Chapter 8

Virtual Machine, Part II

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

Virtual machine: lecture plan

Overview

- Program control

Branching

- Abstraction
- Implementation

Functions

- Abstraction
- Implementation

Implementing function call-and-return:

- Implementation overview
- Run-time simulation
- Detailed implementation

VM implementation on the Hack platform:

- Standard mapping
- VM translator:
  proposed implementation
- Project 8 overview

Program control

$$x = -b + \sqrt{b^2 - 4 \cdot a \cdot c}$$

High-level expression:

```
x = -b + sqrt(power(b,2) - 4 * a * c)
```

Or:

```
x = -b + sqrt(disc(a,b,c))
```

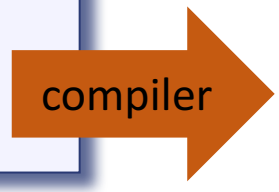
Functions:

- `sqrt`, `power`, and `disc` are *abstractions*
- The basic language can be extended *at will*.

Program control

High-level code

```
if !(a==0)
   x=(-b+sqrt(disc(a,b,c))/(2*a);
else
   x=-c/b;
// code continues
```



VM code (pseudo)

```
    push a
    push 0
    eq
    not
    if-goto A_NEQ_ZERO
    // We get here if a==0
    push c
    neg
    push b
    call div
    pop x
    goto CONTINUE
label A_NEQ_ZERO
    // We get here if !(a==0)
    push b
    neg
    push a
    push b
    push c
    call disc
    call sqrt
    add
    push 2
    push a
    call mult
    call div
    pop x
label CONTINUE
// code continues
```

Program control

High-level code

```
if !(a==0)
   x=(-b+sqrt(disc(a,b,c))/(2*a);
else
   x=-c/b;
// code continues
```

compiler

VM code (pseudo)

```
    push a
    push 0
    eq
    not
    if-goto A_NEQ_ZERO
    // We get here if a==0
    push c
    neg
    push b
    call div
    pop x
    goto CONTINUE
label A_NEQ_ZERO
    // We get here if !(a==0)
    push b
    neg
    push a
    push b
    push c
    call disc
    call sqrt
    add
    push 2
    push a
    call mult
    call div
    pop x
label CONTINUE
// code continues
```

Program control

VM branching commands:

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

VM function commands:

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

Challenges:

- Understanding what the commands do (abstraction)
- Realizing the commands on the host platform (implementation)

VM code (pseudo)

```
    push a
    push 0
    eq
    not
    if-goto A_NEQ_ZERO
    // We get here if a==0
    push c
    neg
    push b
    call div
    pop x
    goto CONTINUE
label A_NEQ_ZERO
    // We get here if !(a==0)
    push b
    neg
    push a
    push b
    push c
    call disc
    call sqrt
    add
    push 2
    push a
    call mult
    call div
    pop x
label CONTINUE
// code continues
```

Take Home Lessons

Understading how programs are executed:

- Branching
- Function call-and-return
- Recursion

Related implementation issues:

- Dynamic memory management
- Stack processing
- Pointers
- Completing the VM implementation.

Virtual machine: lecture plan

Overview

- Program control

Branching

- Abstraction
- Implementation

Functions

- Abstraction
- Implementation

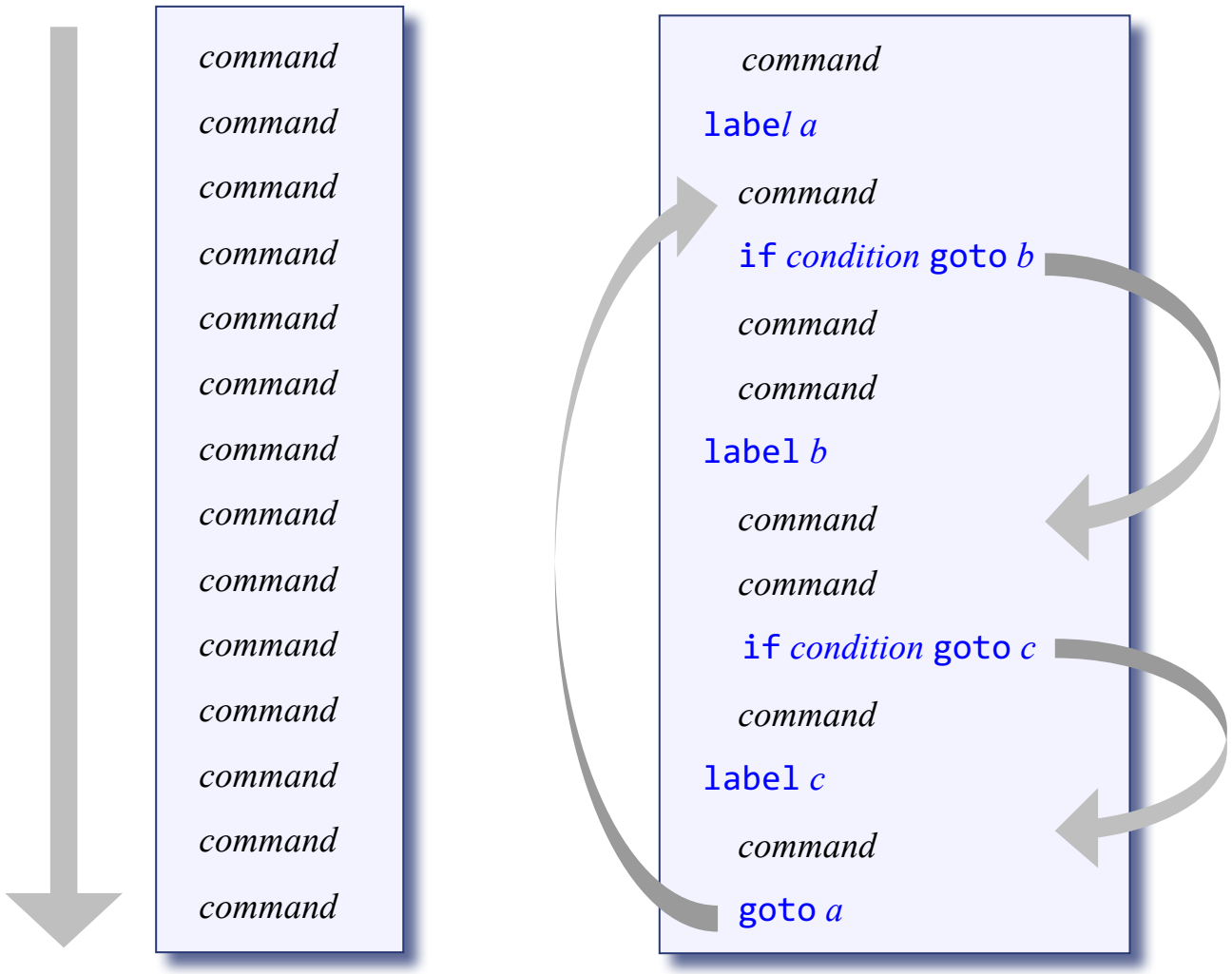
Implementing function call-and-return:

- Implementation overview
- Run-time simulation
- Detailed implementation

VM implementation on the Hack platform:

- Standard mapping
- VM translator:
  proposed implementation
- Project 8 overview

Branching



Branching:

- Unconditional
- Conditional

Branching

High-level program

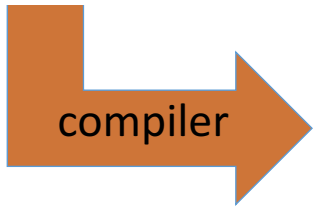
```
// Returns x * y
int mult(int x, int y) {
   int sum = 0;
   int n = 1;
   // sum = sum + x, y times
   while !(n > y) {
      sum += x;
      n++;
   }
   return sum;
}
```

Let's focus on the while logic

Branching

High-level program

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



Pseudo VM code

```
function mult(x,y)
   push 0
   pop sum
   push 1
   pop n
label WHILE_LOOP
   push n
   push y
   gt
   if-goto ENDLOOP
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto WHILE_LOOP
label ENDLOOP
   push sum
   return
```

Unconditional branching:

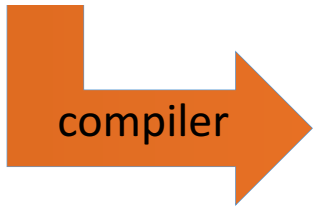
`goto` *label*

Jumps to execute the command just after *label*.

Branching

High-level program

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



Pseudo VM code

```
function mult(x,y)
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto ENDLOOP
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto WHILE_LOOP
label ENDLOOP
   push sum
   return
```

Conditional branching:

`if-goto` *label*

VM logic:

1. *cond* = pop;
2. if *cond* jump to execute the command just after *label*.

(Requires pushing the condition to the stack before the `if-goto` command)

Branching

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.

VM language

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 segment commands

`pop` *segment i*
`push` *segment i*

Functions

High-level programming languages can be extended using:

- Subroutines
- Functions
- Procedures
- Methods
- Etc.

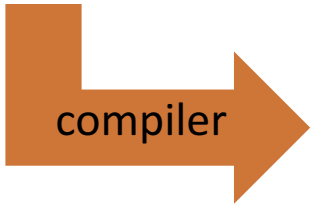
} functions

(different names
of the same thing)

Functions in the VM language

High-level program

```
...
sqrt(x – 17 + x * 5)
...
```



Pseudo VM code

```
...
push x
push 17
sub
push x
push 5
call Math.multiply
add
call Math.sqrt
...
```

The VM language features:

- primitive operations (fixed): `add`, `sub`, ...
- abstract operations (extensible): `multiply`, `sqrt`, ...

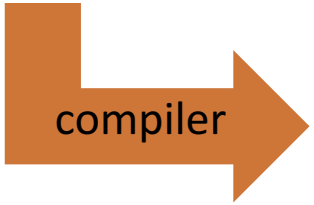
Programming style:

- Applying a primitive operator or calling a function have the same look-and-feel.

Functions in the VM language: defining

High-level program

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



Pseudo VM code

```
function mult(x,y)
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto END
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto LOOP
label END
   push sum
   return
```

Final VM code

```
function mult 2     // 2 local vars.
   push constant 0  // sum=0
   pop local 0
   push constant 1  // n=1
   pop local 1
label LOOP
   push local 1     // if !(n>y)
   push argument 1  // goto END
   gt
   if-goto END
   push local 0     // sum+=x
   push argument 0
   add
   pop local 0
   push local 1     // n++
   push constant 1
   add
   pop local 1
   goto LOOP
label END
   push local 0    // return sum
   return
```

Functions in the VM language: executing

```
// Computes 3+5*8
 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

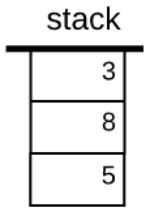
```
// Computes the product of two given arguments
 0 function mult 2
 1   push constant 0
 2   pop local 0
 3   push constant 1
 4   pop local 1
 5 label LOOP
 6   push local 1
 7   push argument 1
     //... computes the product into local 0
19 label END
20   push local 0
21   return
```

(same code as previous slide, with line numbers, for easy reference)

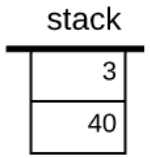
callee

main view:

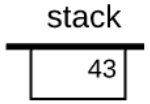
after line 3 is executed:



after line 4 is executed:



after line 5 is executed:



Magic!

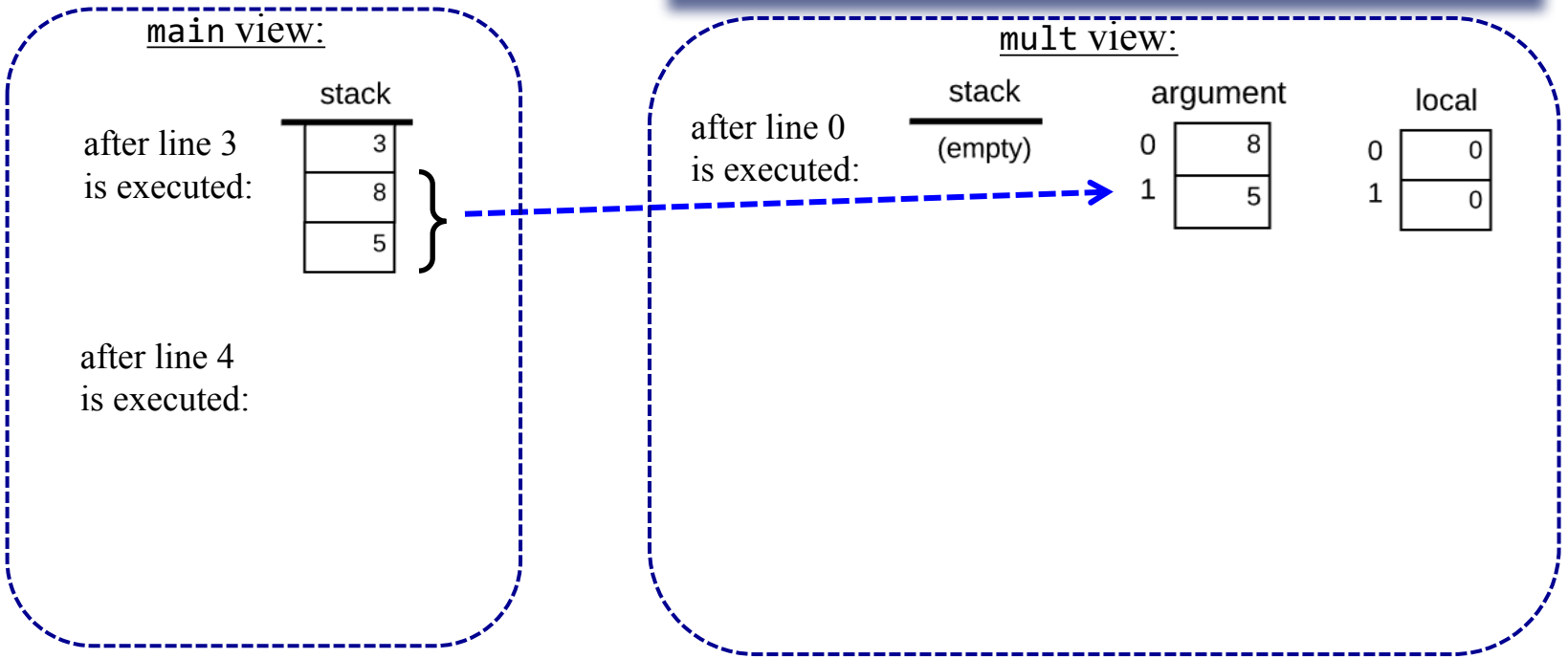
Functions in the VM language: executing

```
// Computes 3+5*8
 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



Functions in the VM language: executing

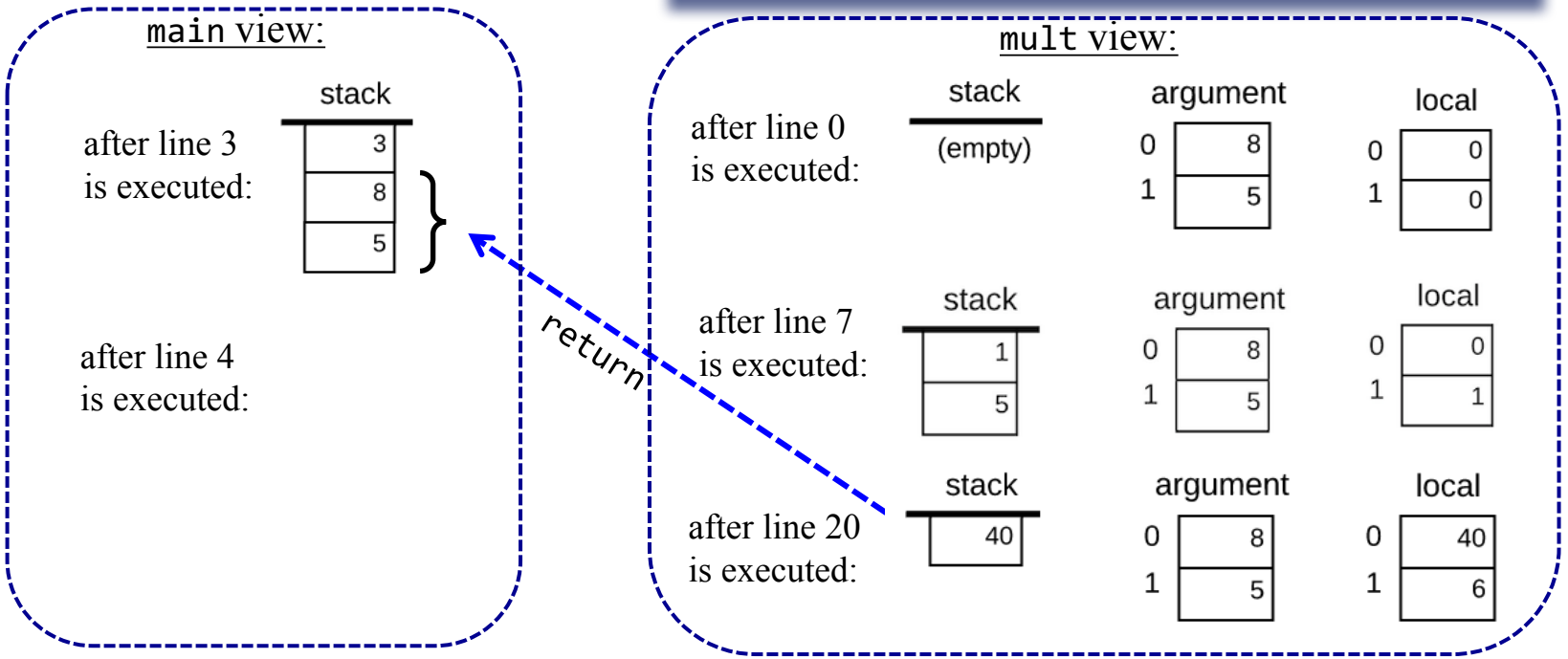
```
// Computes 3+5*8
 0 function main 0
 1   push constant 3
 2   push constant 8
 3   push constant 5
 4   call mult 2
 5   add
 6   return
```

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
```

(same code as previous slide, with line numbers, for easy reference)

callee



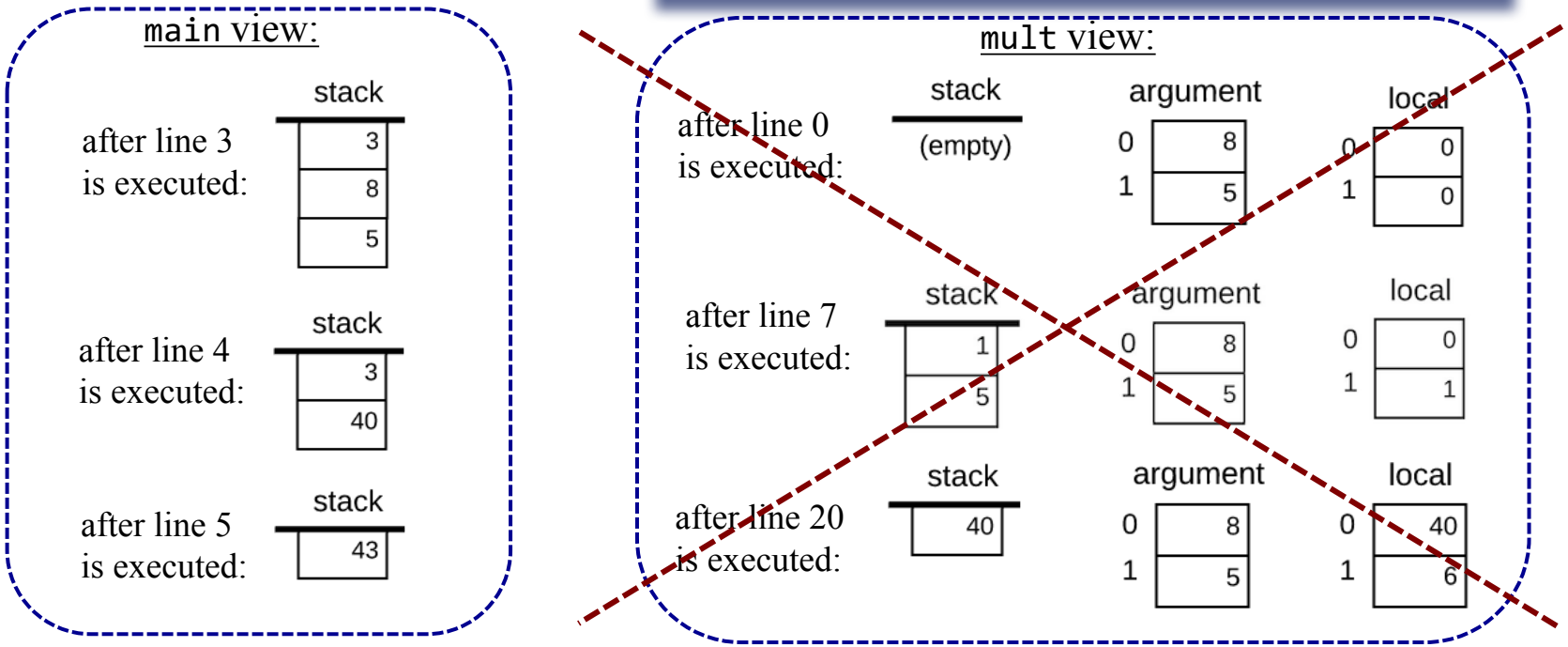
Functions in the VM language: executing

