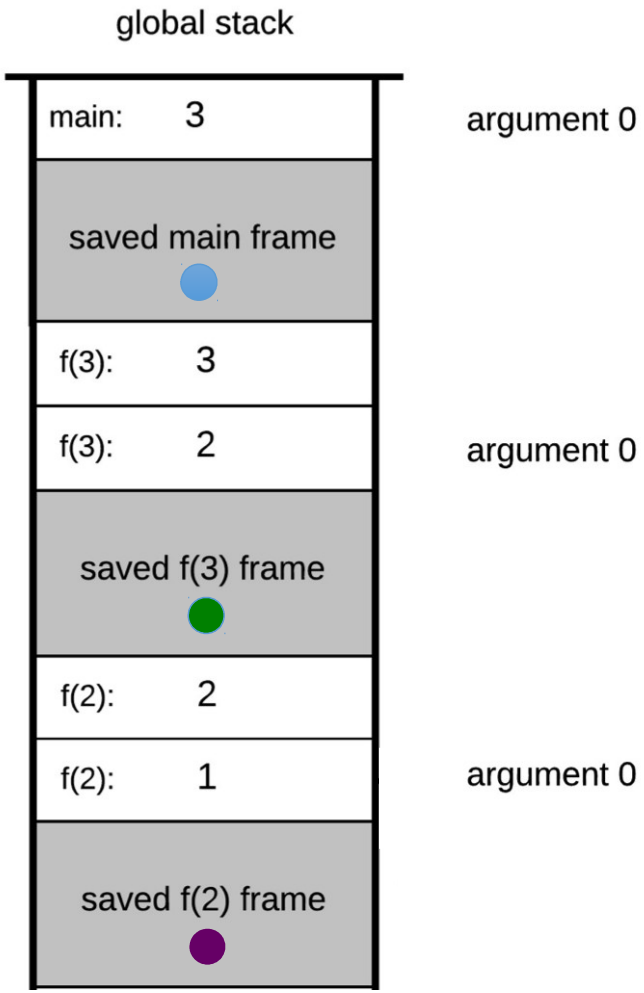


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



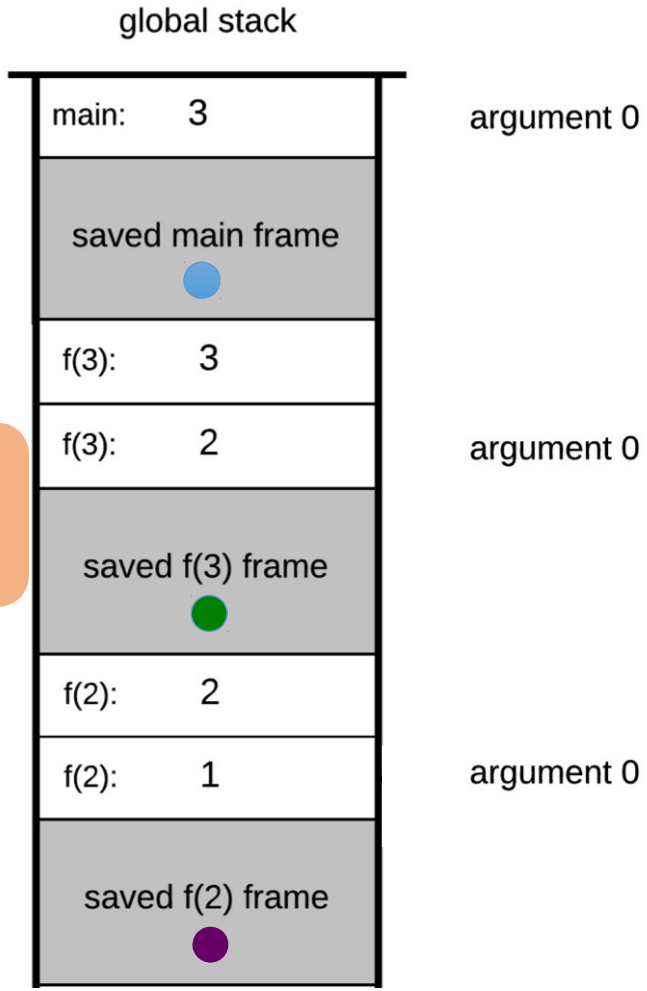
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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