```
// Computes 3+5*8
 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
22   return
```

callee



Functions in the VM language: implementation

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

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Implementation

How to orchestrate the drama just described?

We can write low-level code that manages the parameter passing, the saving and re-instating of function states, etc.

This task can be realized by writing code that...

- Handles the VM command `call`
- Handles the VM command `function`
- Handles the VM command `return`.

Functions in the VM language: implementation

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

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Handling `call`:

- Determine the return address within the caller's code;
- Save the caller's return address, stack and memory segments;
- Pass parameters from the caller to the callee;
- Jump to execute the callee.

Functions in the VM language: implementation

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

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Handling `function`:

- Initialize the local variables of the callee;
- Handle some other simple initializations (later).

Functions in the VM language: implementation

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

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Handling `return`:

(a function always ends by pushing a return value on the stack)

- Return the *return value* to the caller;
- Recycle the memory resources used by the callee;
- Reinstate the caller's stack and memory segments;
- Jump to the return address in the caller's code.

Virtual machine: lecture plan

Overview

- Program control

Branching

- Abstraction
- Implementation

Functions

- Abstraction
- Implementation

Implementing function call-and-return:

- Implementation overview
- Run-time simulation
- Detailed implementation

VM implementation on the Hack platform:

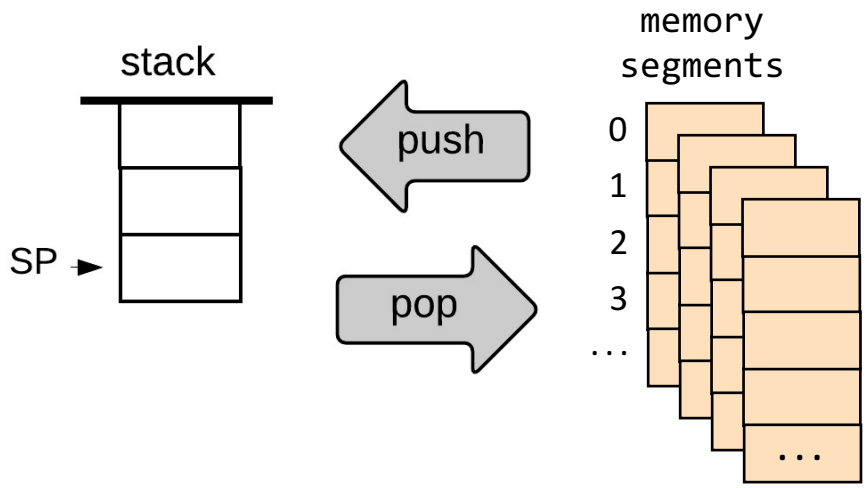
- Standard mapping
- VM translator:
  proposed implementation
- Project 8 overview

Function execution

- 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*

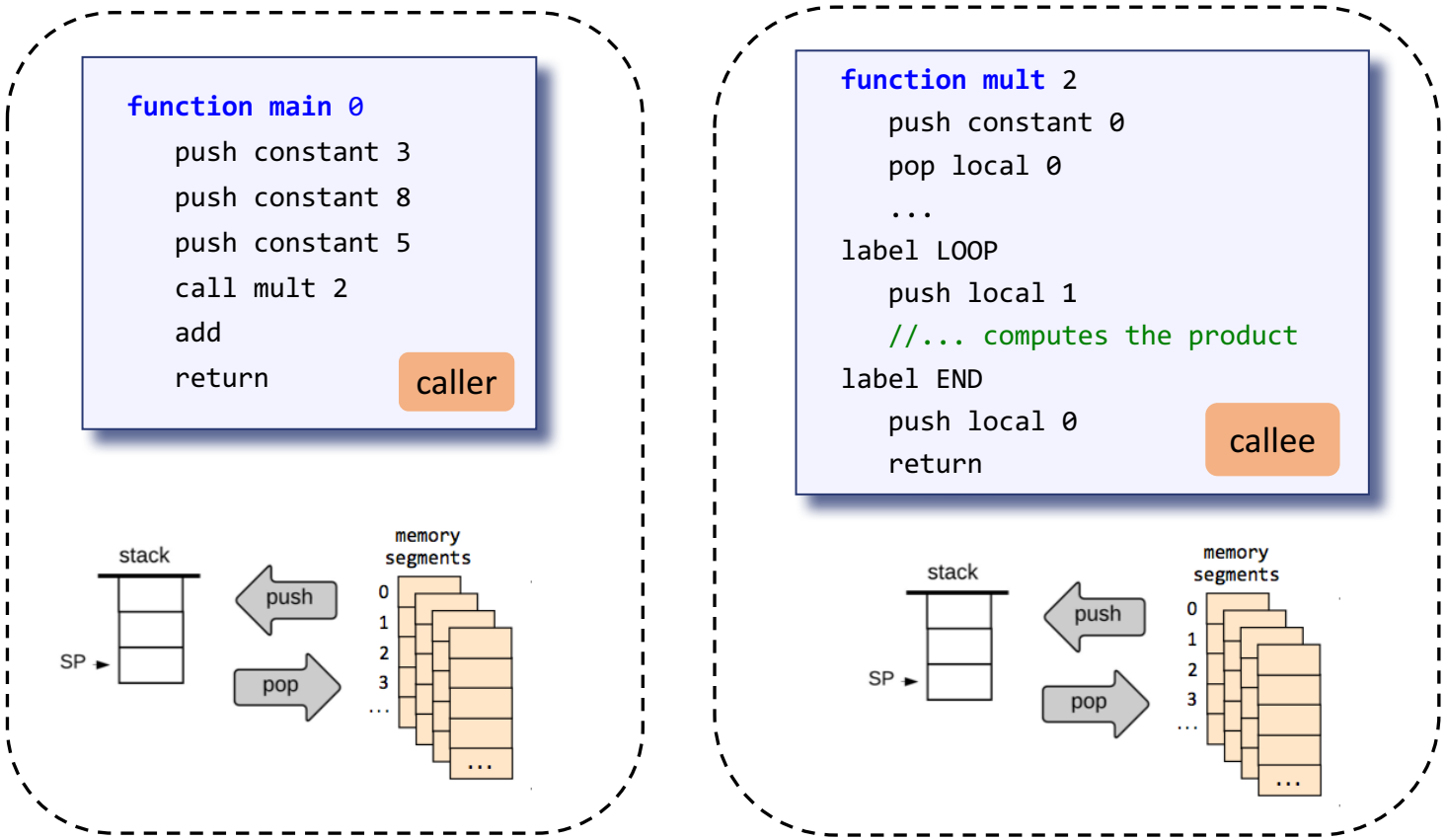
The function's state



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

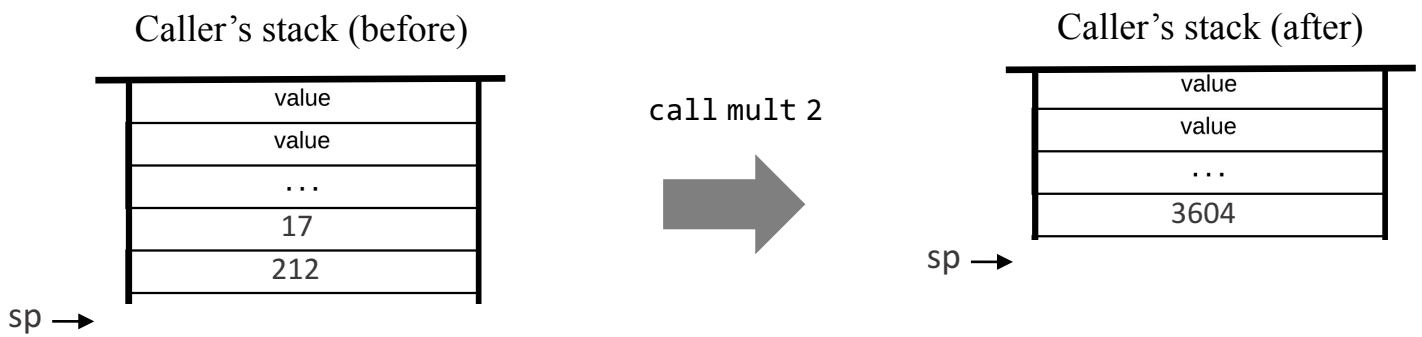


Challenge:

- 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)`

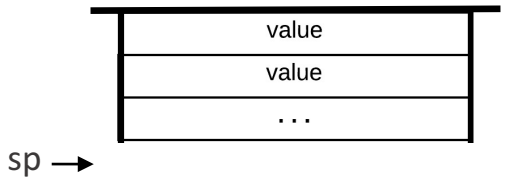


Net effect:

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

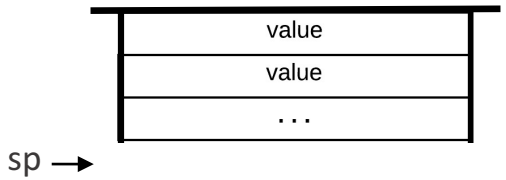
Function call and return: implementation

The function is running,
doing something



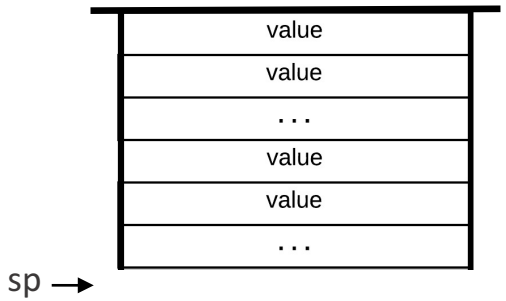
Function call and return: implementation

The function prepares
to call another function:



Function call and return: implementation

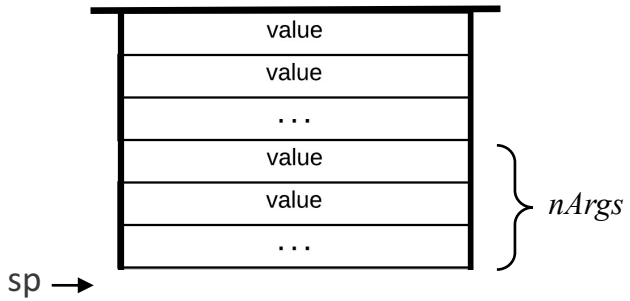
The function prepares
to call another function:



Function call and return: implementation

The function says:

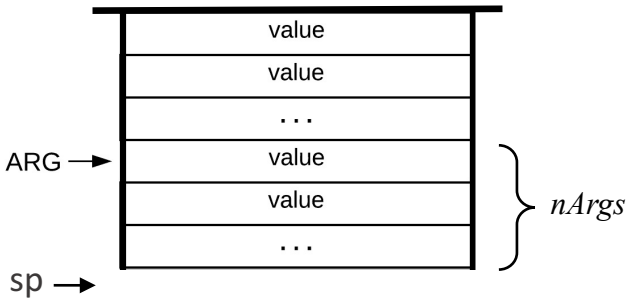
`call` *foo nArgs*



Function call and return: implementation

The function says:

`call` *foo nArgs*



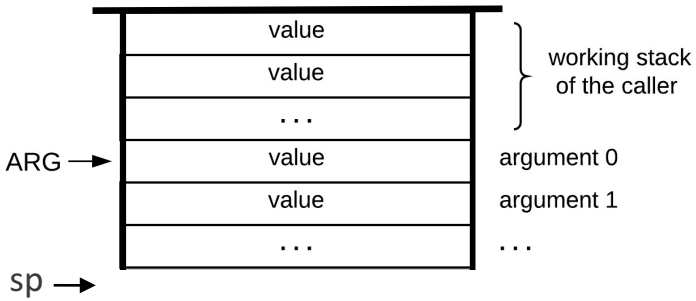
VM implementation (handling `call`):

1. Sets `ARG`

Function call and return: implementation

The function says:

`call` *foo nArgs*



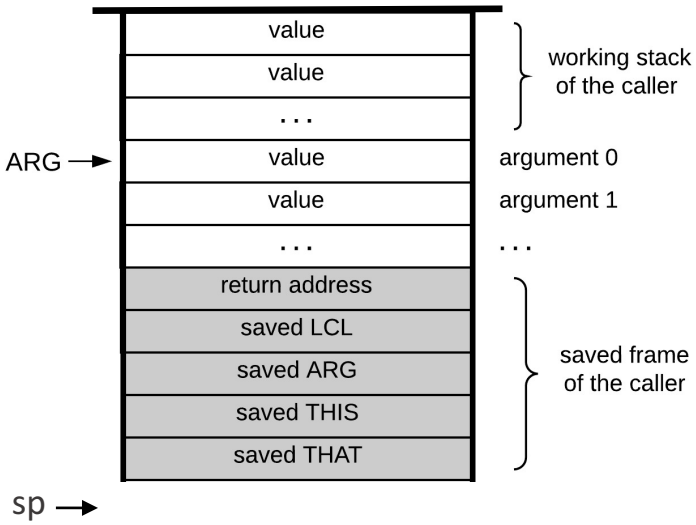
VM implementation (handling `call`):

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

Function call and return: implementation

The function says:

`call` *foo nArgs*



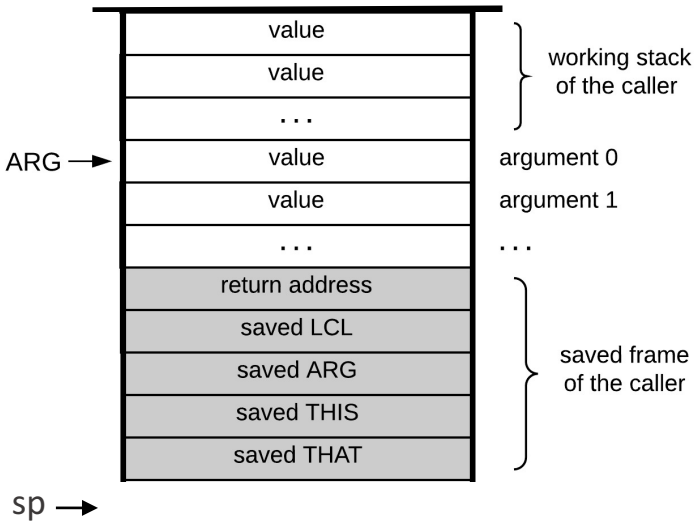
VM implementation (handling `call`):

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

Function call and return: implementation

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*

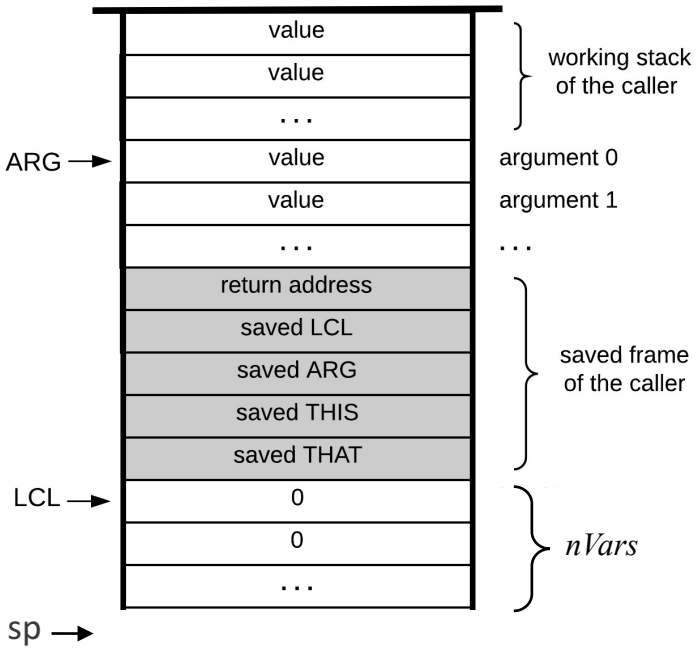
VM implementation (handling `function`):

Sets up the `local` segment of the called function

Function call and return: implementation

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*

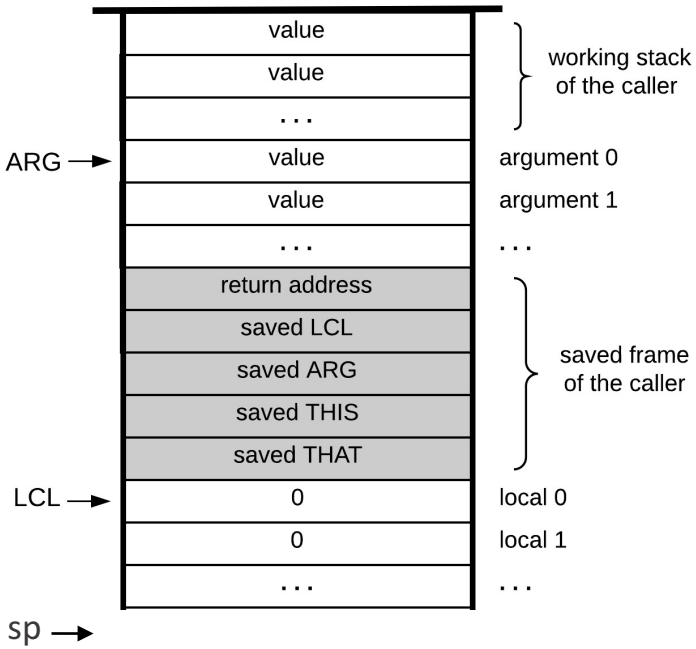
VM implementation (handling `function`):

Sets up the `local` segment of the called function

Function call and return: implementation

The called function is entered:

`function` *foo nVars*



VM implementation (handling `call`):

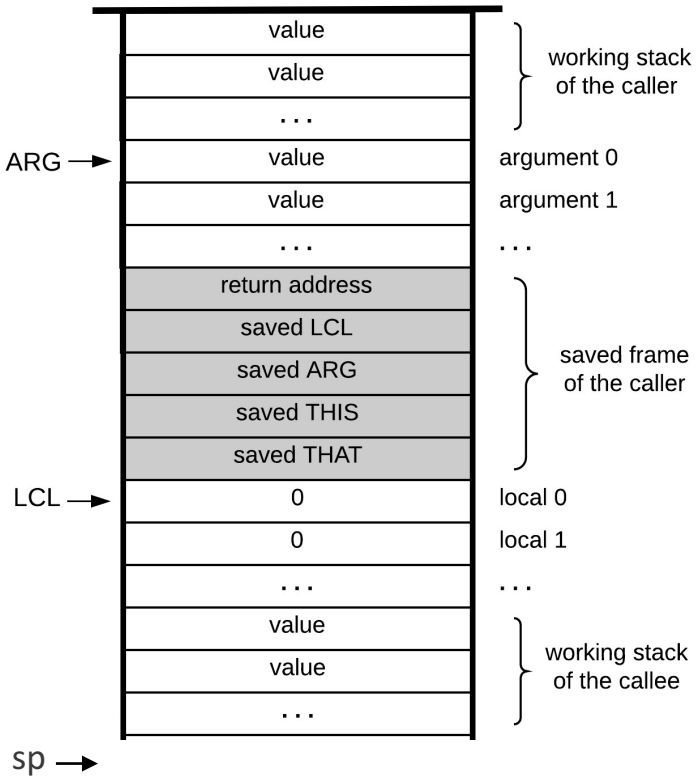
1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling `function`):

Sets up the `local` segment of the called function

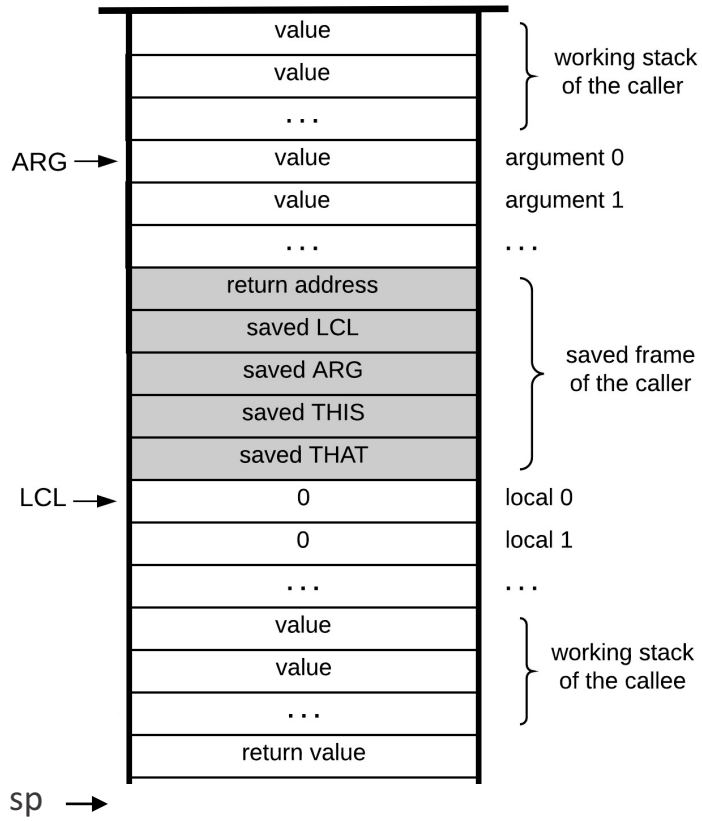
Function call and return: implementation

The called function is running,
doing something



Function call and return: implementation

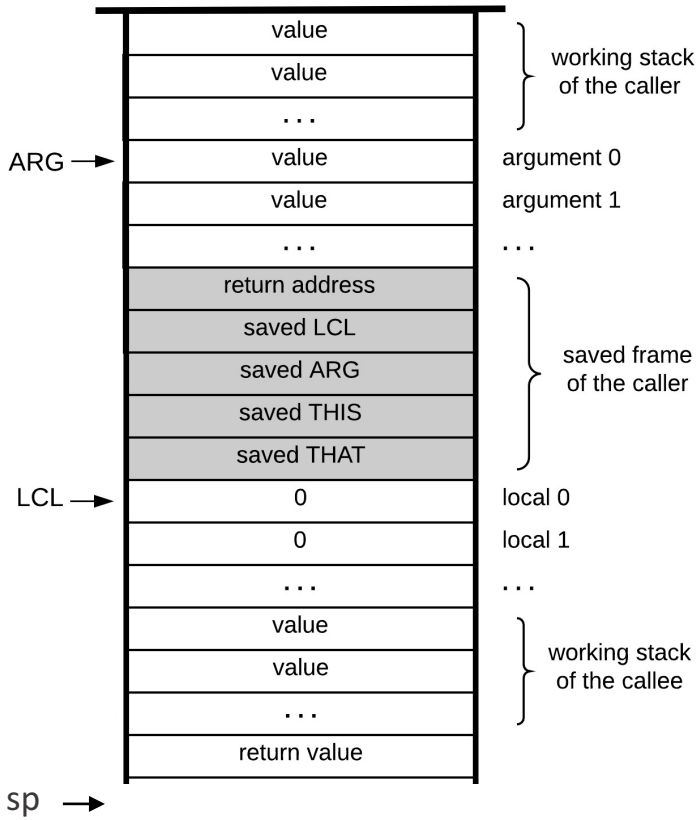
The called function prepares to return:
it pushes a *return value*



Function call and return: implementation

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`):

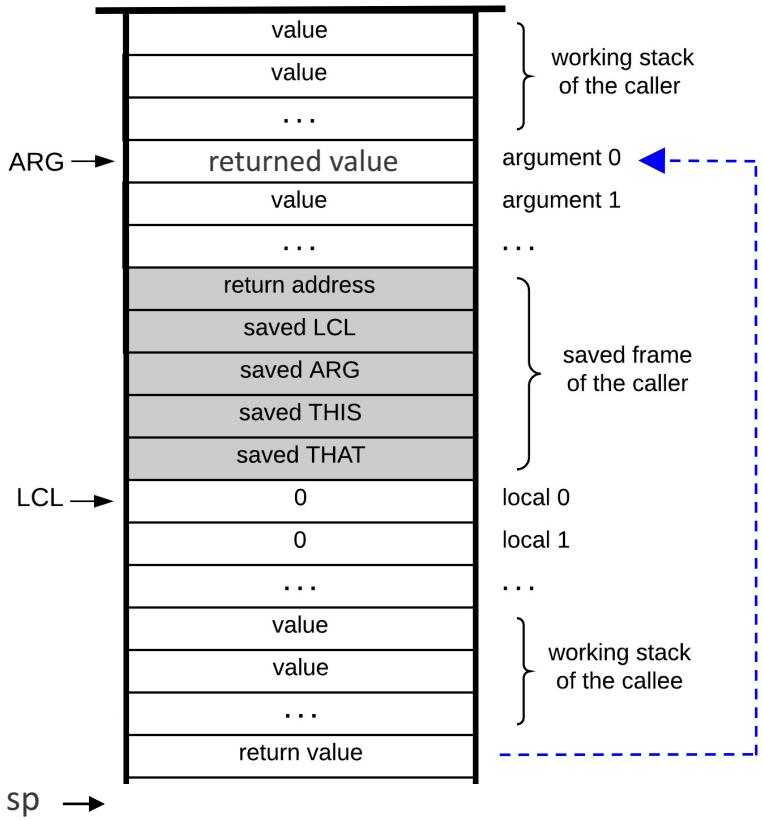
Sets up the `local` segment of the called function

VM implementation (handling `return`):

Function call and return: implementation

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

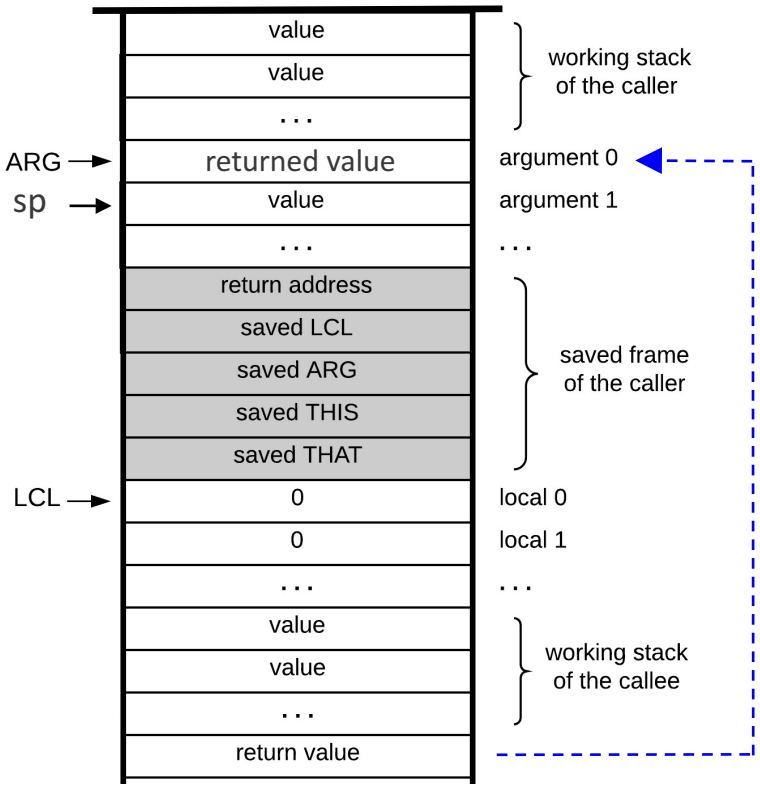
VM implementation (handling `return`):

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

Function call and return: implementation

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

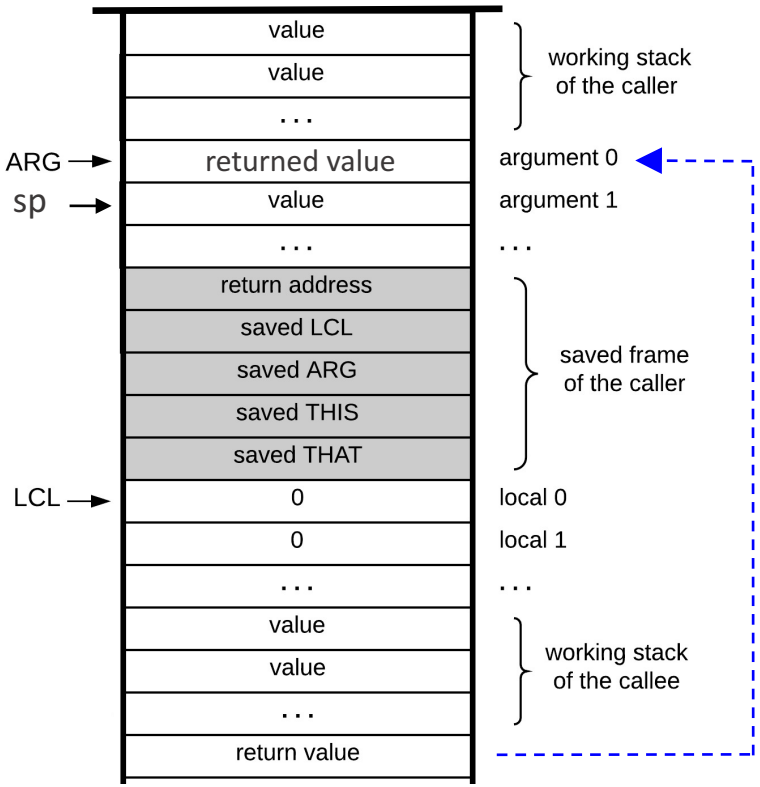
VM implementation (handling `return`):

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

Function call and return: implementation

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

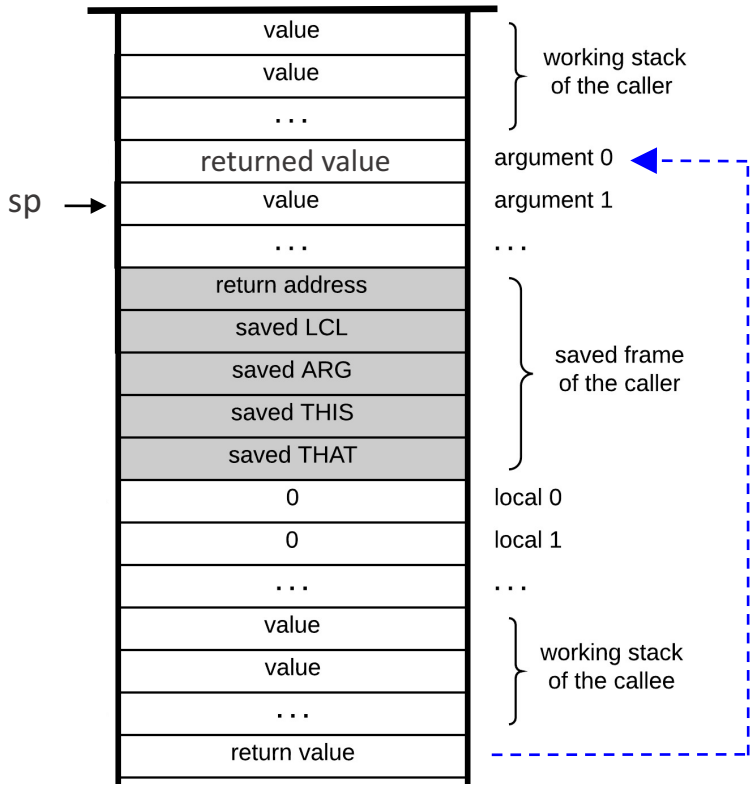
VM implementation (handling `return`):

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

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`):

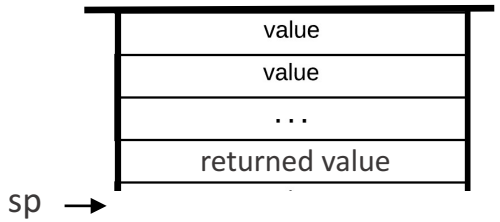
Sets up the `local` segment of the called function

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

The caller
resumes its execution



VM implementation (handling `call`):

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

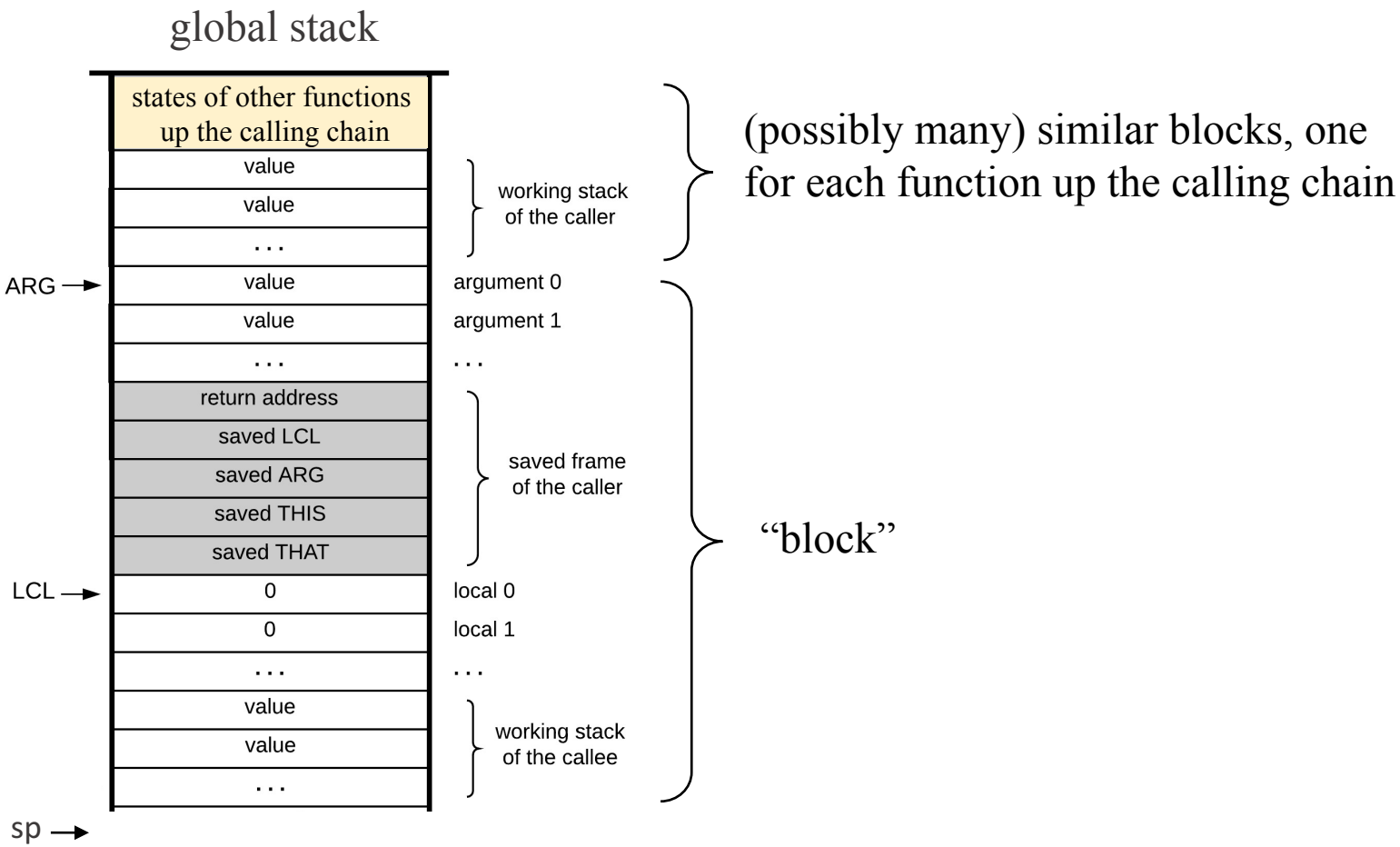
VM implementation (handling `return`):

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)

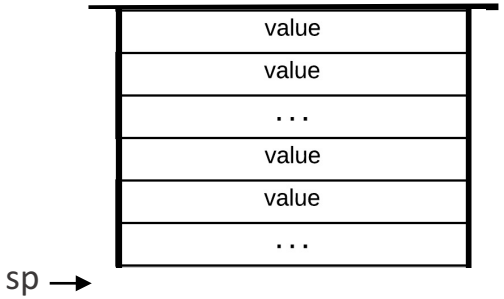
The global stack



Recap: function call and return

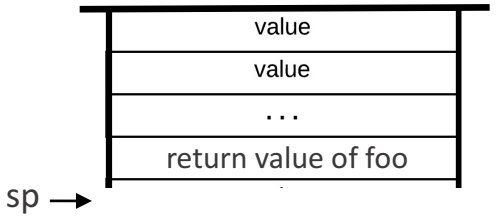
The caller says:

`call` *foo nArgs*

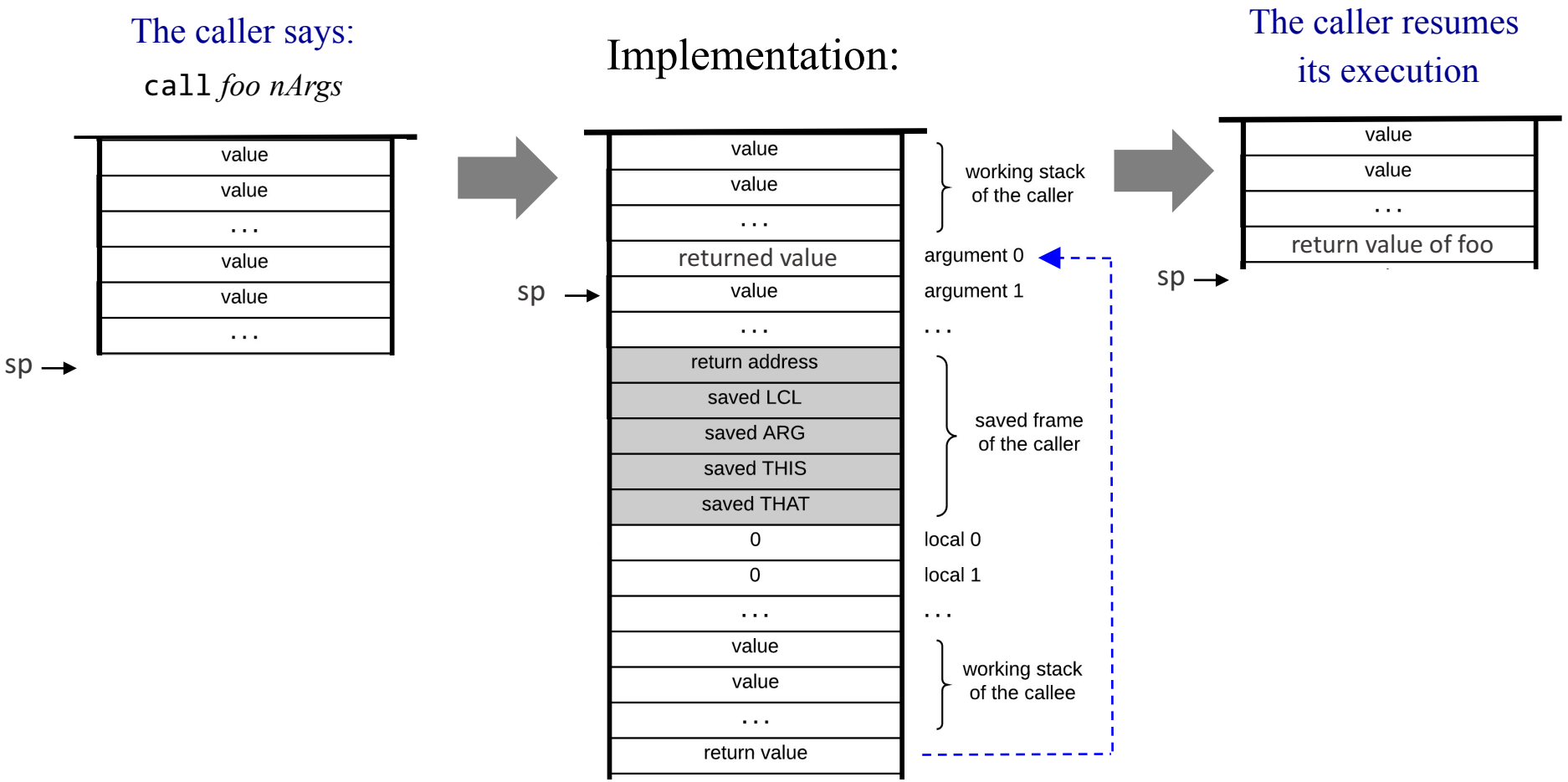


Abstraction:

The caller resumes its execution



Recap: function call and return



Any sufficiently advanced technology is indistinguishable from magic.

—Arthur C. Clarke (1962)

Virtual machine: lecture plan

Overview

- Program control

Branching

- Abstraction
- Implementation

Functions

- Abstraction
- Implementation

Implementing function call-and-return:

- Implementation overview
- Run-time simulation
- Detailed implementation

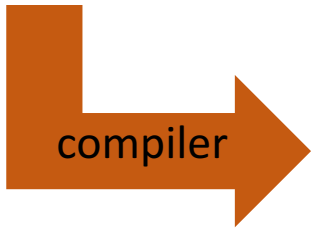
VM implementation on the Hack platform:

- Standard mapping
- VM translator:
  proposed implementation
- Project 8 overview

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);
}
```



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
```

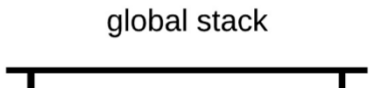
VM program

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

function factorial 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
```

Run-time example

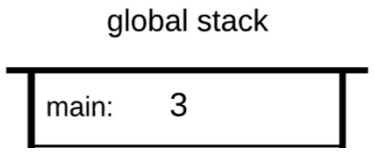
VM program

```
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
```



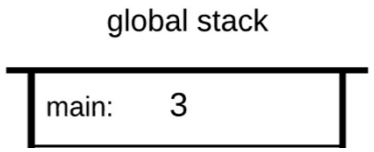
Run-time example

```
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
```



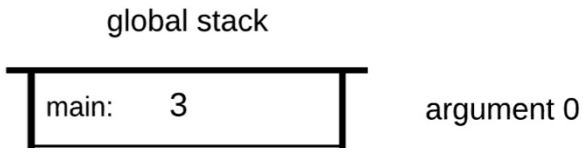
Run-time example

```
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
```



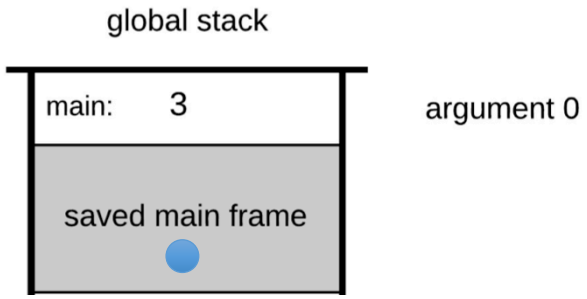
Run-time example

```
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
```



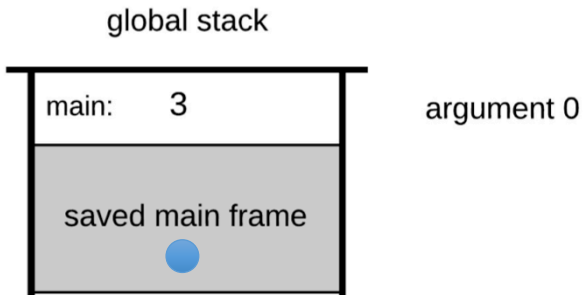
Run-time example

```
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
```



Run-time example

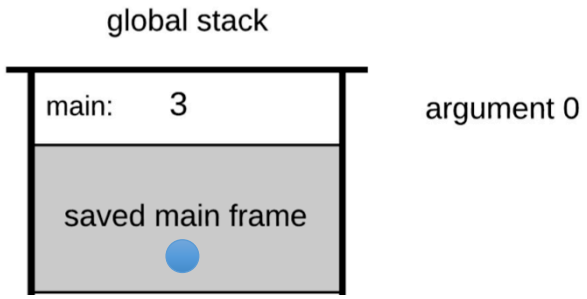
```
function main 0
   push constant 3
   call factorial 1
   return

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
```



Run-time example

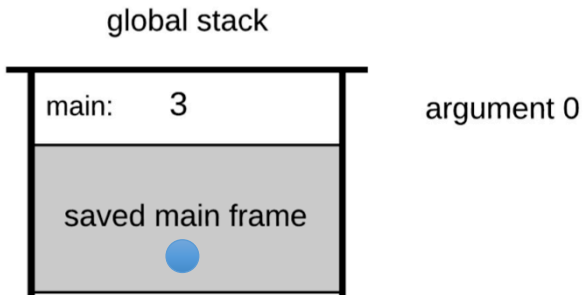
```
function main 0
   push constant 3
   call factorial 1
   return

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



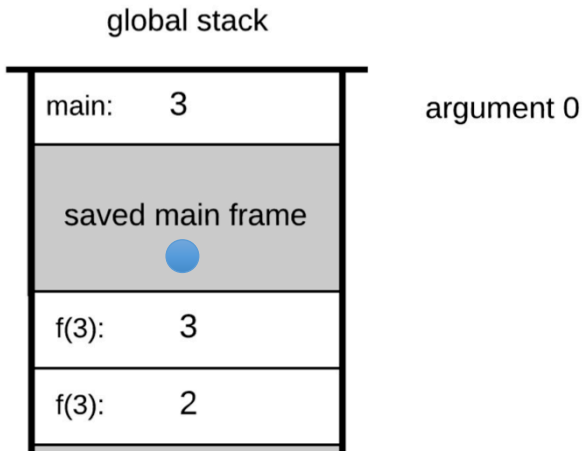
Run-time example

```
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
```



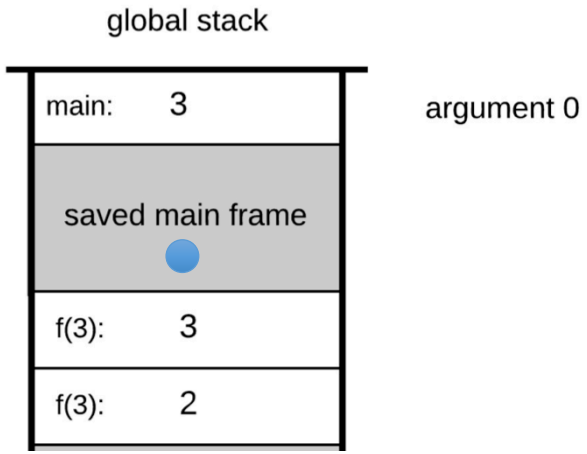
Run-time example

```
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
```



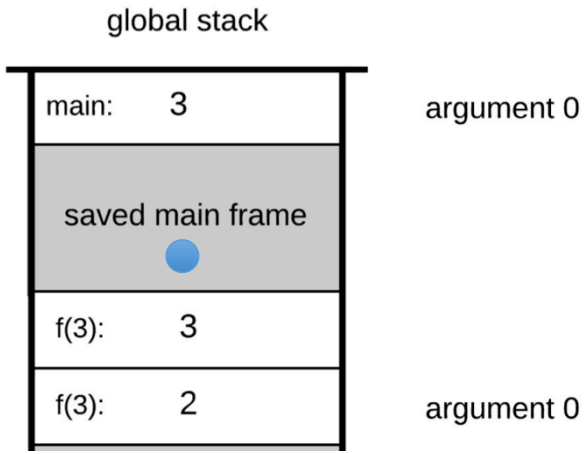
Run-time example

```
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
```



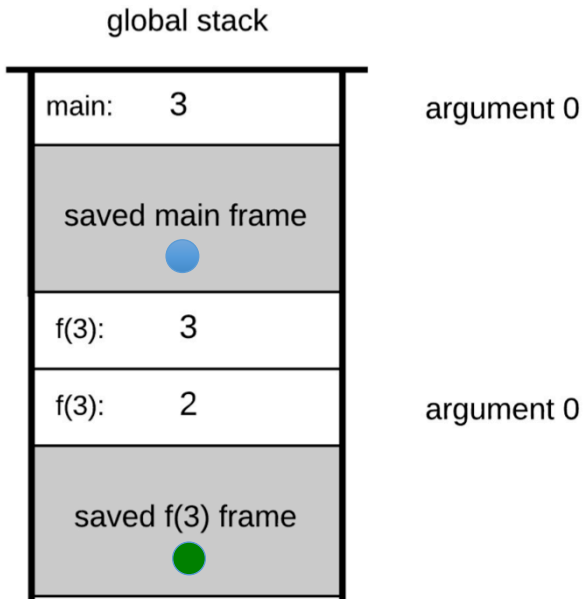
Run-time example

```
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
```



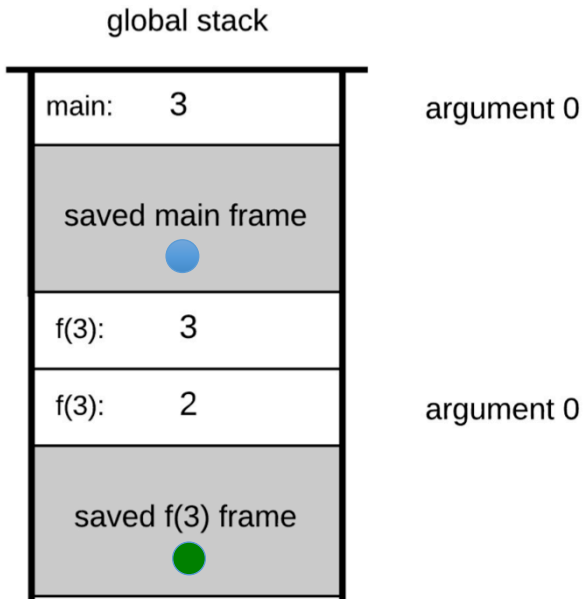
Run-time example