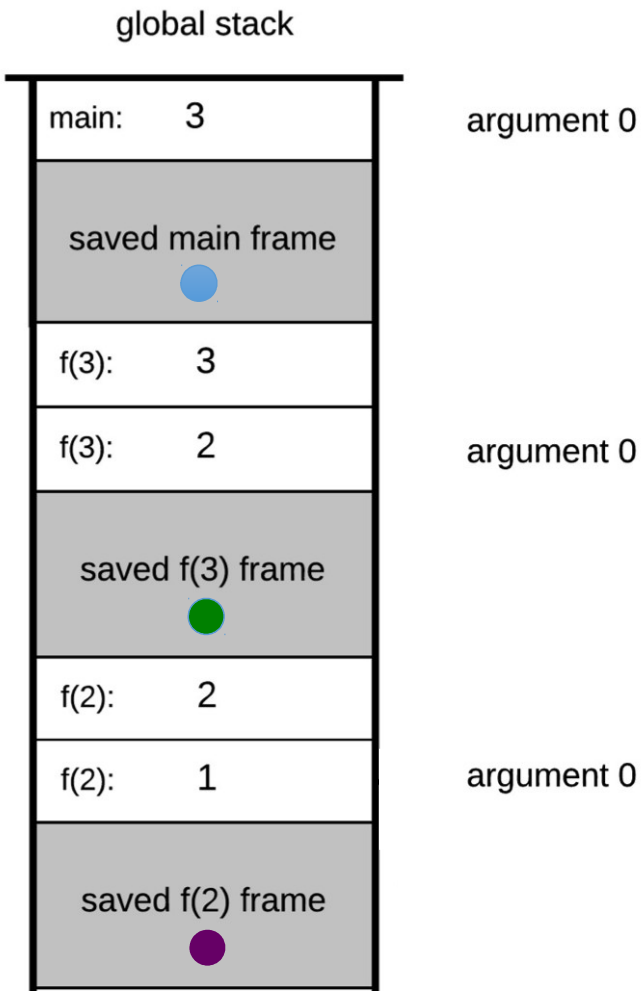
Q: Is 1=1?
A: Yes, and thus, goto BASECASE
”

Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```

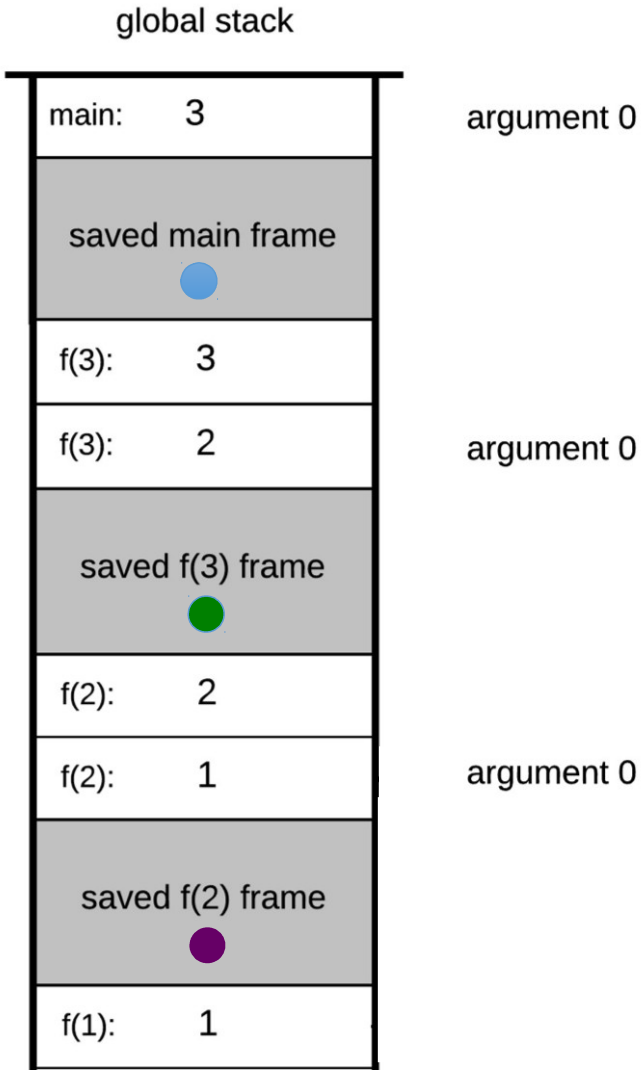


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```

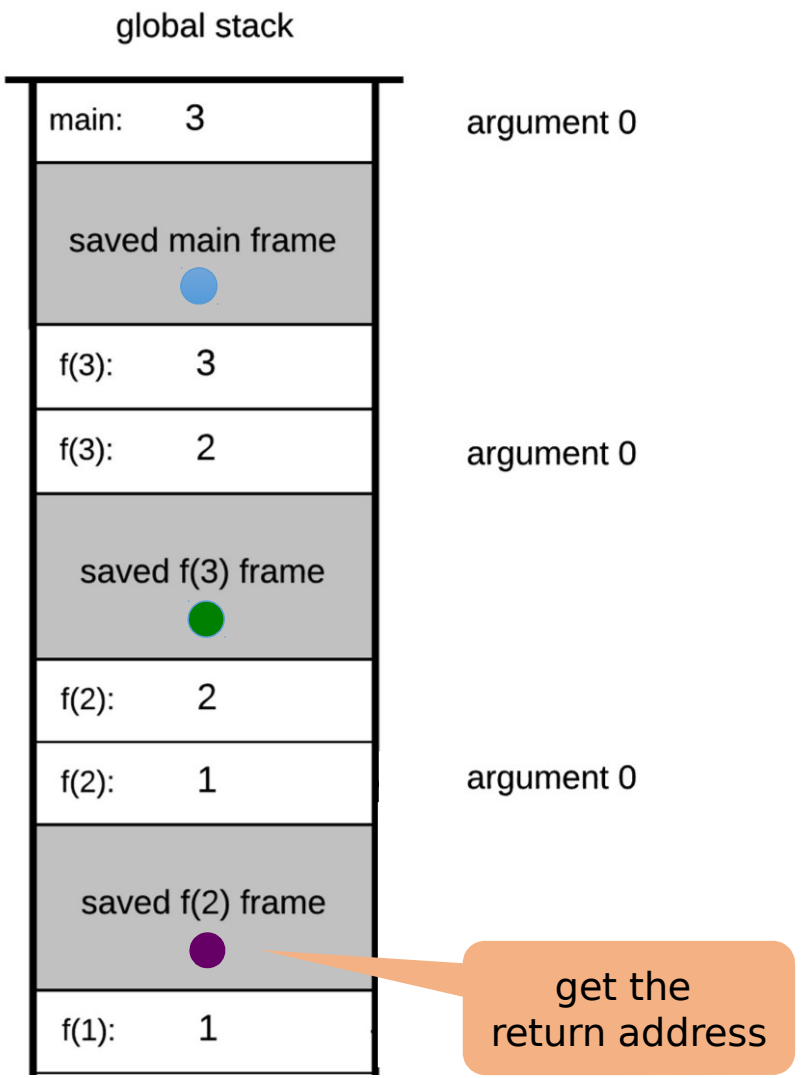


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```

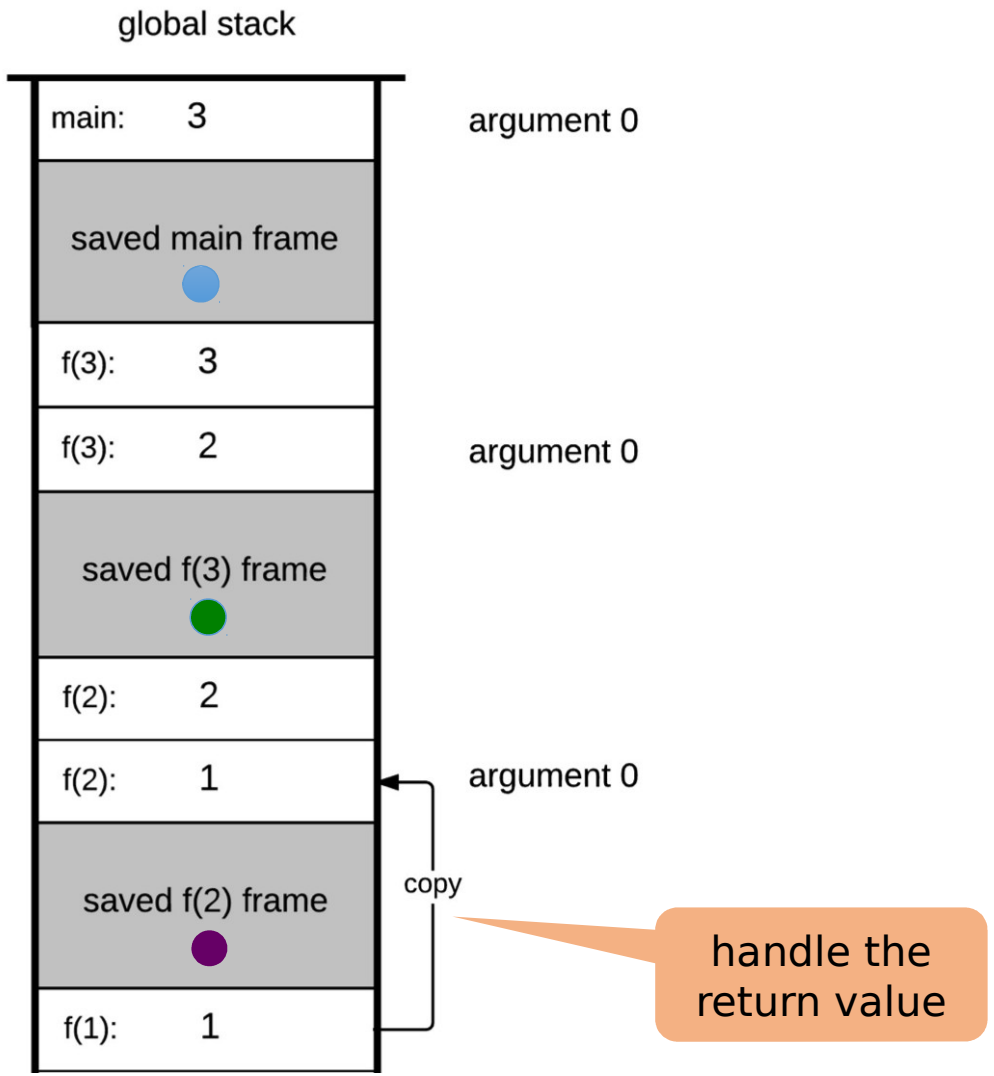


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