```
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
```



Run-time example

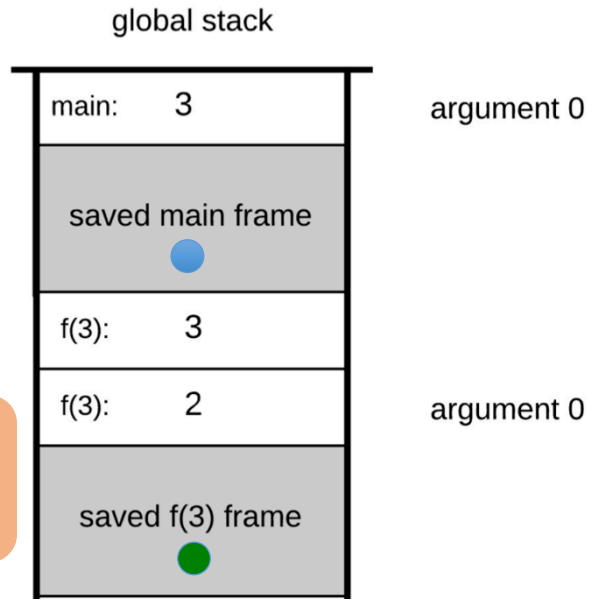
```
function main 0
   push constant 3
   call factorial 1
   return

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
```



Run-time example

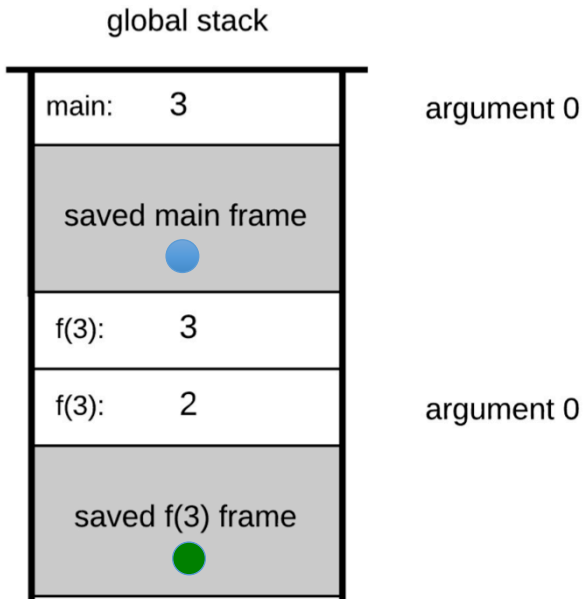
```
function main 0
   push constant 3
   call factorial 1
   return

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



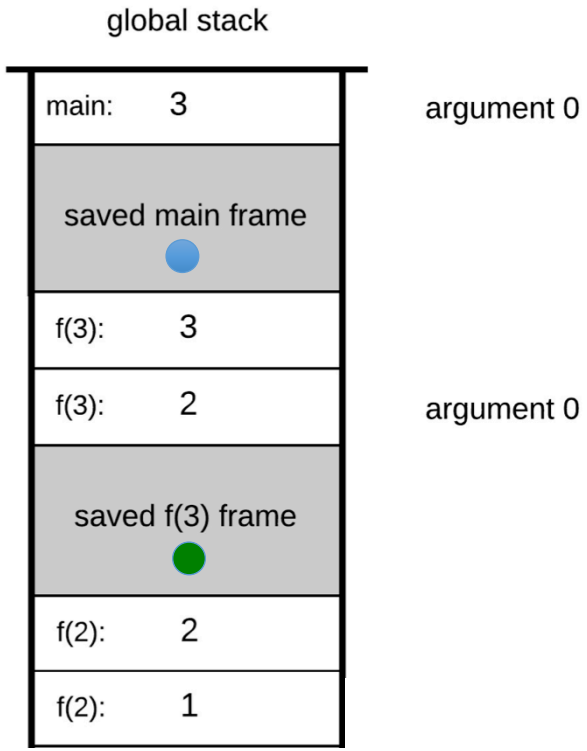
Run-time example

```
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
```



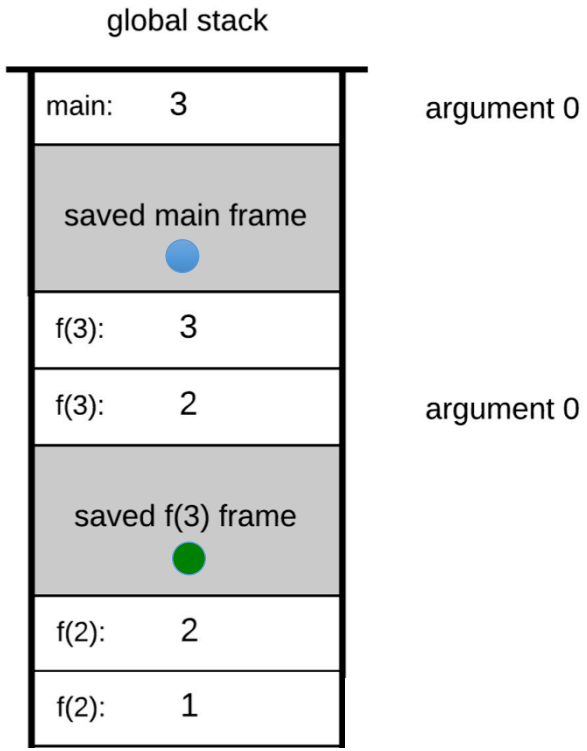
Run-time example

```
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
```



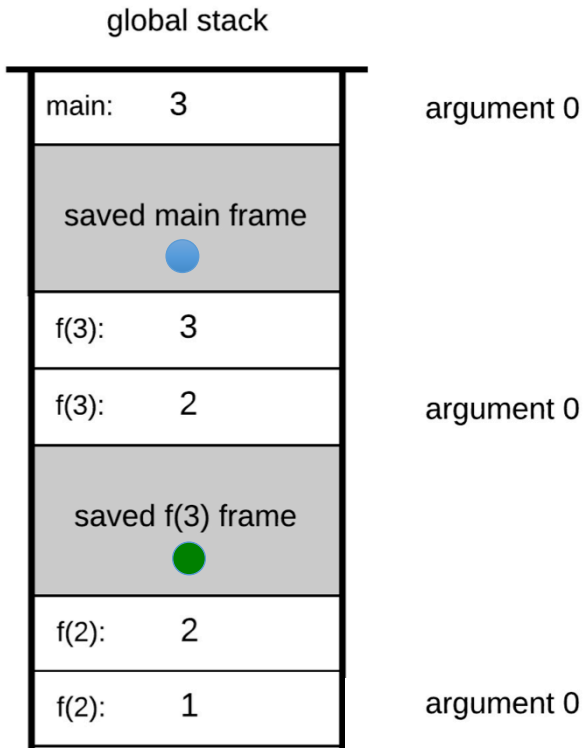
Run-time example

```
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
```



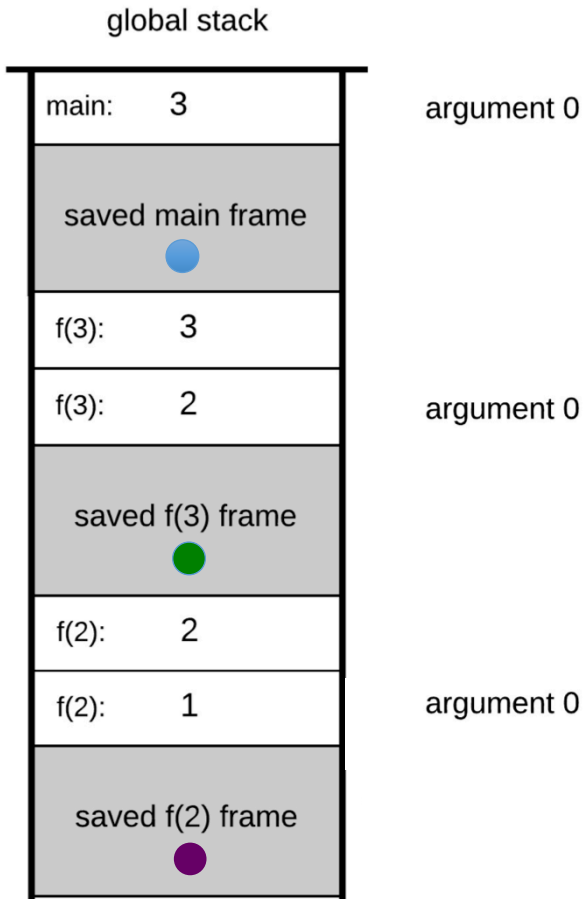
Run-time example

```
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
```



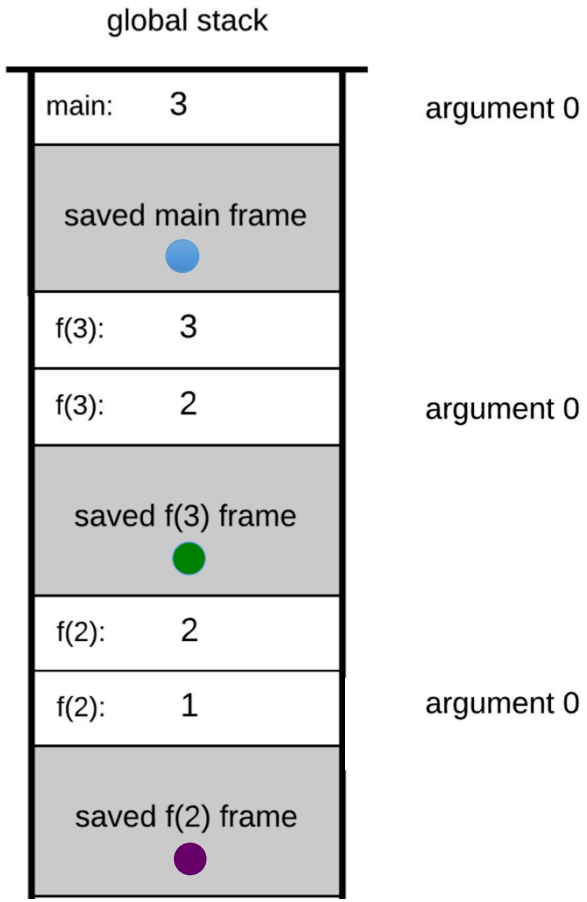
Run-time example

```
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
```



Run-time example

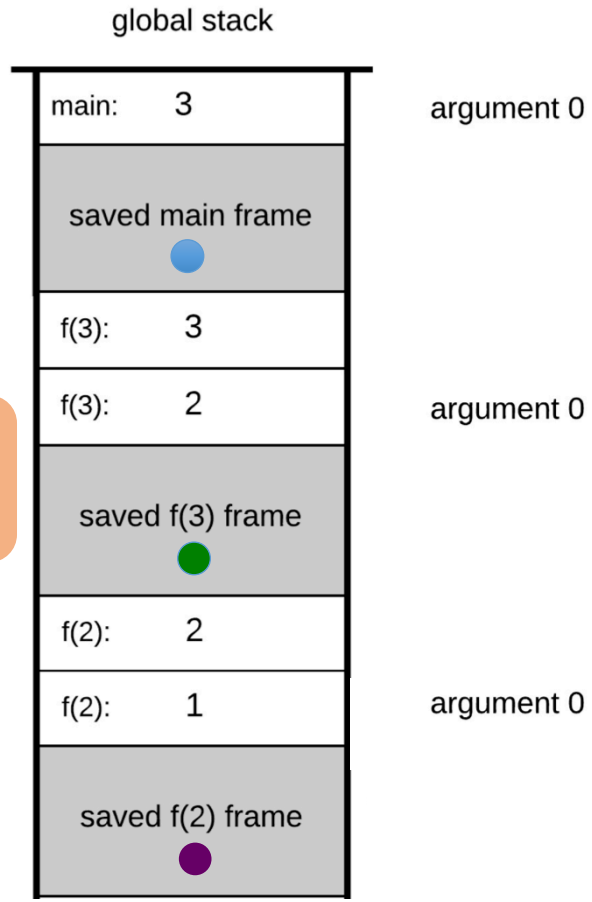
```
function main 0
   push constant 3
   call factorial 1
   return

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
```



Run-time example

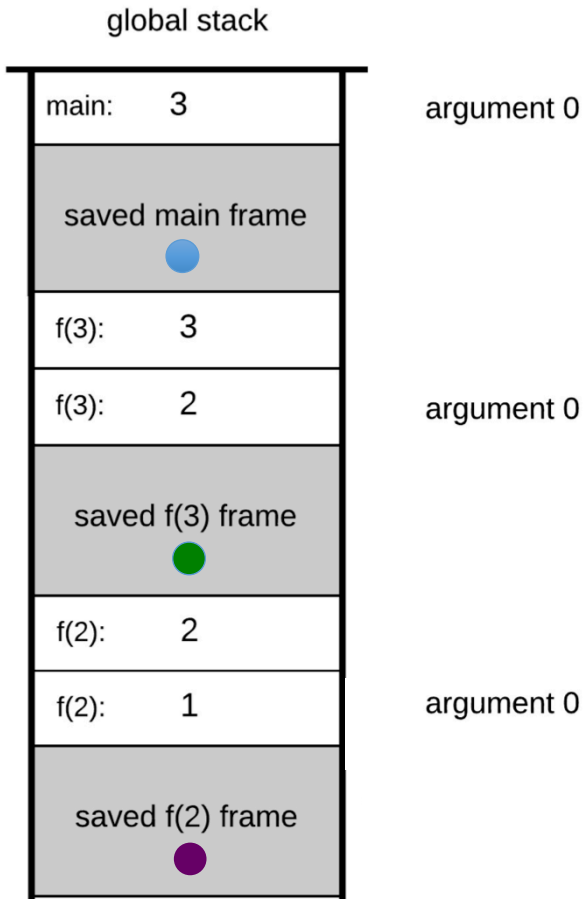
```
function main 0
   push constant 3
   call factorial 1
   return

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



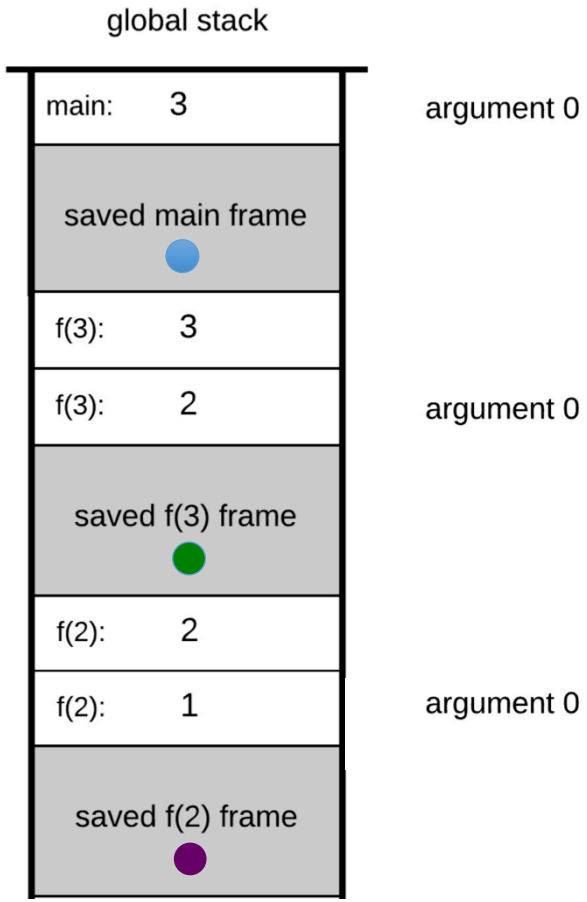
Run-time example

```
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
```



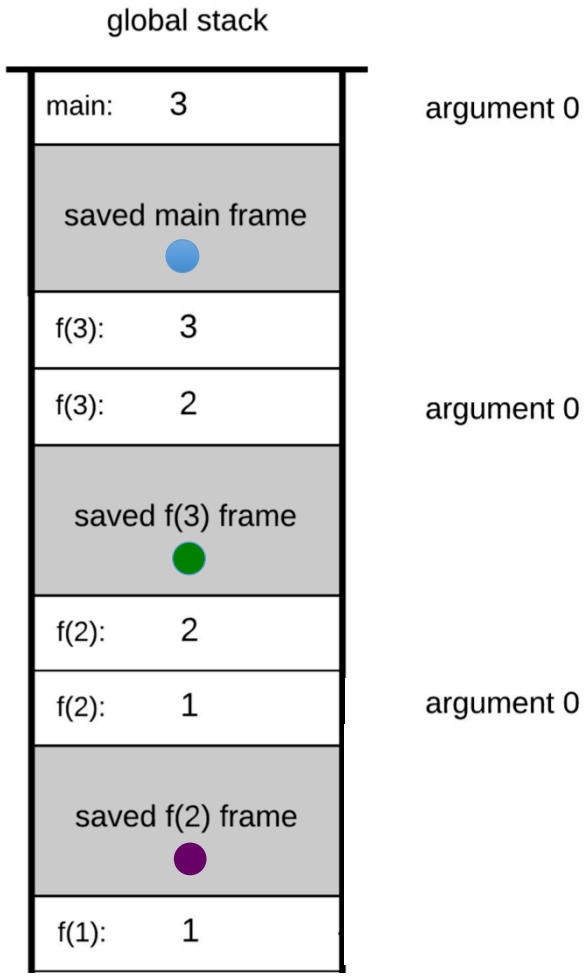
Run-time example

```
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
```



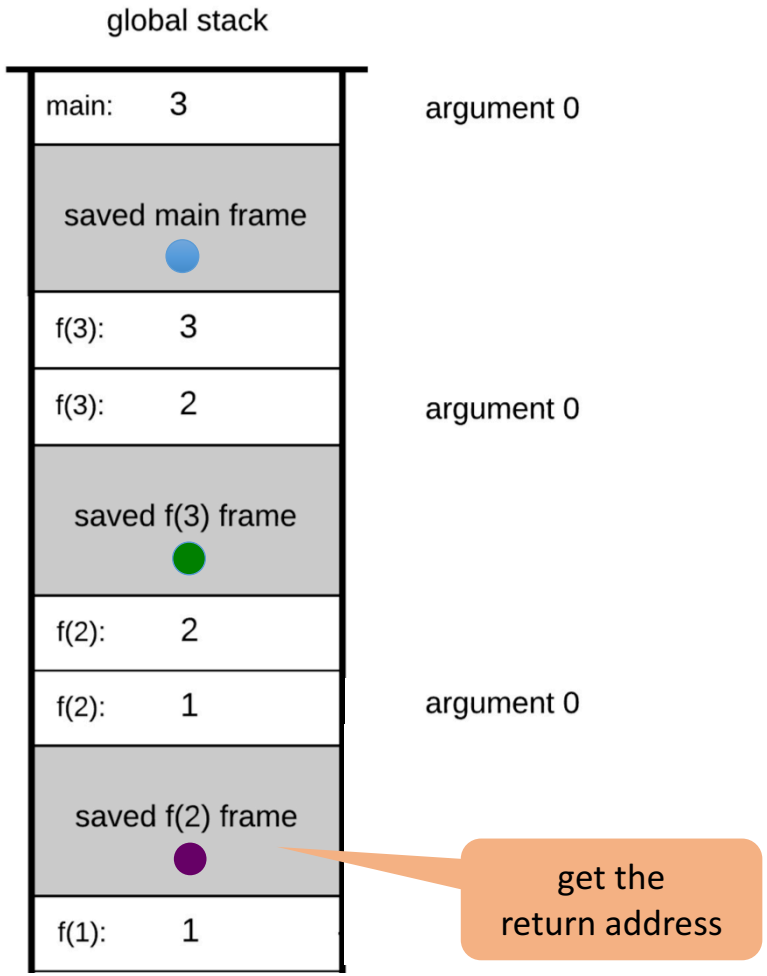
Run-time example

```
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
```



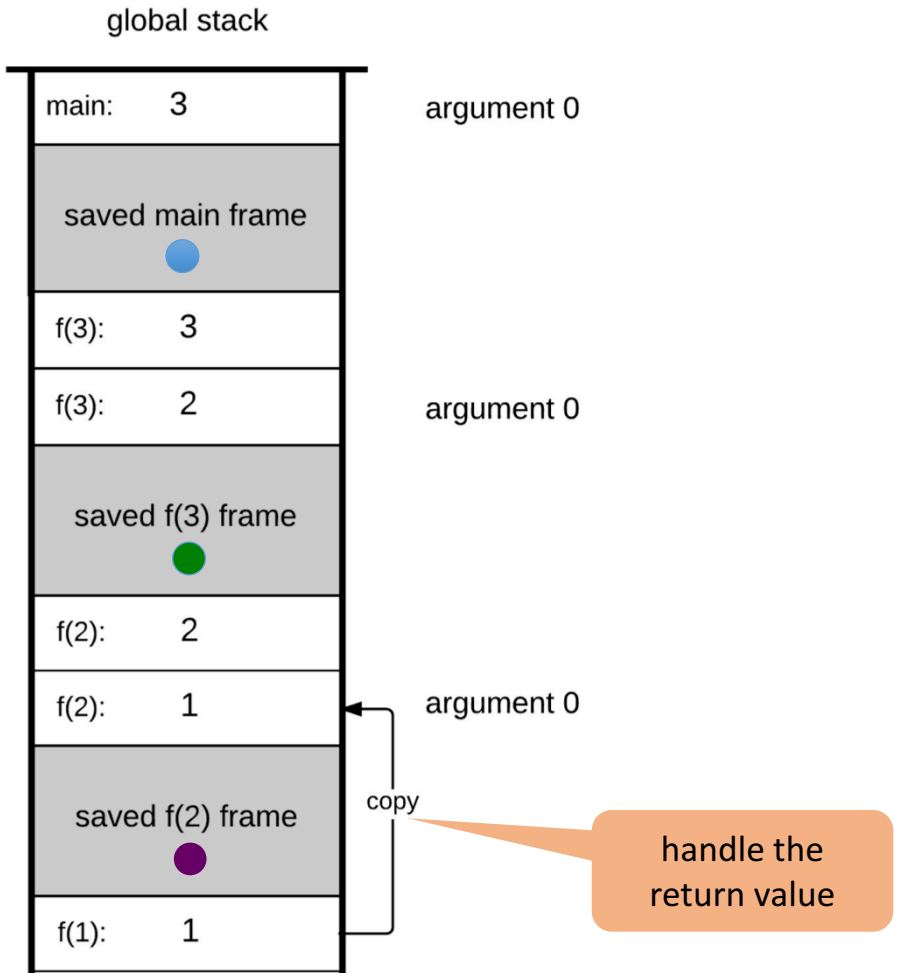
Run-time example

```
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
```



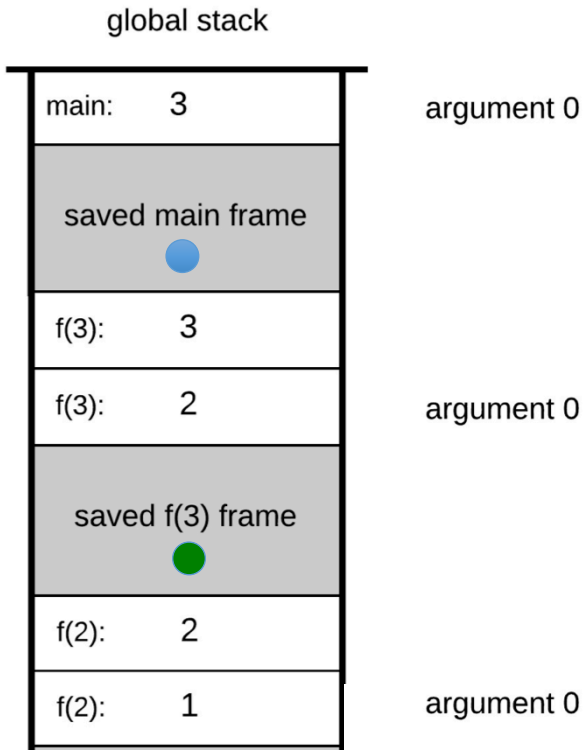
Run-time example

```
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
```



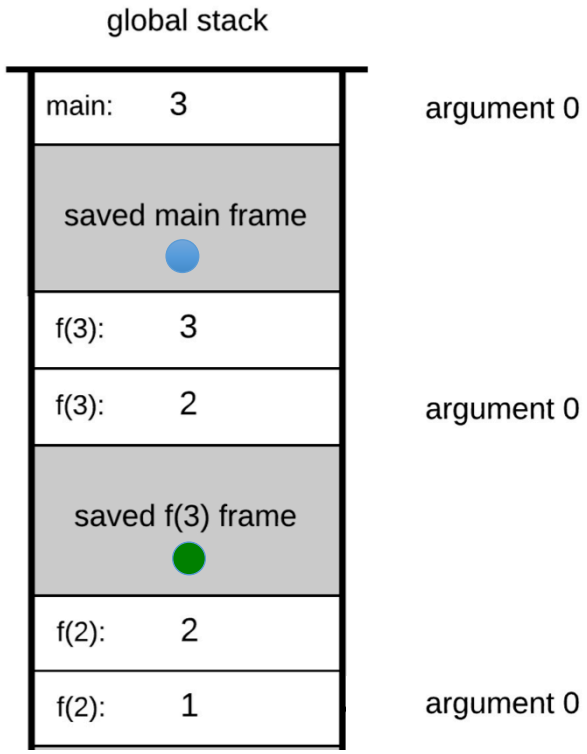
Run-time example

```
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
```



Run-time example

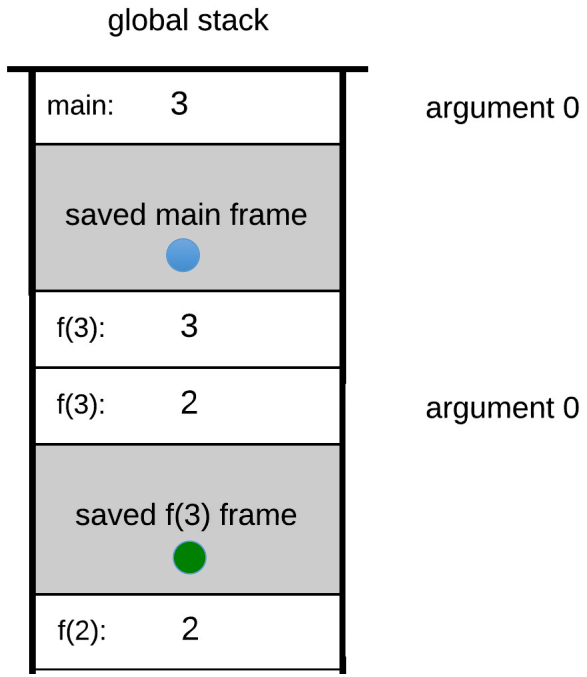
```
function main 0
   push constant 3
   call factorial 1
   return

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 for end result)

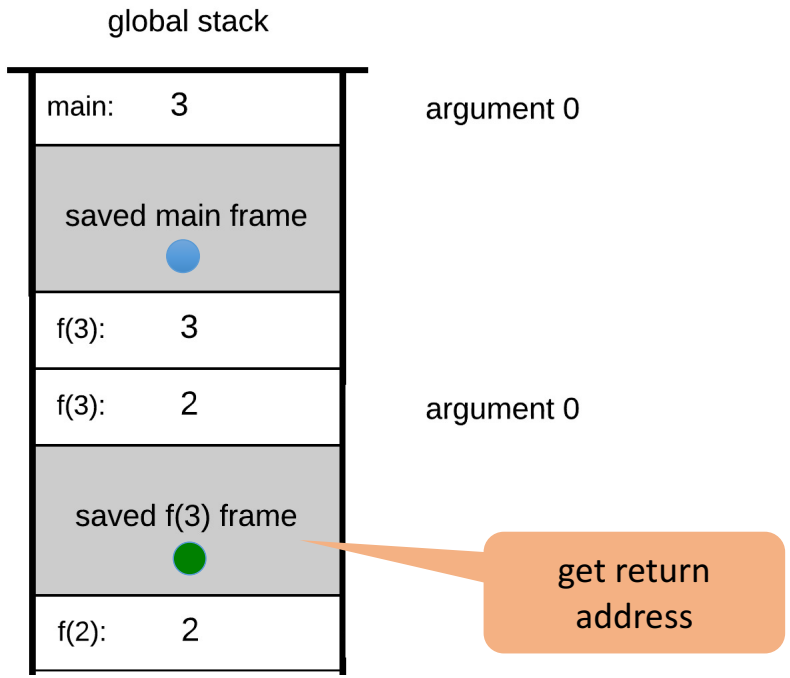
Run-time example

```
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
```



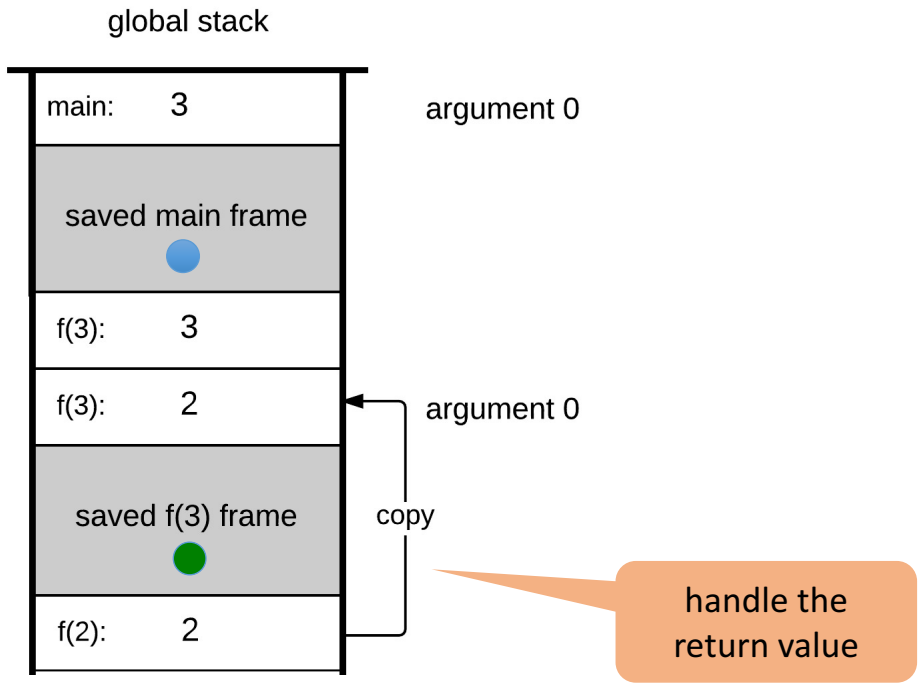
Run-time example

```
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
```



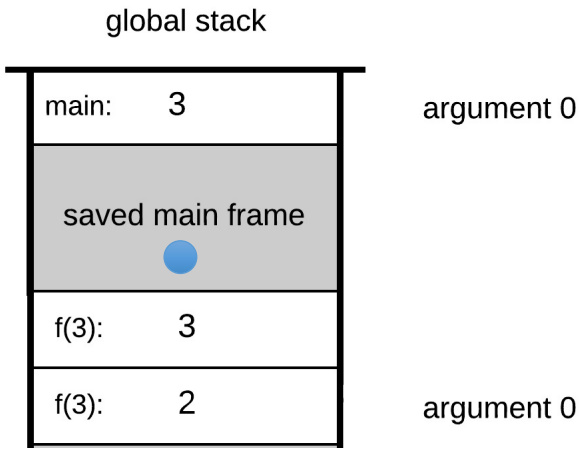
Run-time example

```
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
```



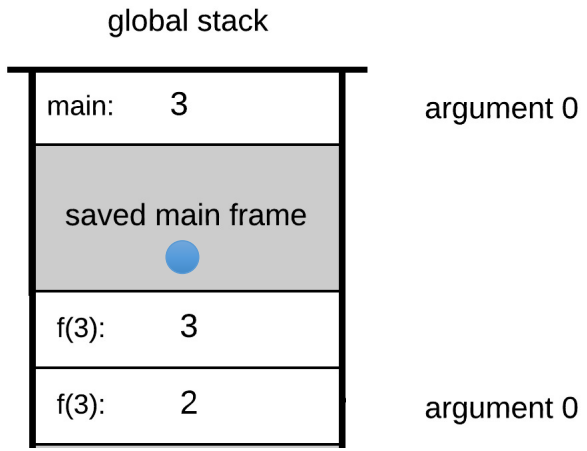
Run-time example

```
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
```



Run-time example

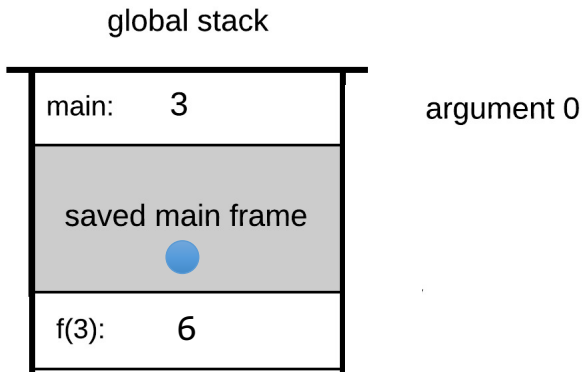
```
function main 0
   push constant 3
   call factorial 1
   return

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 for end result)

Run-time example

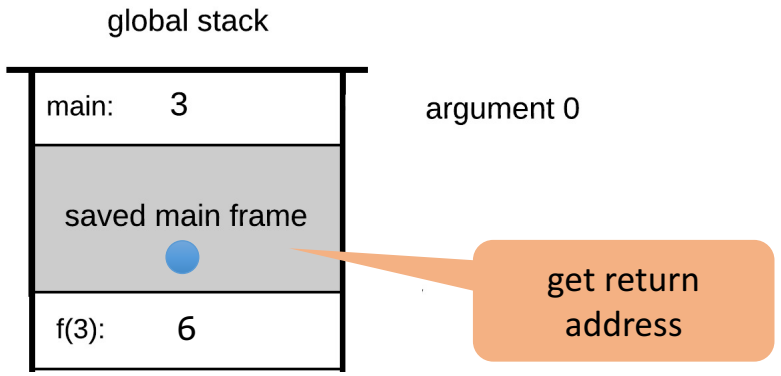
```
function main 0
   push constant 3
   call factorial 1
   return

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 for end result)

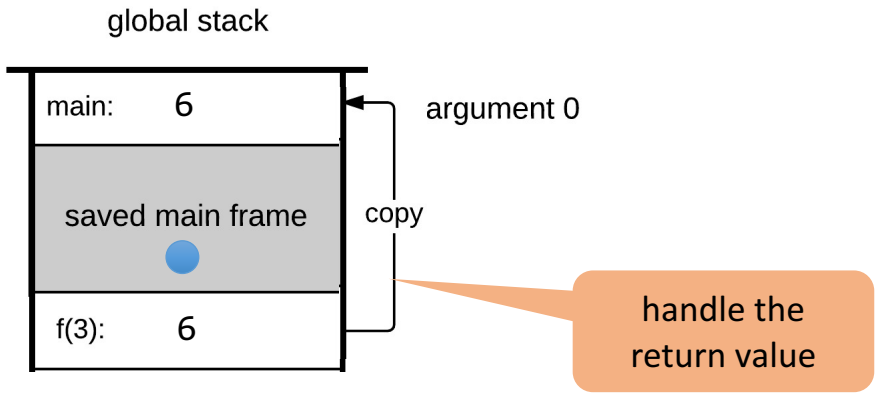
Run-time example

```
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
```



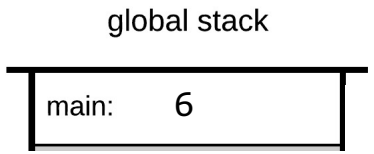
Run-time example

```
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
```



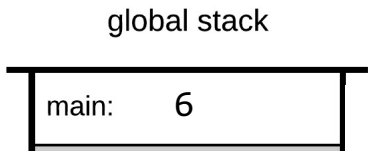
Run-time example

```
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
```



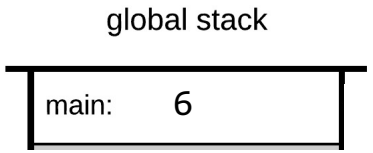
Run-time example

```
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
```



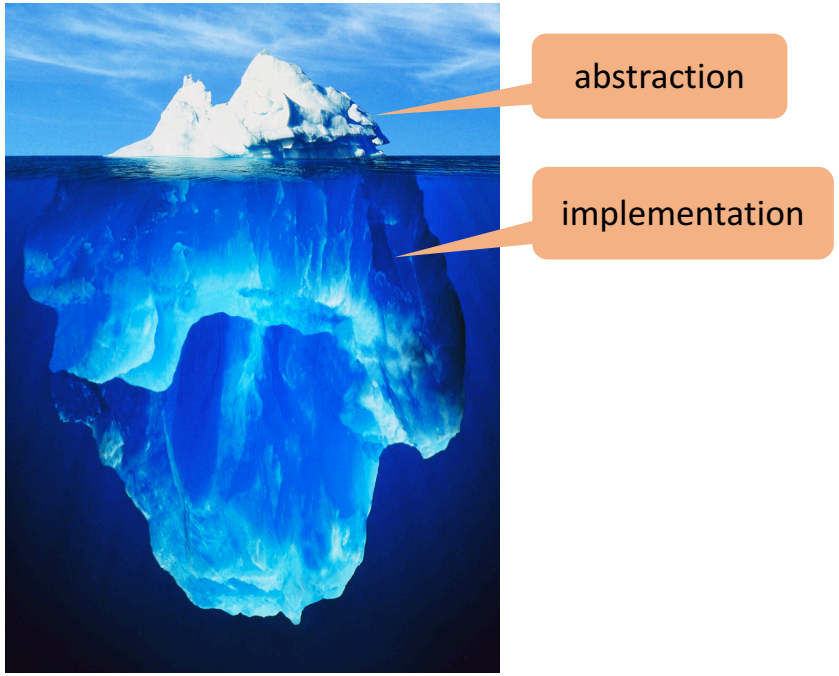
Recap

```
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 happenned...



Virtual machine: lecture plan

Overview

- Program control

Branching

- Abstraction
- Implementation

Functions

- Abstraction
- Implementation

Implementing function call-and-return:

- Implementation overview
- Run-time simulation
- Detailed implementation

VM implementation on the Hack platform:

- Standard mapping
- VM translator:
  proposed implementation
- Project 8 overview

Function call and return

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
```

callee

We focus on the VM function commands:

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

Contract: the caller's view

VM code

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, I must push as many arguments as the function expects to get
- Next, I invoke the function using `call` *functionName nArgs*
- After the called function returns, the argument values that I pushed before the call have disappeared from the stack, and a *return value* (that always exists) appears at the top of the stack;
- After the called function returns, 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

```
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
```

callee

- Before I 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
```

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)
```

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 fo
   goto Bar
(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)
```

psuedo code

We now turn to describe the more detailed handling of the VM commands:

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

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
```

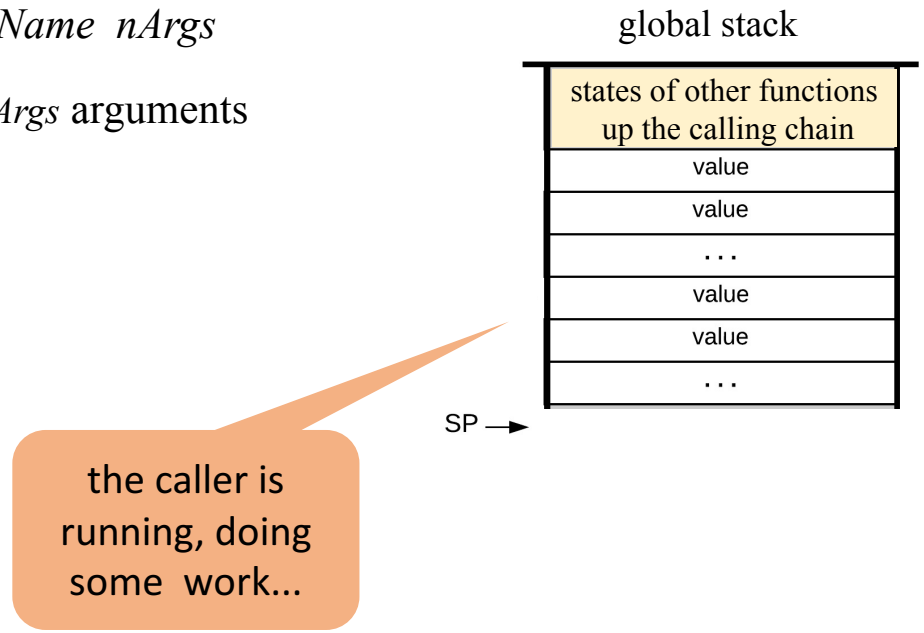
Generated assembly code

```
(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)
```

Handling `call`

VM command: `call` *functionName nArgs*

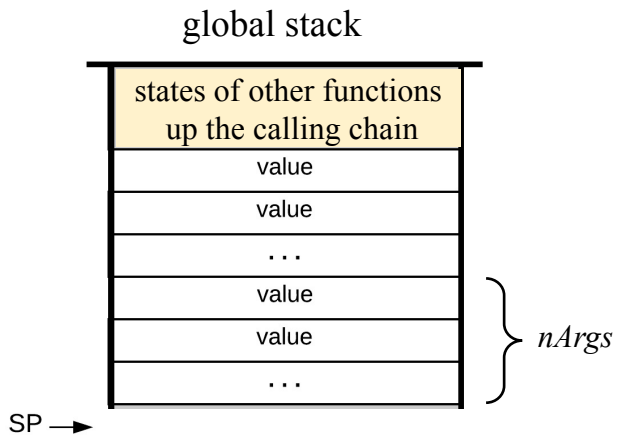
(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



Handling `call`

VM command: `call` *functionName nArgs*

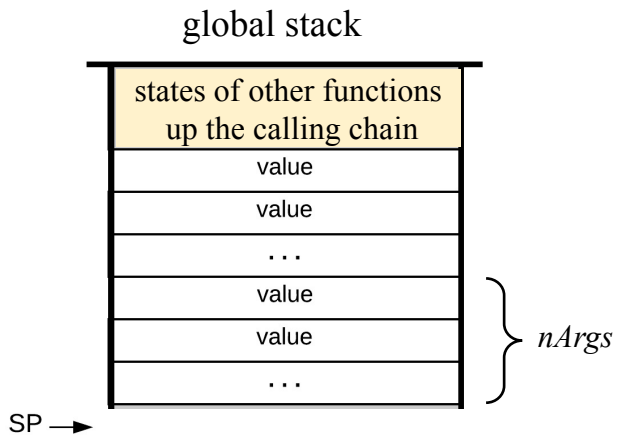
(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



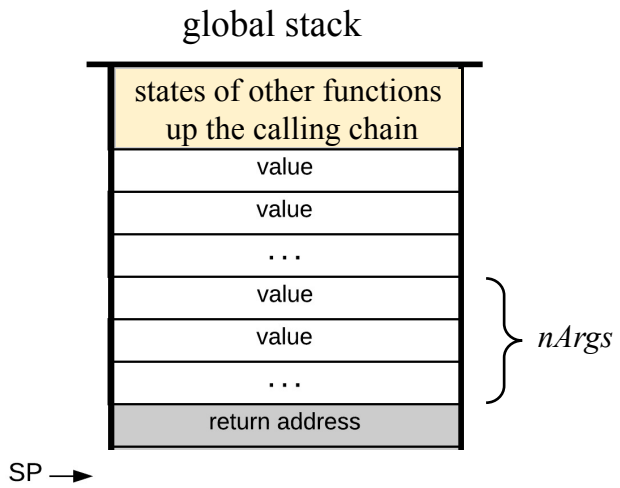
Assembly code (generated by the translator):

```
push retAddrLabel      // Using a translator-generated label
```

Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



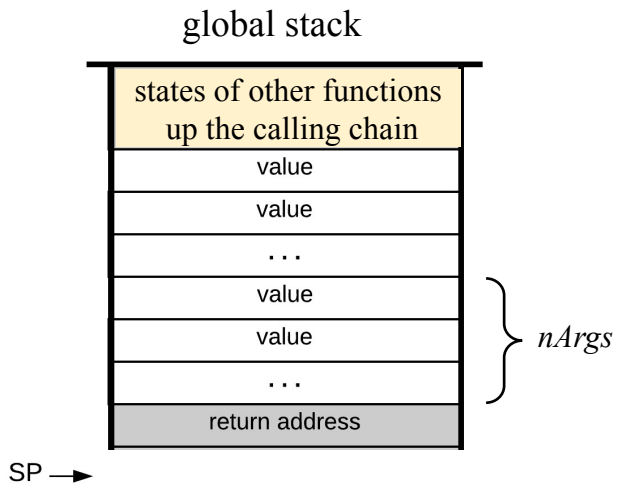
Assembly code (generated by the translator):

```
push retAddrLabel      // Using a translator-generated label
```

Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



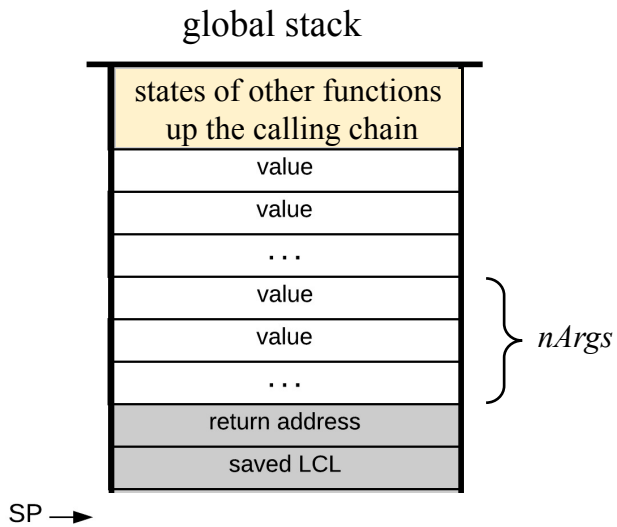
Assembly code (generated by the translator):

```
push retAddrLabel      // Using a translator-generated label
push LCL               // Saves LCL of the caller
```

Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



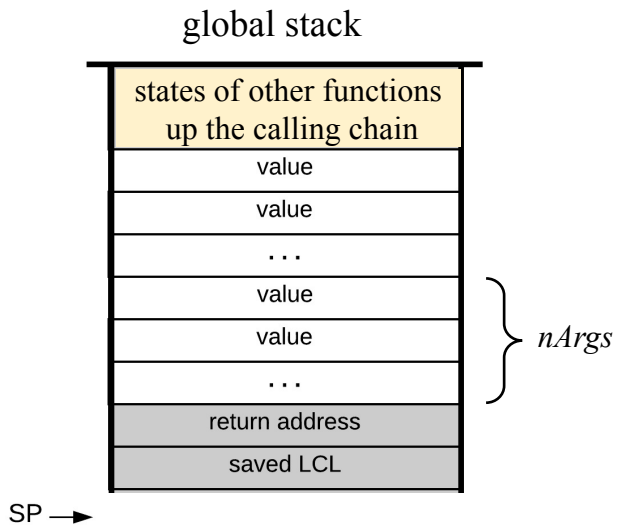
Assembly code (generated by the translator):

```
push retAddrLabel      // Using a translator-generated label
push LCL               // Saves LCL of the caller
```

Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



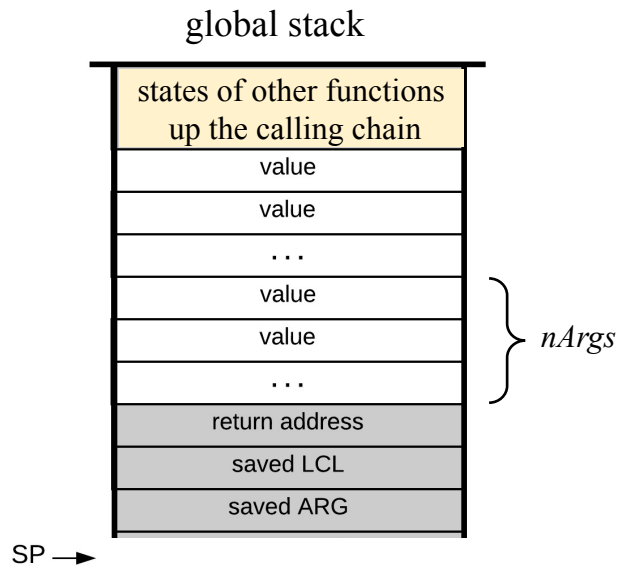
Assembly code (generated by the translator):

```
push retAddrLabel     // Using a translator-generated label
push LCL              // Saves LCL of the caller
push ARG              // Saves ARG of the caller
```

Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



Assembly code (generated by the translator):

```
push retAddrLabel     // Using a translator-generated label
push LCL              // Saves LCL of the caller
push ARG              // Saves ARG of the caller
```

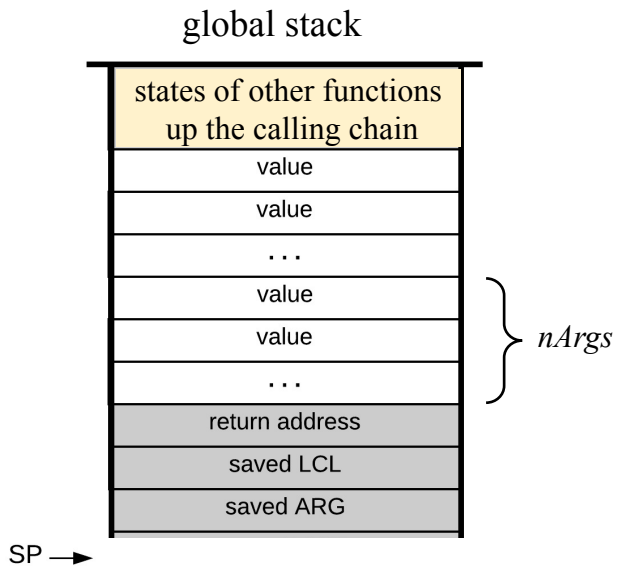
Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)

Assembly code (generated by the translator):

```
push retAddrLabel     // Using a translator-generated label
push LCL              // Saves LCL of the caller
push ARG              // Saves ARG of the caller
push THIS             // Saves THIS of the caller
```



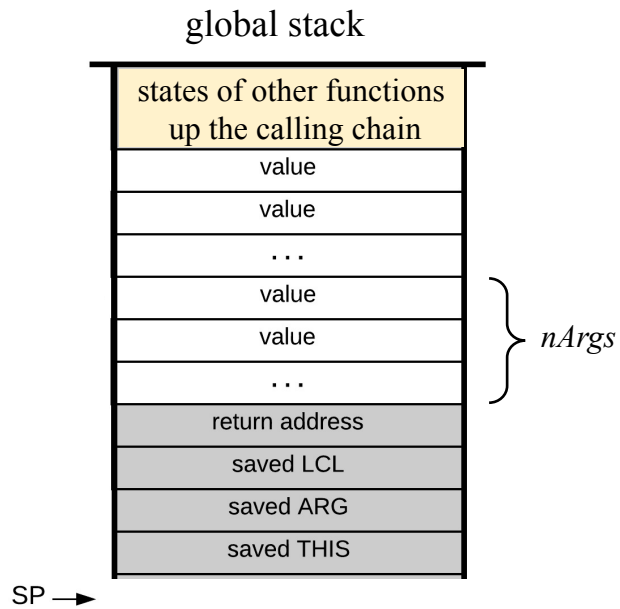
Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)

Assembly code (generated by the translator):

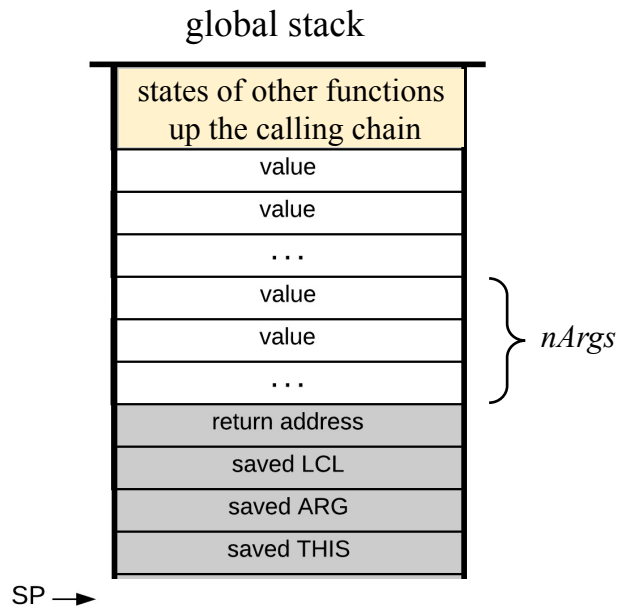
```
push retAddrLabel     // Using a translator-generated label
push LCL              // Saves LCL of the caller
push ARG              // Saves ARG of the caller
push THIS             // Saves THIS of the caller
```



Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



Assembly code (generated by the translator):

```
push retAddrLabel    // Using a translator-generated label
push LCL             // Saves LCL of the caller
push ARG             // Saves ARG of the caller
push THIS            // Saves THIS of the caller
push THAT            // Saves THAT of the caller
```

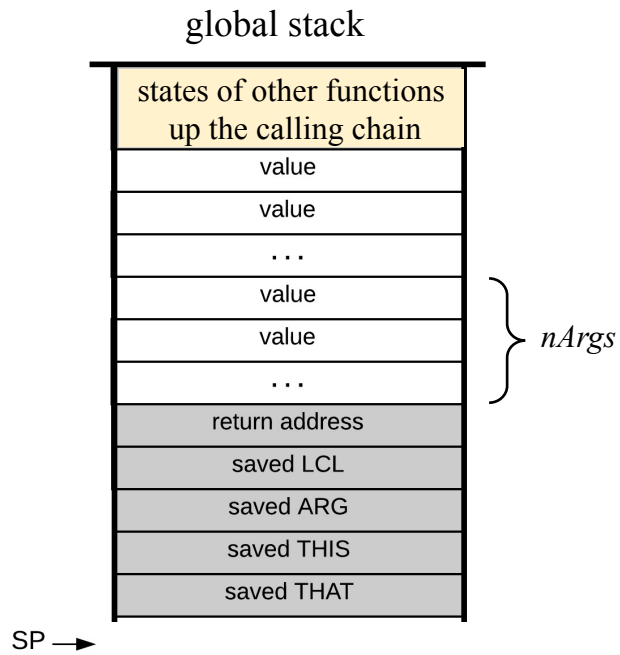
Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)

Assembly code (generated by the translator):

```
push retAddrLabel     // Using a translator-generated label
push LCL              // Saves LCL of the caller
push ARG              // Saves ARG of the caller
push THIS             // Saves THIS of the caller
push THAT             // Saves THAT of the caller
```



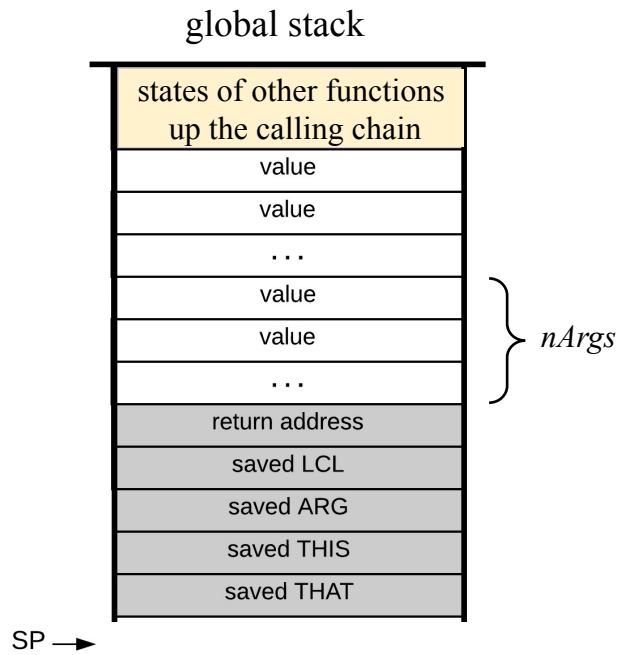
Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)

Assembly code (generated by the translator):

```
push retAddrLabel      // Using a translator-generated label
push LCL               // Saves LCL of the caller
push ARG               // Saves ARG of the caller
push THIS              // Saves THIS of the caller
push THAT              // Saves THAT of the caller
ARG = SP-5-nArgs       // Repositions ARG
```



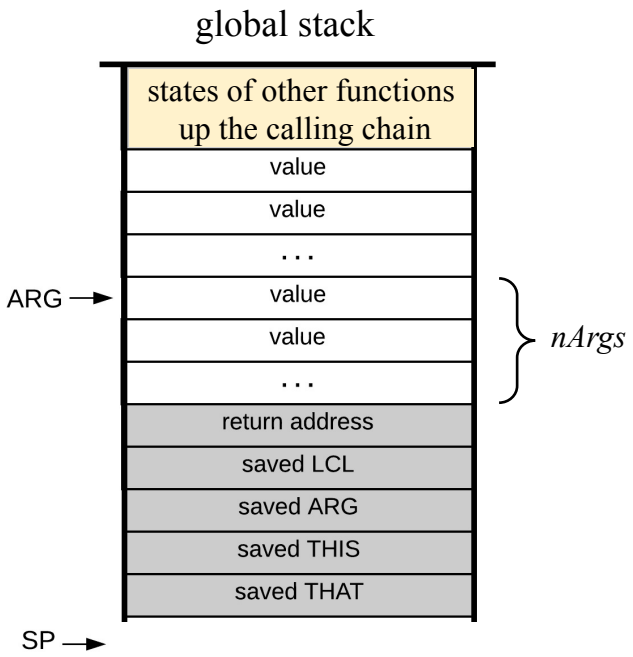
Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)

Assembly code (generated by the translator):

```
push retAddrLabel     // Using a translator-generated label
push LCL              // Saves LCL of the caller
push ARG              // Saves ARG of the caller
push THIS             // Saves THIS of the caller
push THAT             // Saves THAT of the caller
ARG = SP-5-nArgs      // Repositions ARG
```



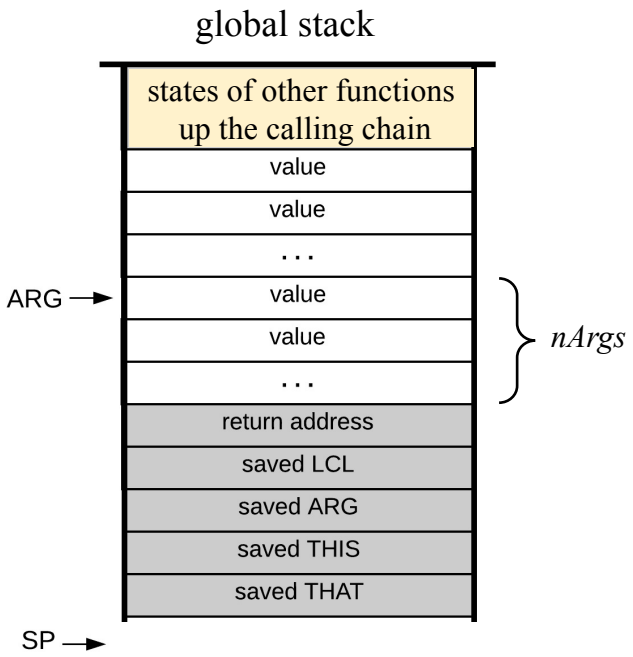
Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)

Assembly code (generated by the translator):

```
push retAddrLabel     // Using a translator-generated label
push LCL              // Saves LCL of the caller
push ARG              // Saves ARG of the caller
push THIS             // Saves THIS of the caller
push THAT             // Saves THAT of the caller
ARG = SP-5-nArgs      // Repositions ARG
LCL = SP              // Repositions LCL
```



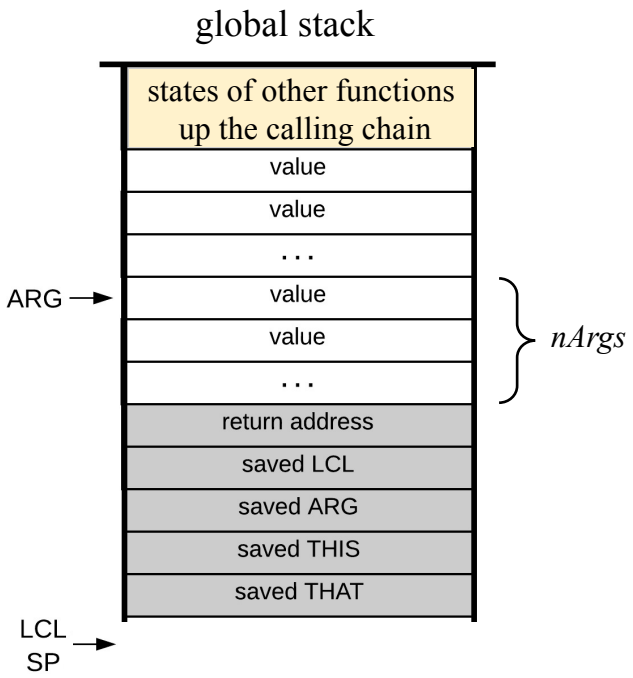
Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)

Assembly code (generated by the translator):

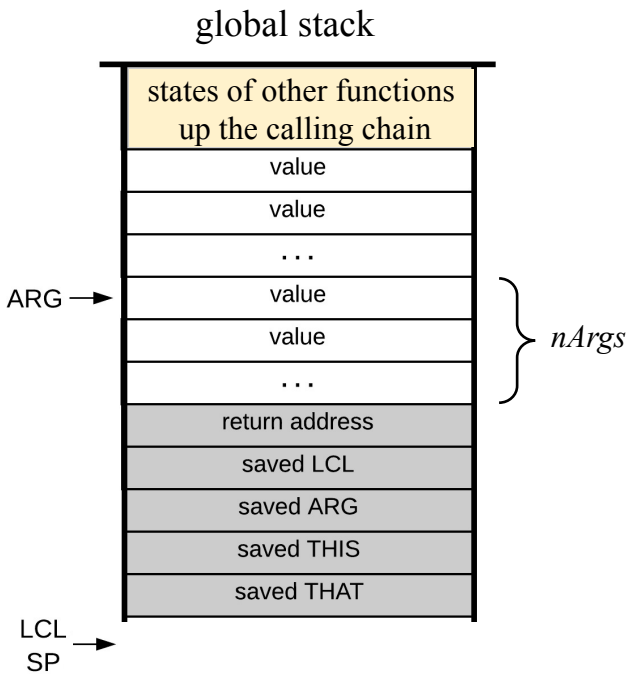
```
push retAddrLabel     // Using a translator-generated label
push LCL              // Saves LCL of the caller
push ARG              // Saves ARG of the caller
push THIS             // Saves THIS of the caller
push THAT             // Saves THAT of the caller
ARG = SP-5-nArgs      // Repositions ARG
LCL = SP              // Repositions LCL
```



Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



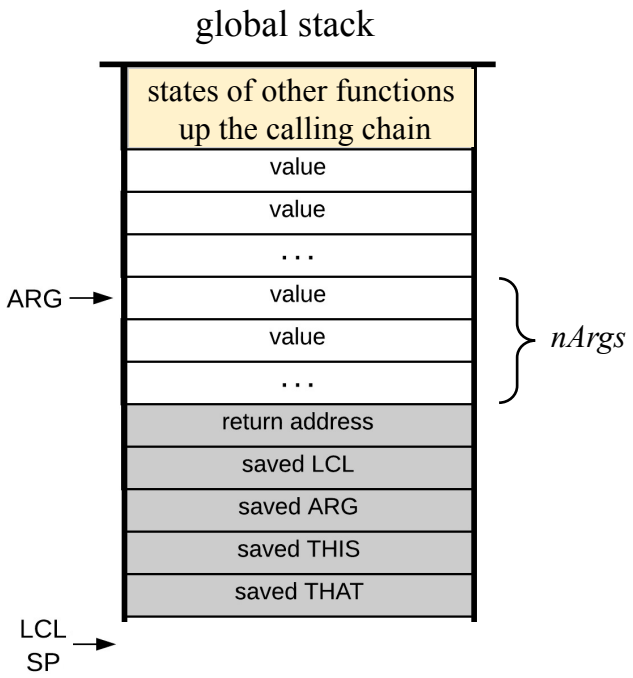
Assembly code (generated by the translator):

```
push retAddrLabel      // Using a translator-generated label
push LCL               // Saves LCL of the caller
push ARG               // Saves ARG of the caller
push THIS              // Saves THIS of the caller
push THAT              // Saves THAT of the caller
ARG = SP-5-nArgs       // Repositions ARG
LCL = SP               // Repositions LCL
goto functionName      // Transfers control to the called function
```

Handling `call`

VM command: `call` *functionName nArgs*

(calls a function, informing that *nArgs* arguments have been pushed onto the stack)



Assembly code (generated by the translator):

```
    push retAddrLabel      // Using a translator-generated label
    push LCL               // Saves LCL of the caller
    push ARG               // Saves ARG of the caller
    push THIS              // Saves THIS of the caller
    push THAT              // Saves THAT of the caller
    ARG = SP-5-nArgs       // Repositions ARG
    LCL = SP               // Repositions LCL
    goto functionName      // Transfers control to the called function
(retAddrLabel)             //  the same translator-generated label
```

Handling `call`

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
```

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)
```

Handling `function`

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

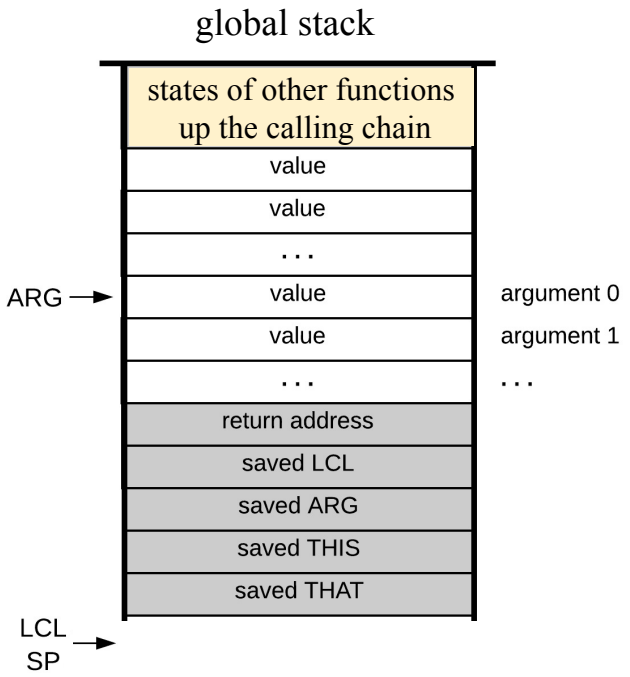
```
(Foo.main)          // created and plugged by the translator
  // assembly code that handles the initialization of the
  // function's execution
  ...
  // assembly code that handles push constant 19
  // assembly code that handles push local 3
  // assembly code that saves the caller's state on 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)
```

Handling `function`

VM command: `function` *functionName nVars*

(here starts a function that has *nVars* local variables)

Assembly code (generated by the translator):



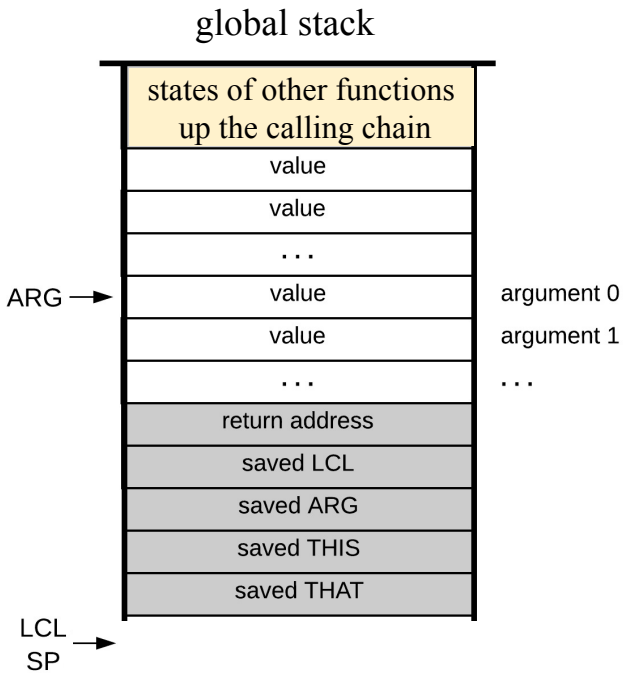
Handling `function`

VM command: `function` *functionName nVars*

(here starts a function that has *nVars* local variables)

Assembly code (generated by the translator):

```
(functionName)          // using a translator-generated label
    repeat nVars times: // nVars = number of local variables
    push 0              // initializes the local variables to 0
```