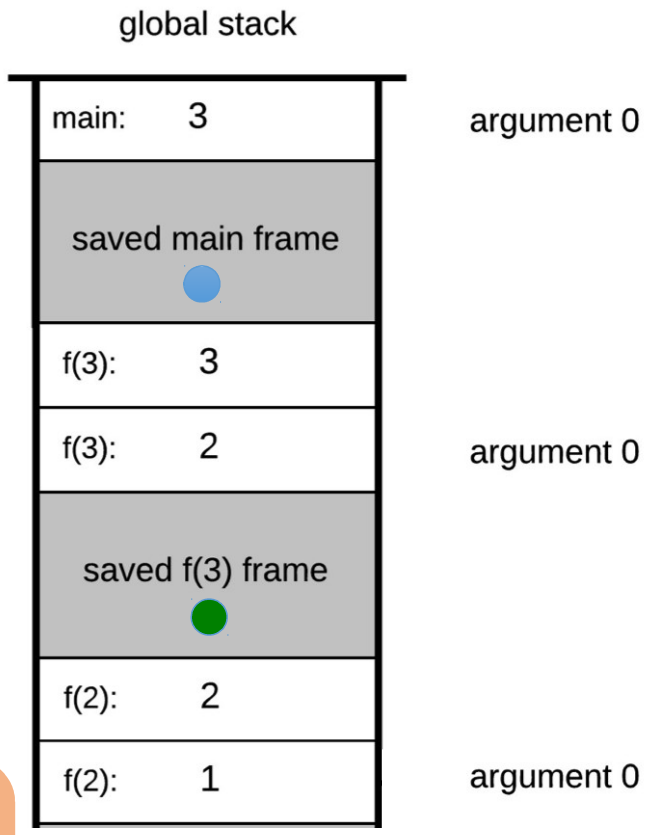
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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 (except the end result)



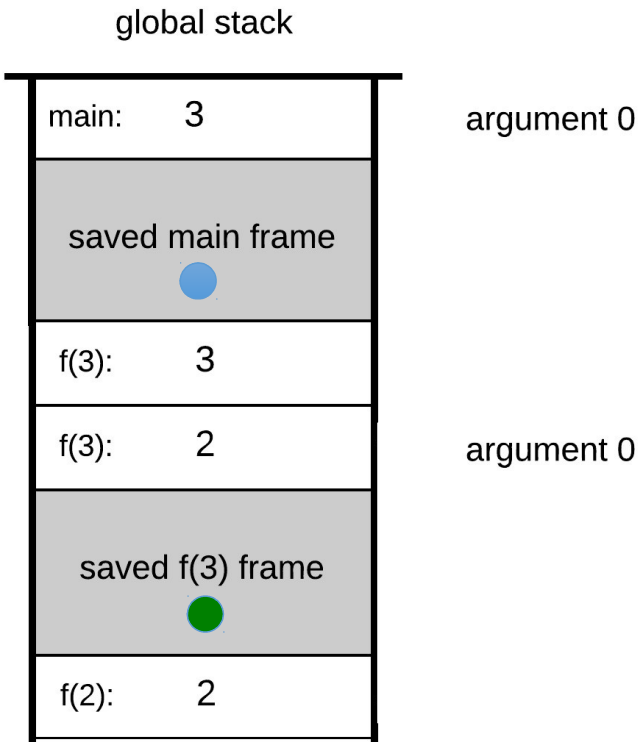
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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 (except the end result)

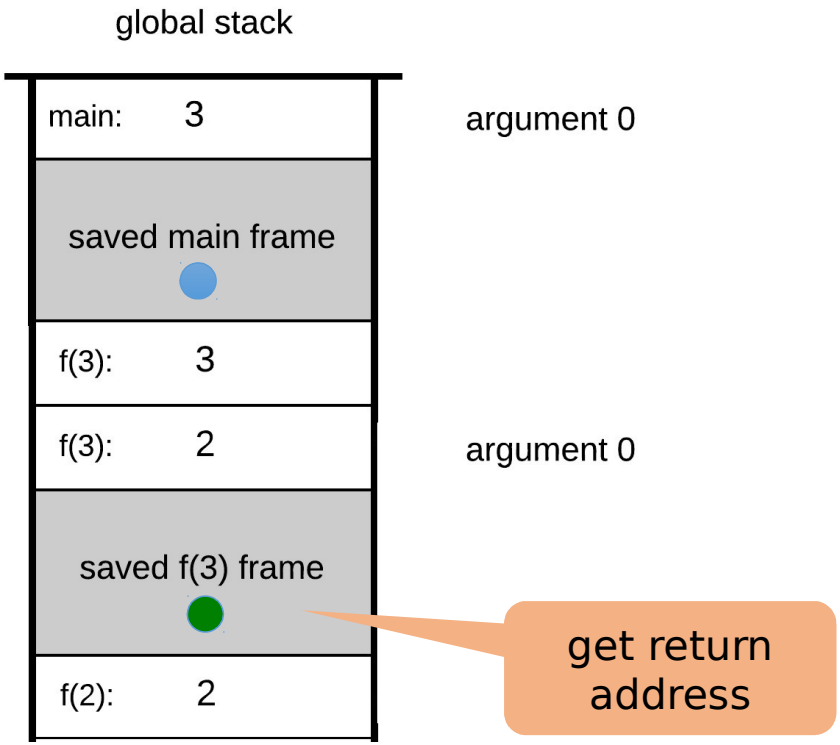


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```

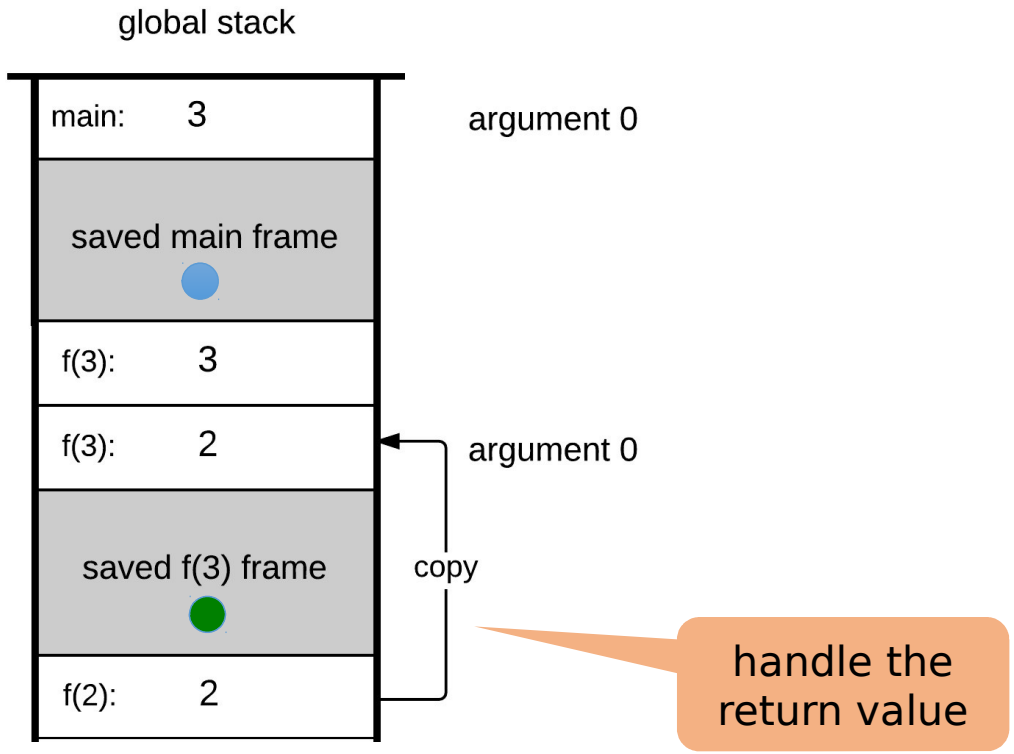


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



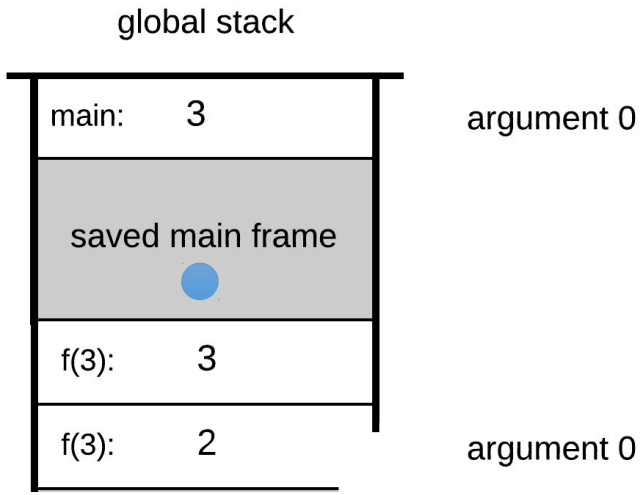
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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 (except the end result)



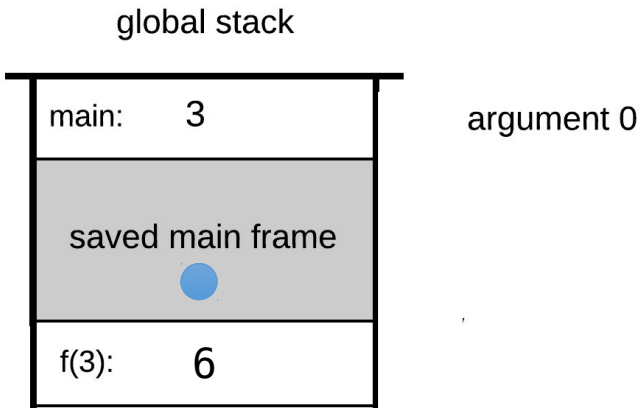
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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 (except the end result)