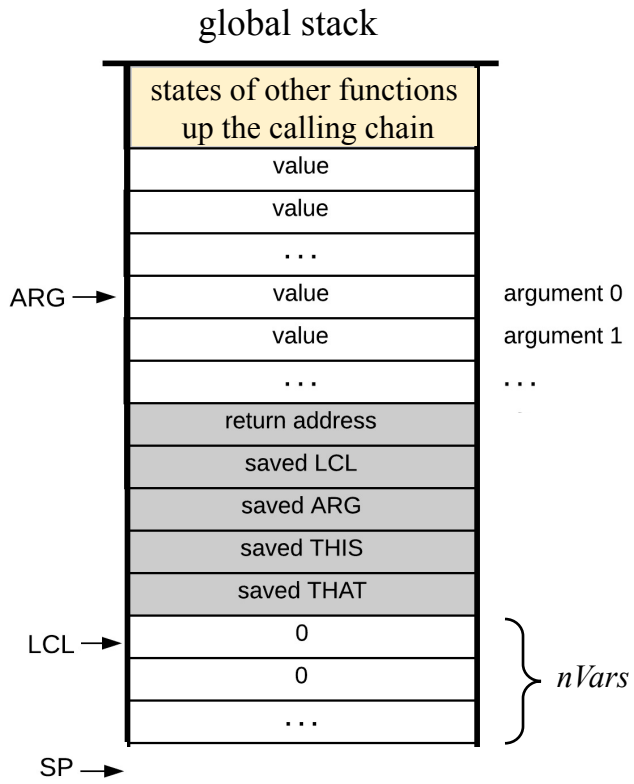
Handling `function`

VM command: `function` *functionName nVars*

(here starts a function that has *nVars* local variables)

Assembly code (generated by the translator):

```
(functionName)          // using a translator-generated label
  repeat nVars times:   // nVars = number of local variables
  push 0                // initializes the local variables to 0
```



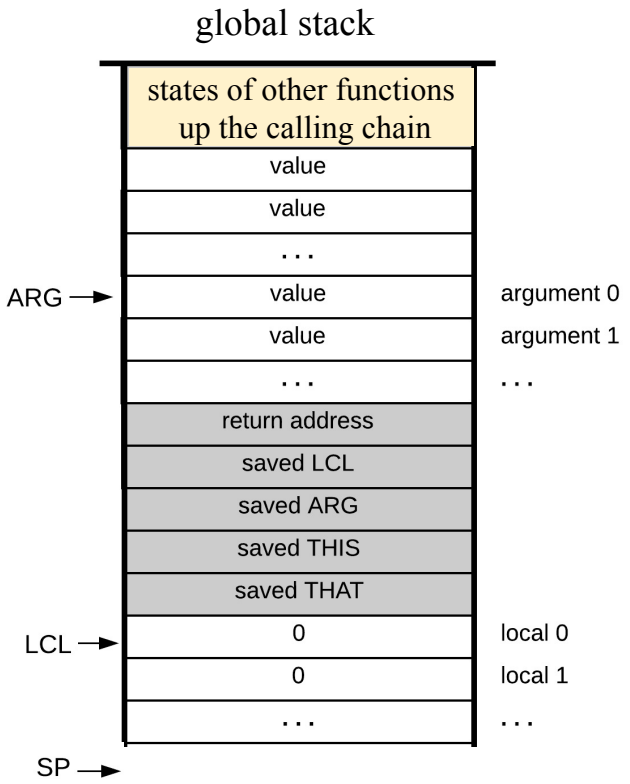
Handling `function`

VM command: `function` *functionName nVars*

(here starts a function that has *nVars* local variables)

Assembly code (generated by the translator):

```
(functionName)          // using a translator-generated label
  repeat nVars times:   // nVars = number of local variables
  push 0                // initializes the local variables to 0
```



Handling `function`

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)
```

Handling `function`

VM code

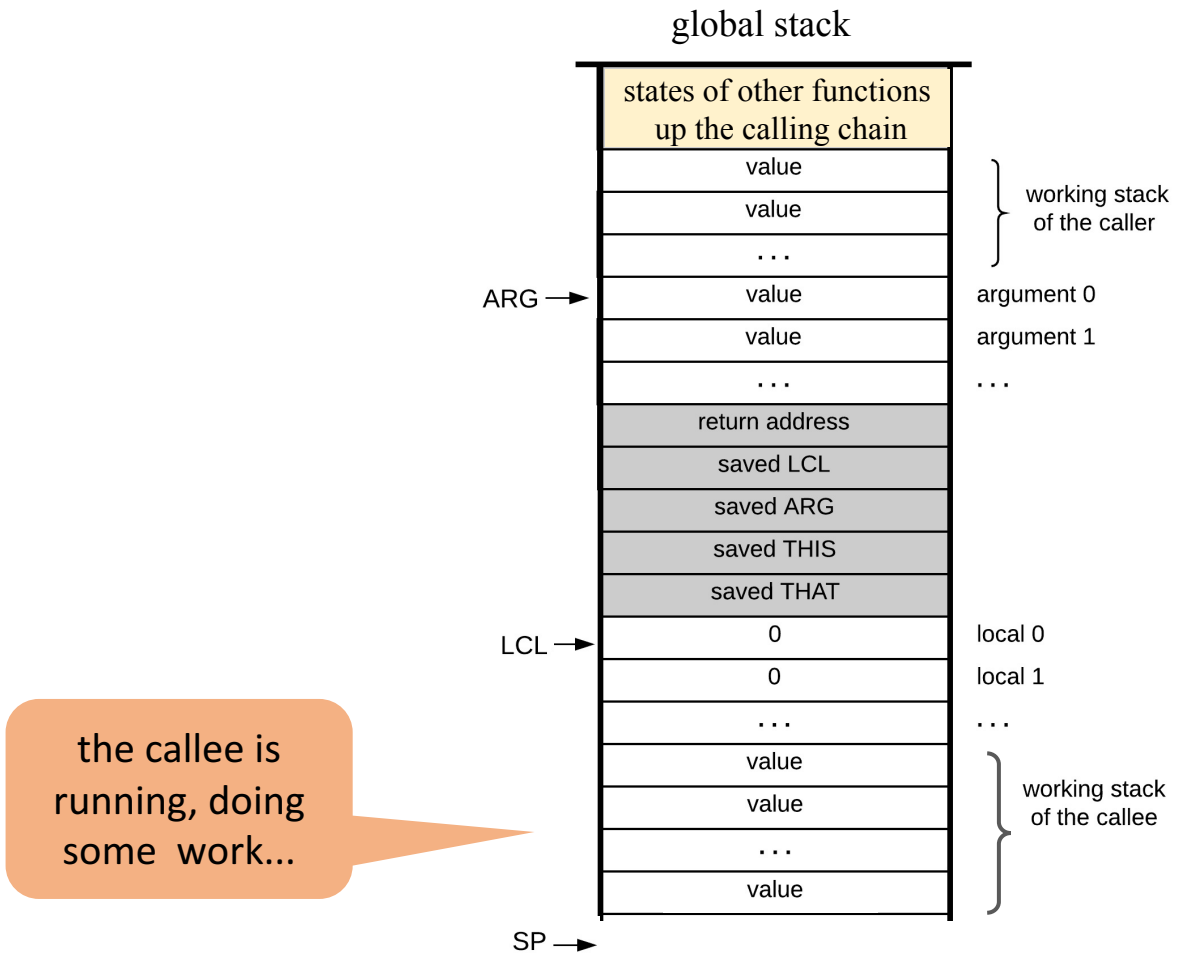
```
function Foo.main 4
  ...
  // computes -(19 * (local 3))
  push constant 19
  push local 3
  call Bar.mult 2
  neg
  ...

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
```

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)
```

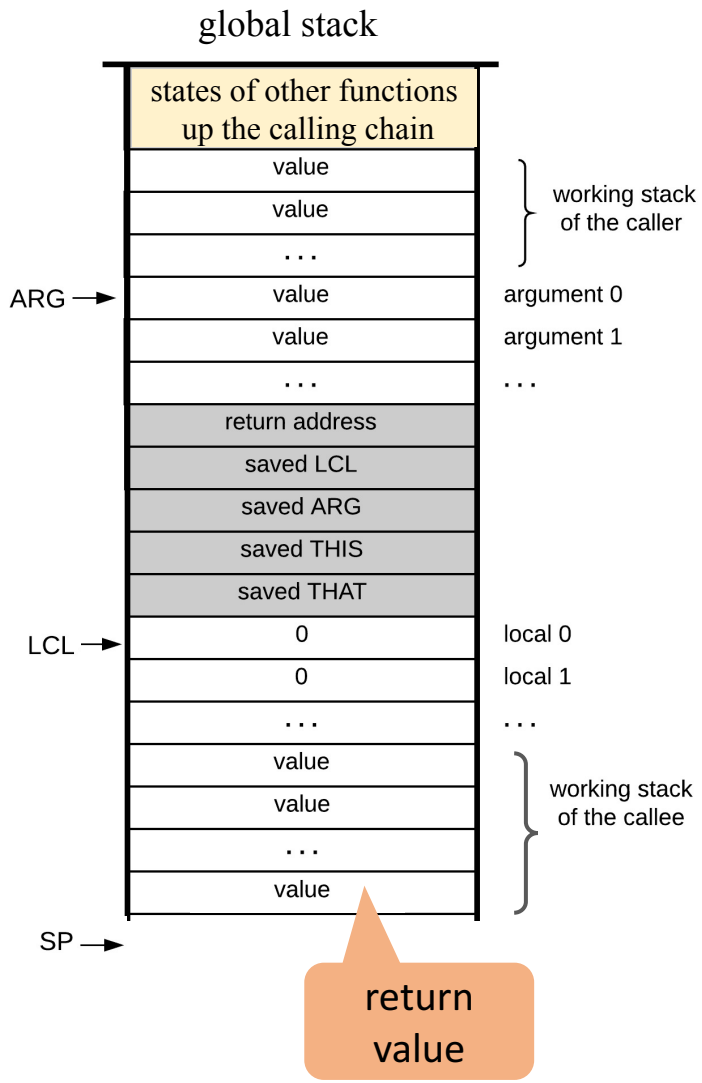
Handling `return`

VM command: `return`



Handling `return`

VM command: `return`

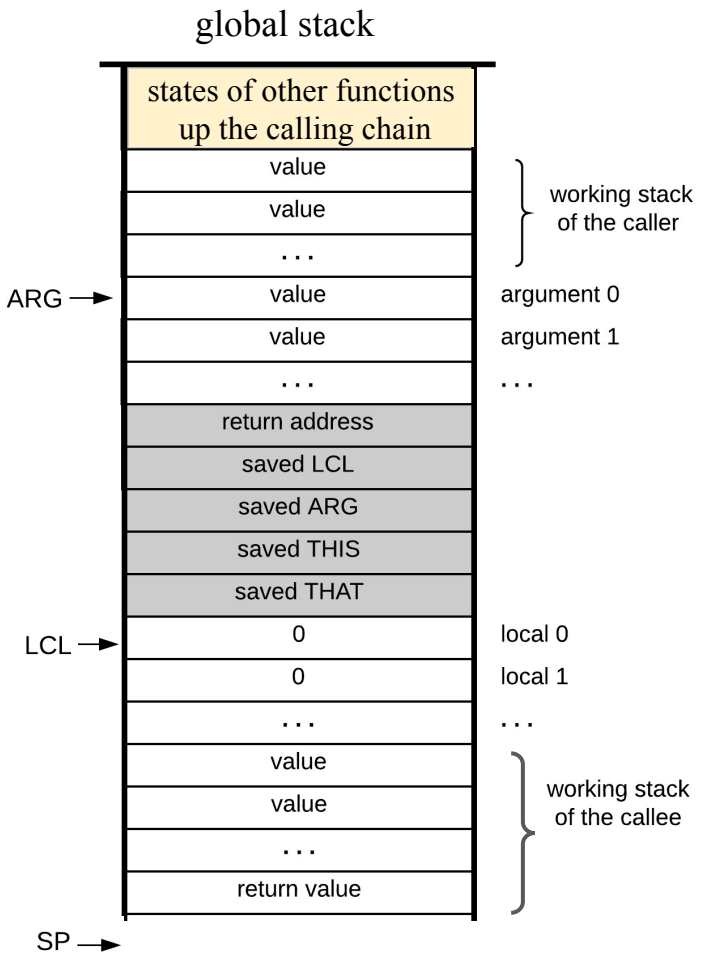


Handling `return`

VM command: `return`

Assembly code (generated by the translator):

```
endFrame = LCL                // endframe is a temporary variable
retAddr = *(endFrame – 5)     // gets the return address
```

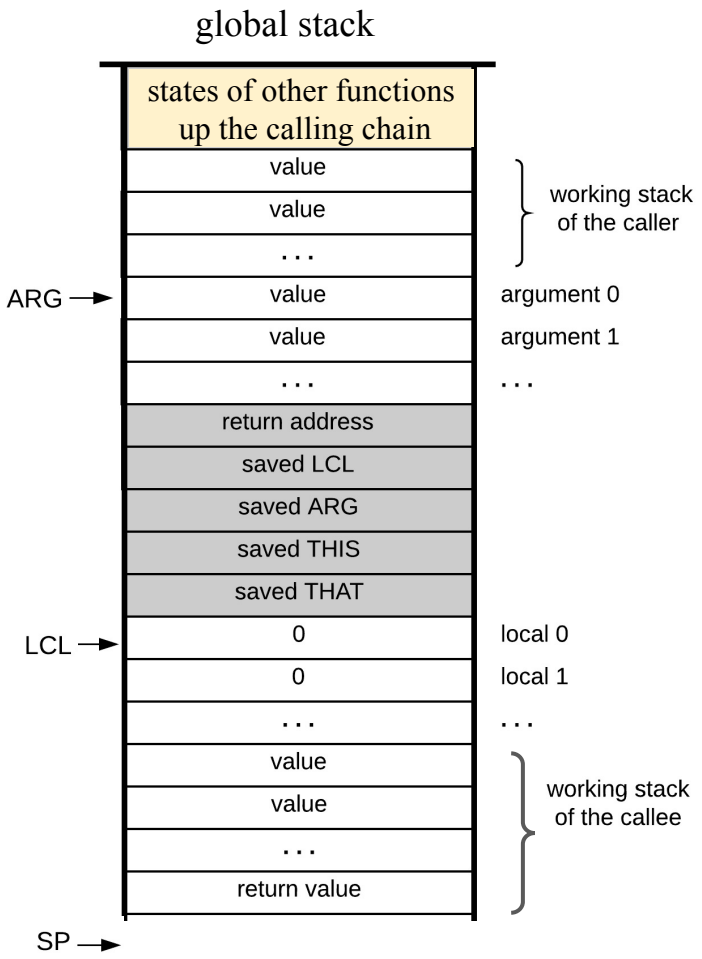


Handling `return`

VM command: `return`

Assembly code (generated by the translator):

```
endFrame = LCL          // endframe is a temporary variable
retAddr = *(endFrame - 5)  // gets the return address
*ARG = pop()            // repositions the return value for the caller
```

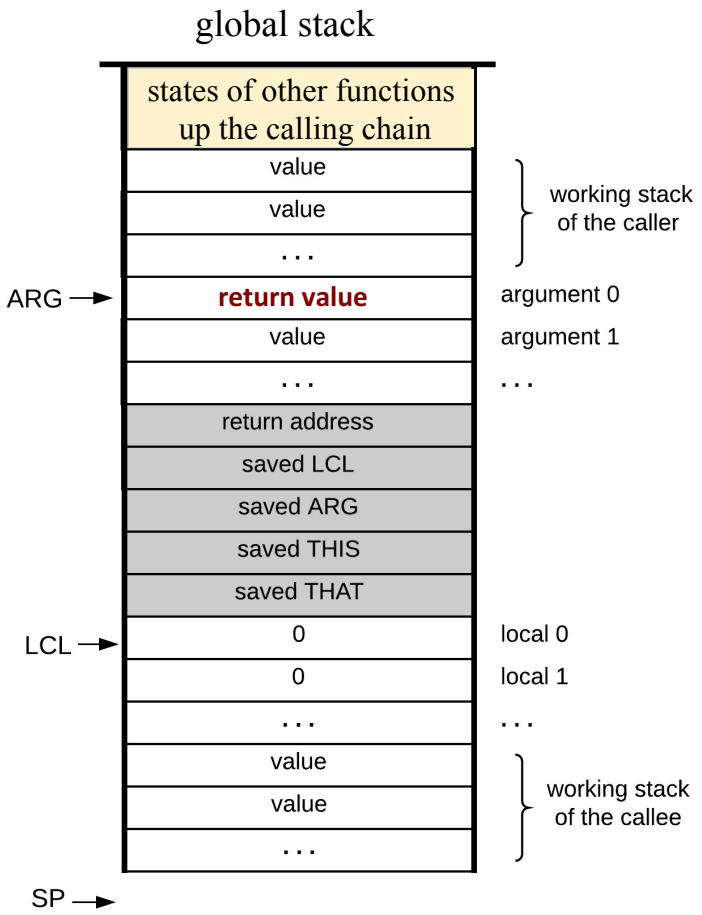


Handling `return`

VM command: `return`

Assembly code (generated by the translator):

```
endFrame = LCL          // endframe is a temporary variable
retAddr = *(endFrame - 5)  // gets the return address
*ARG = pop()            // repositions the return value for the caller
```

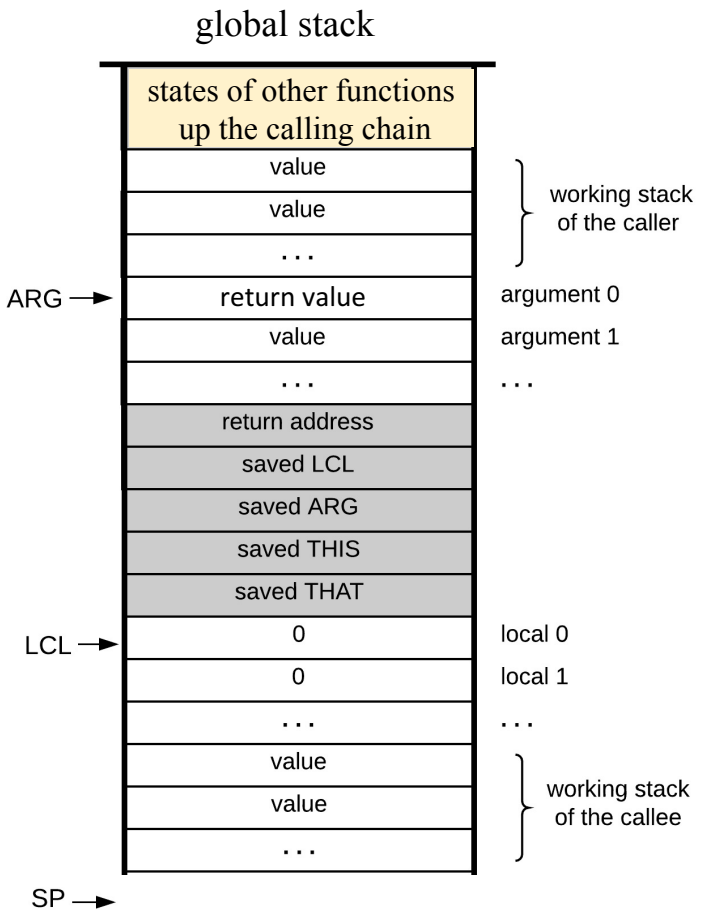


Handling `return`

VM command: `return`

Assembly code (generated by the translator):

```
endFrame = LCL          // endframe is a temporary variable
retAddr = *(endFrame - 5)  // gets the return address
*ARG = pop()            // repositions the return value for the caller
SP = ARG + 1            // repositions SP of the caller
```

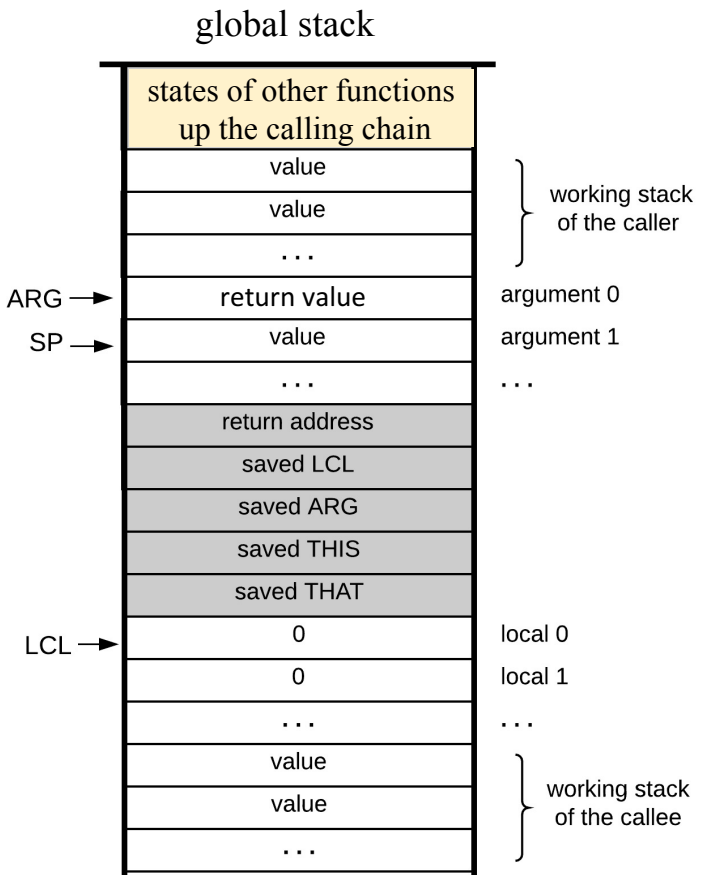


Handling `return`

VM command: `return`

Assembly code (generated by the translator):

```
endFrame = LCL          // endframe is a temporary variable
retAddr = *(endFrame - 5)  // gets the return address
*ARG = pop()            // repositions the return value for the caller
SP = ARG + 1            // repositions SP of the caller
```

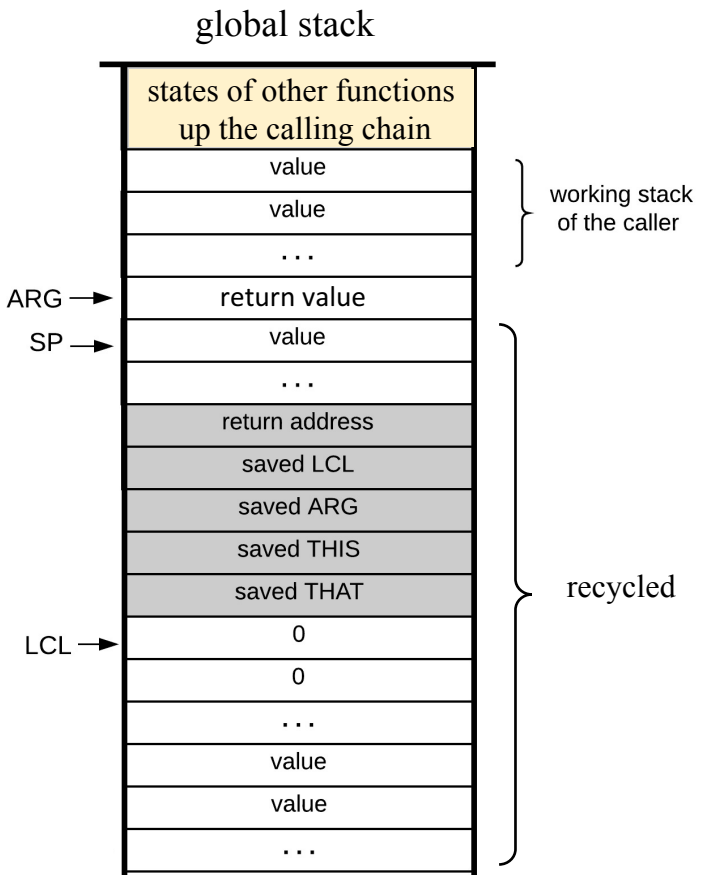


Handling `return`

VM command: `return`

Assembly code (generated by the translator):

```
endFrame = LCL                // endframe is a temporary variable
retAddr = *(endFrame – 5)     // gets the return address
*ARG = pop()                  // repositions the return value for the caller
SP = ARG + 1                  // repositions SP of the caller
```