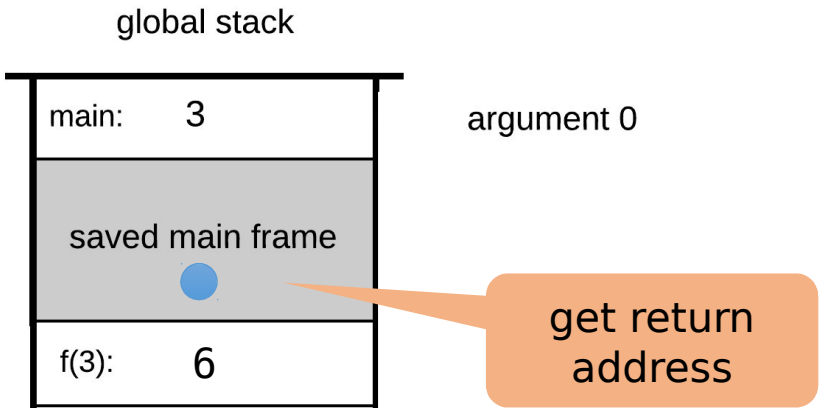


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



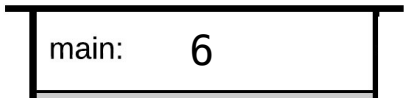
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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



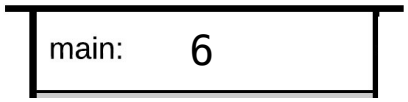
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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



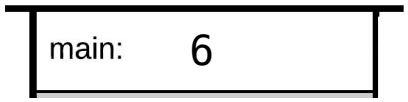
Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

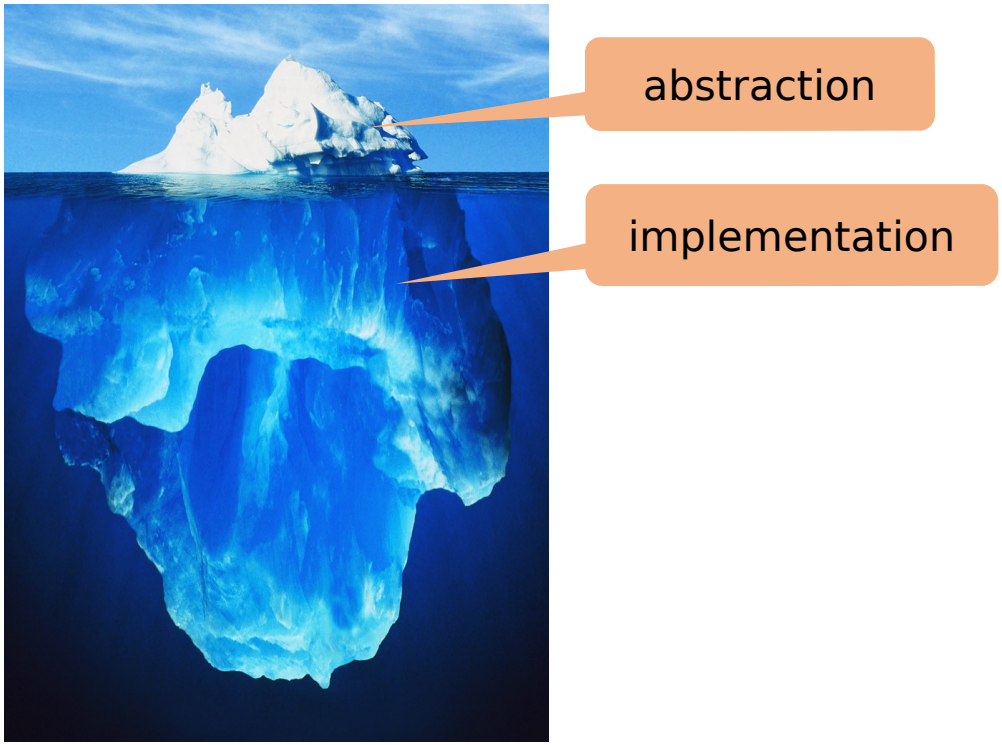
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   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...

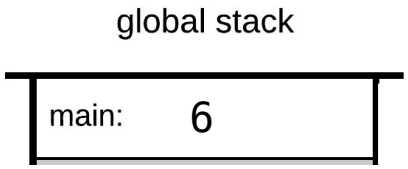


Function call and return: example factorial

UNIVERSITY OF LEEDS

VM program

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

function factorial 0
   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
```



The caller (main function) wanted to compute 3!

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

Since we are not going to implement this in our course I won't got into further details

However, the main idea should be clear!

VM translator: Hack assembly

UNIVERSITY OF LEEDS

```
@SP
AM=M+1
A=A-1
M=D
```

Push D

```
@mem_loc
D=M
@SP
AM=M+1
A=A-1
M=D
```

Push from memory location

```
@value
D=A
@SP
AM=M+1
A=A-1
M=D
```

Push value

PUSH

```
@SP
AM=M-1
D=M
M=D
```

Pop into D

```
@SP
AM=M-1
D=M
@mem_loc
D=M
```

Push into memory

POP

label *l-name*
(*l-name*)

LABEL

```
@label
0;JMP
```

GOTO

```
@SP
AM=M-1
D=M
@label
D;JNE
```

IF-GOTO

VM translator: Hack assembly

UNIVERSITY OF LEEDS

```
FRAME = LCL
RET = *(FRAME-5)
*ARG = pop()
SP = ARG+1
THAT = *(FRAME-1)
THIS = *(FRAME-2)
ARG = *(FRAME-3)
LCL = *(FRAME-4)
goto RET
```

return

```
push return-address
push LCL
push ARG
push THIS
push THAT
ARG = SP-n-5
LCL = SP
goto f
(return=address)
```

call f n

```
@k
D=-A
(label)
@SP
AM=M+1
A=A-1
M=0
@label
D=D+1;JLT
```

(k local variables initialized to 0 = Repeat k times push 0)

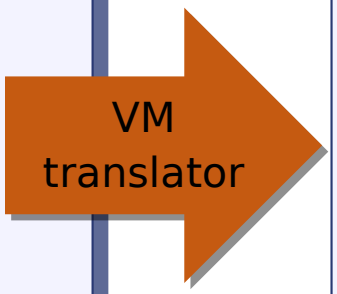
function *f k*

VM translator

UNIVERSITY OF LEEDS

VM code (example)

```
...
push constant 17
goto LOOP
...
call Foo.bar 3
...
function Foo.bar 2
...
    push local 4
return
...
```



Generated assembly code

```
...
//push constant 17
@ 17
D=A
... //additional assembly commands that complete the
      // implementation of push constant 17

//goto LOOP
... //generated assembly code that implements goto LOOP

//call Foo.bar 3
... //generated ... code that implements callFoo.bar3

//function Foo.bar 2
... //generated ... that implements function Foo.bar2

//push local4
... //generated ... that implements push local4

//return
... //generated assembly code that implements return

...
```

Source: VM language

UNIVERSITY OF LEEDS

Arithmetic / Logical commands

add
sub
neg
eq
gt
lt
and
or
not

Branching commands

label *label*
goto *label*
if-goto *label*

Function commands

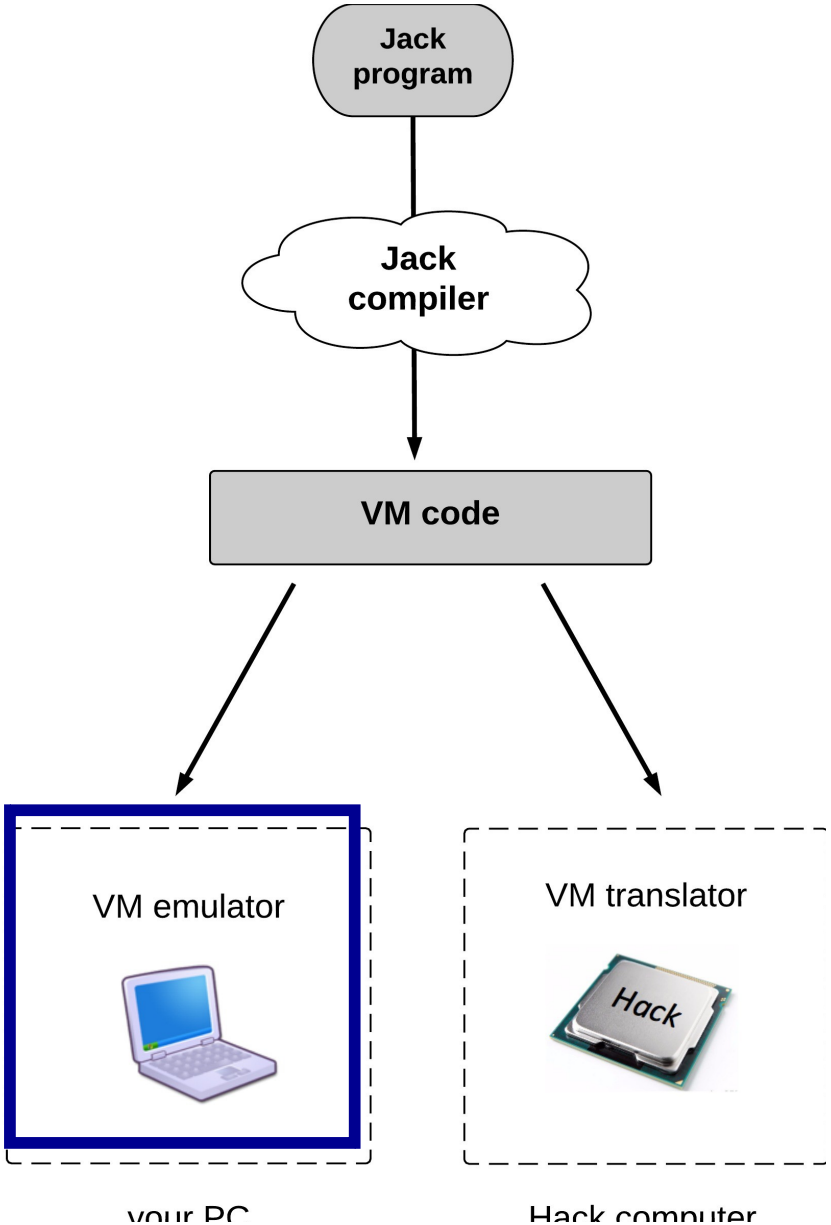
function *functionName nVars*
call *functionName nArgs*
return

Memory access commands

pop *segment i*
push *segment i*

VM emulator

UNIVERSITY OF LEEDS



Typical uses of the VM emulator:

- Running (compiled) Jack programs
- Experimenting with VM commands
- Observing the VM internals (stack, memory segments)
- Observing how the VM is realized on the host platform.