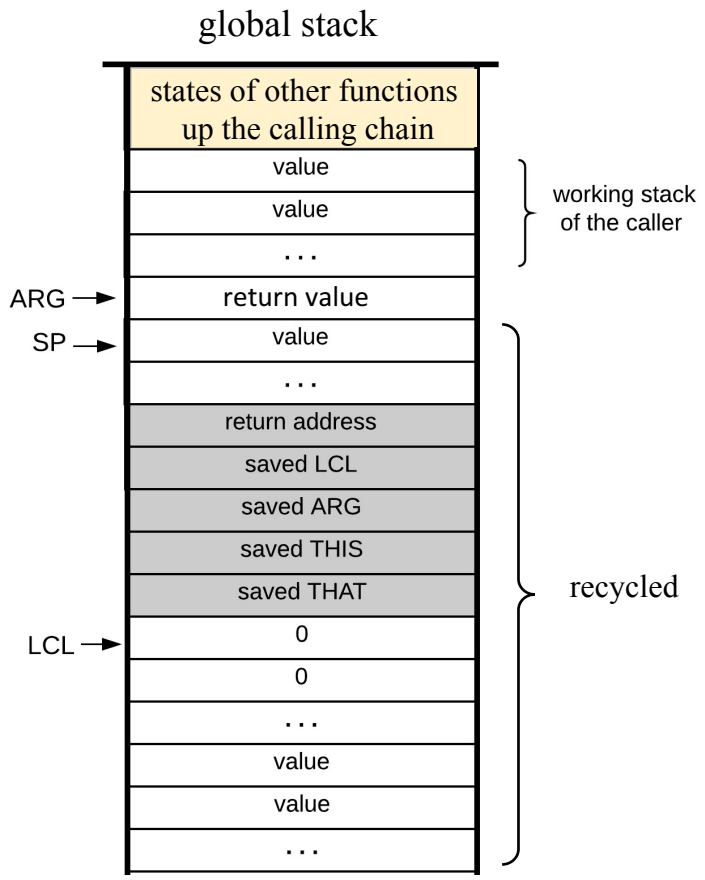


Handling `return`

VM command: `return`

Assembly code (generated by the translator):

```
endFrame = LCL          // endframe is a temporary variable
retAddr = *(endFrame – 5)  // gets the return address
*ARG = pop()            // repositions the return value for the caller
SP = ARG + 1            // repositions SP of the caller
THAT = *(endFrame – 1)  // restores THAT of the caller
THIS = *(endFrame – 2)  // restores THIS of the caller
ARG = *(endFrame – 3)   // restores ARG of the caller
LCL = *(endFrame – 4)   // restores LCL of the caller
goto retAddr            // goes to the caller's return address
```

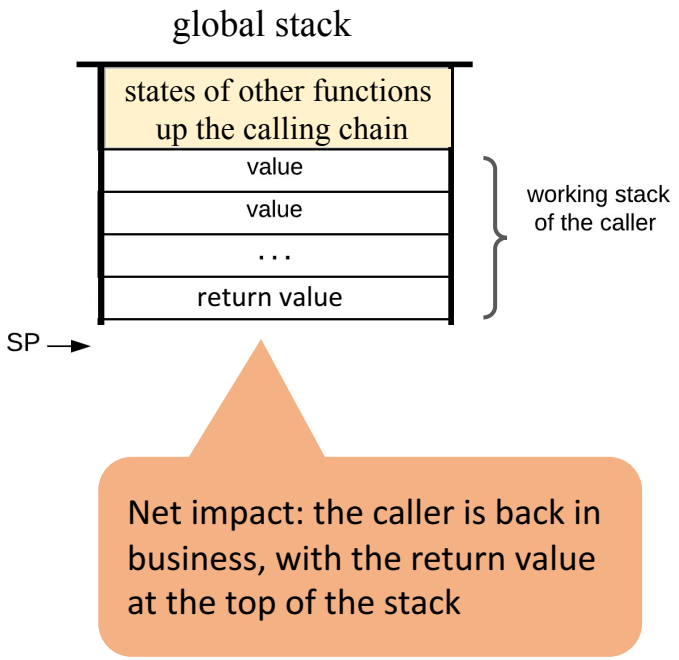


Handling `return`

VM command: `return`

Assembly code (generated by the translator):

```
endFrame = LCL          // endframe is a temporary variable
retAddr = *(endFrame - 5)  // gets the return address
*ARG = pop()            // repositions the return value for the caller
SP = ARG + 1            // repositions SP of the caller
THAT = *(endFrame - 1)  // restores THAT of the caller
THIS = *(endFrame - 2)  // restores THIS of the caller
ARG = *(endFrame - 3)   // restores ARG of the caller
LCL = *(endFrame - 4)   // restores LCL of the caller
goto retAddr            // goes to the caller's return address
```



Function call and return

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
```

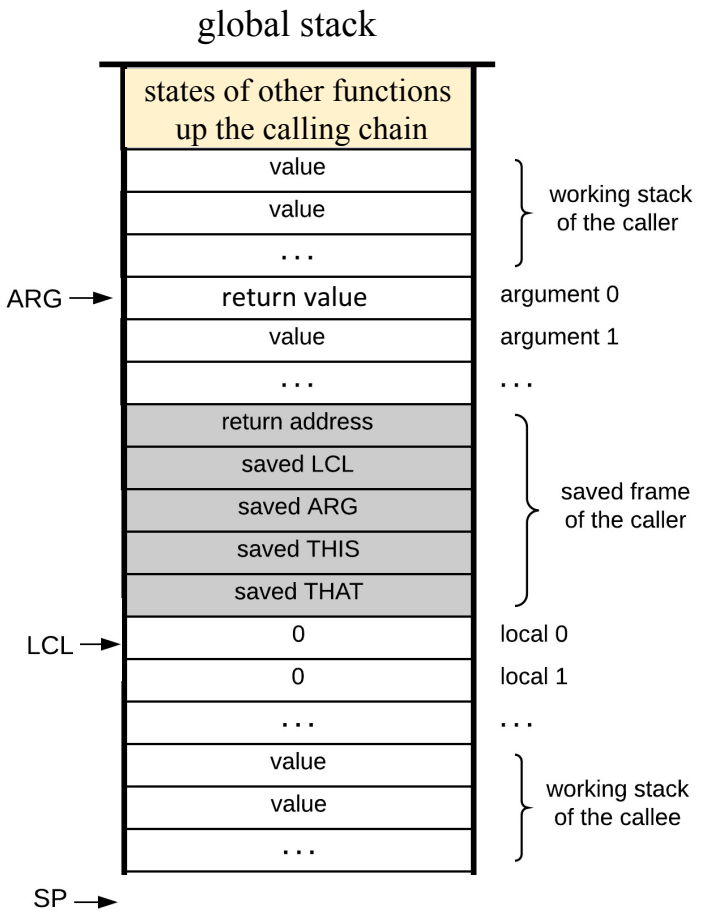
VM translator

Generated assembly code

```
(Foo.main)
   // assembly code that handles the setting up of
   // a 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,
    // handles some pointers, and then:
    goto Bar.mult     // (in assembly)
(Foo$ret.1)
    // assembly code that handles neg
    ...
(Bar.mult)
    // assembly code that handles the setting up of
    // a function's execution
    ...
    // assembly code that handles push local 1
    // Assembly code that moves the return value to the
    // caller, reinstates the caller's state, and then:
    goto Foo$ret.1     // (in assembly)
```

Recap

- We showed how to generate the assembly code that, when executed, will end up building and maintaining the global stack during run-time
- This code will implement the function call-and-return commands and behavior
- The code is language- and platform-independent
- It can be implemented in any machine language.



VM language

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`

Virtual machine: lecture plan

Overview

- Program control

Branching

- Abstraction
- Implementation

Functions

- Abstraction
- Implementation

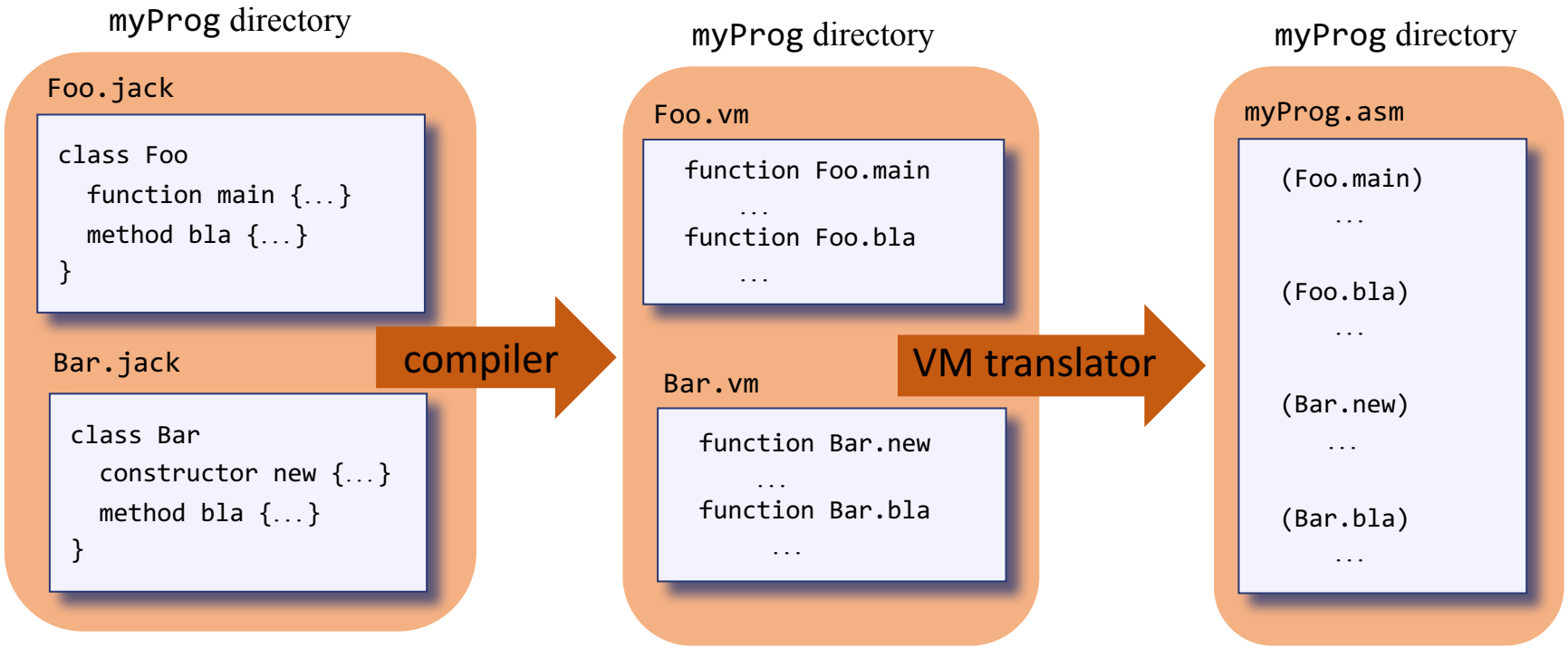
Implementing function call-and-return:

- Implementation overview
- Run-time simulation
- Detailed implementation

VM implementation on the Hack platform:

- Standard mapping
- VM translator:
  proposed implementation
- Project 8 overview

The big picture: program compilation and translation



- Compiling a program directory: `> JackCompiler` *directoryName* (later in the course)
- Translating a program directory: `> VMTranslator` *directoryName*

The VM translator developed in projects 7-8

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

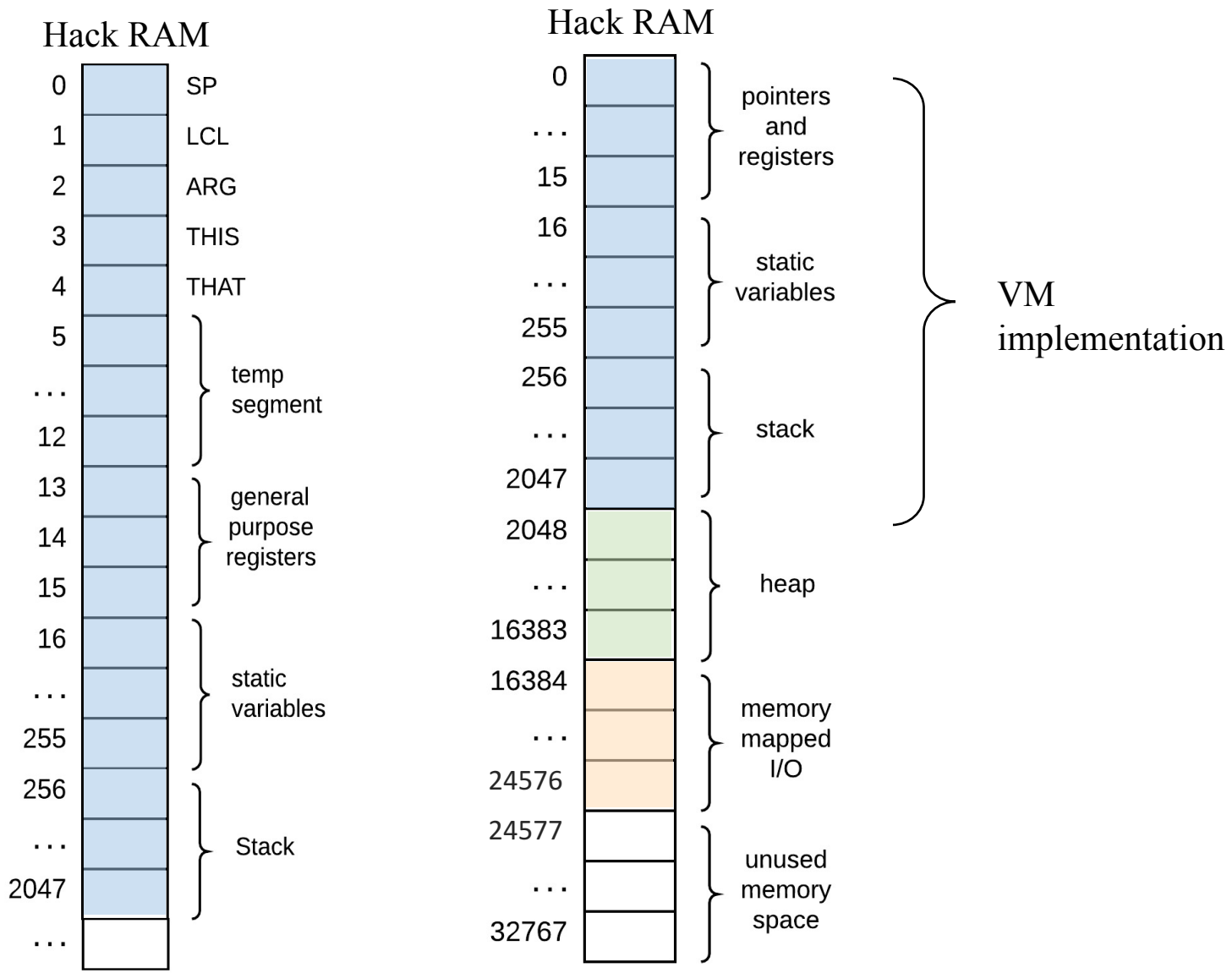
- the stack starts in address 256 in the host RAM
- when the VM implementation starts running, or is reset,
  it starts executing an argument-less OS function named `Sys.init`
- `Sys.init` is designed to call `Main.main`, and then enter an infinite loop

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

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	Each static variable *i* in file `Xxx.vm` is translated into the assembly symbol `Xxx.`*j*., where *j* 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 `$` *label*	Let `foo` be a function within a VM file `Xxx`. Each `label bar` command within `foo` should generate and insert into the assembly code stream a symbol `Xxx.foo$bar`. When translating `goto bar` and `if-goto bar` commands (within `foo`) into assembly, the full label specification `Xxx.foo$bar` must be used instead of `bar`.
functionName	Each `function foo` command within a VM file `Xxx` should generate and insert into the assembly code stream a symbol `Xxx.foo` that labels the entry point to the function's code. In the subsequent assembly process, the assembler will translate this symbol into the physical memory address where the function code starts.
functionName `$ret.`*i*	Let `foo` be a function within a VM file `Xxx`. Within `foo`, each function `call` command should generate and insert into the assembly code stream a symbol `Xxx.foo$ret.`*i*, where *i* is a running integer (one such symbol should be generated for each `call` command within `foo`). 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.

Virtual machine: lecture plan

Overview

- Program control

Branching

- Abstraction
- Implementation

Functions

- Abstraction
- Implementation

Implementing function call-and-return:

- Implementation overview
- Run-time simulation
- Detailed implementation

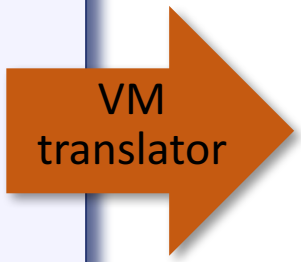
VM implementation on the Hack platform:

- Standard mapping
- VM translator:
  proposed implementation
- Project 8 overview

The VM translator

VM code (example)

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



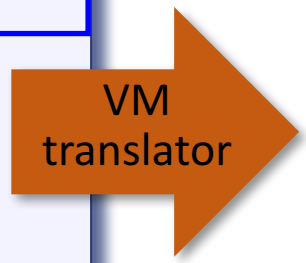
Generated assembly code

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

The VM translator

VM code

```
...
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
```

The VM translator

VM code

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

VM translator

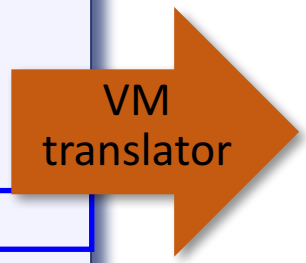
Generated assembly code

```
...
// 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 call Foo.bar 3
```

The VM translator

VM code

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



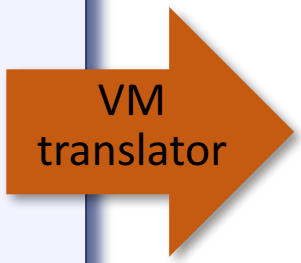
Generated assembly code

```
...
// 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 call Foo.bar 3

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

The VM translator

VM code

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



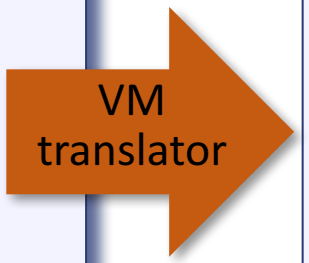
Generated assembly code

```
...
// 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 call Foo.bar 3

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

// push local 4
... // generated ... that implements push local 4
```

The VM translator

VM code

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



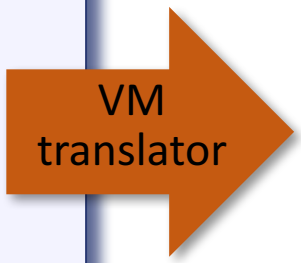
Generated assembly code

```
...
// 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 call Foo.bar 3

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

// push local 4
... // generated ... that implements push local 4

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

The VM translator

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 call Foo.bar 3

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

// push local 4
... // generated ... that implements push local 4

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

...
```

The VM translator:

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

The VM translator

Source language:

Arithmetic / Logical commands

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

Memory access commands

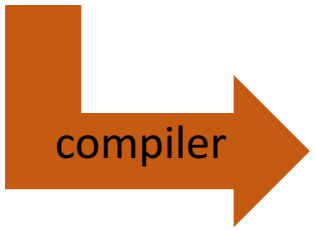
```
pop segment i
push segment i
```

Branching commands

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

Function commands

```
function functionName nVars
call functionName nArgs
return
```



Target language:

A instruction: `@value` where *value* is either a non-negative decimal constant or a symbol referring to such a constant

Semantics:
- sets the `A` register to *value*;
- makes `M` the `RAM` location whose address is *value*. (`M` stands for `RAM[A]`)

C instruction: *dest* `=` *comp* `;` *jump* (*dest* and *jump* are optional)

where:

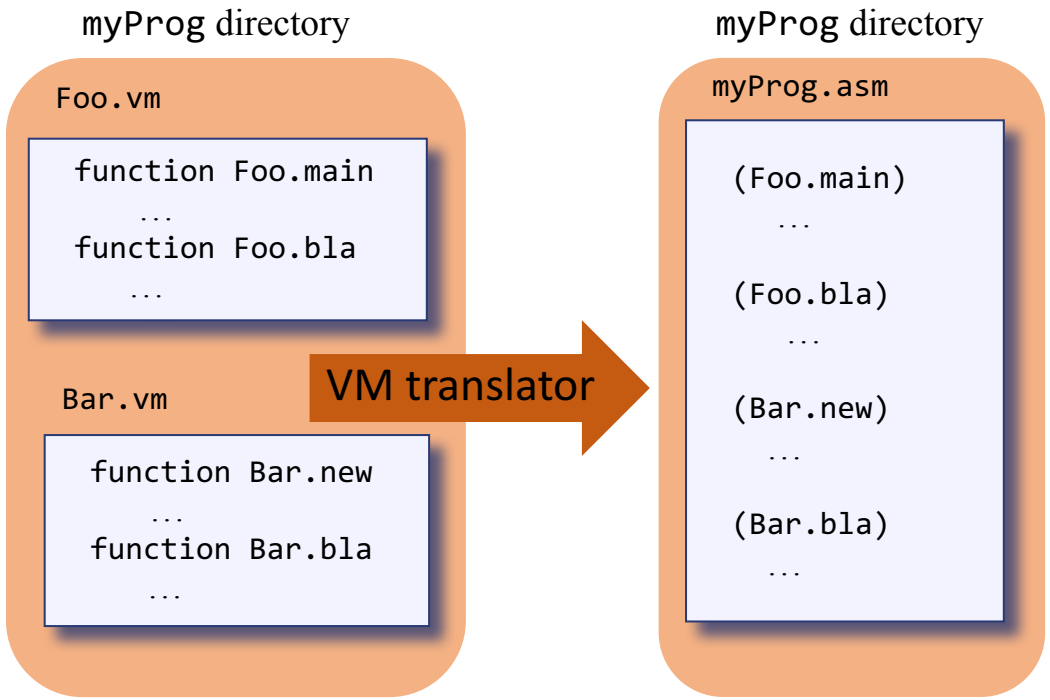
comp =
```
0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D&A, D|A,
   M,      !M,      -M,      M+1,      M-1,      D+M,  D-M,  M-D,  D&M,  D|M
```

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

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

Semantics:
- computes the value of *comp* and stores the result in *dest*;
- if (*comp jump* `0`) is true, jumps to execute the instruction in `ROM[A]`.

The VM translator



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`)

same design as the basic VM translator built in project 7

Main

Input:
- *fileName*`.vm` : the name of a single source file, or
- *directoryName*: the name of a directory containing one or more `.vm` source files

Output:
- *fileName*`.asm` file, or
- *directoryName*`.asm` file

Process:

- Constructs a `CodeWriter`
- If the input is a `.vm` file:
  - Constructs a `Parser` to handle the input file
  - Marches through the input file, parsing each line and generating code from it
- If the input is a directory:
  - Handles every `.vm` file in the directory in the manner described above.

Implementation note:

An extension of the `Main` program written in project 7.

Parser

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

Implementation notes

- Same `Parser` that was implemented in project 7
- If the `Parser` that you've developed in project 7 does not handle the parsing of the VM commands `goto`, `if-goto`, `label`, `call`, `function`, and `return`, add this parsing functionality now.

CodeWriter

The API of the basic `CodeWriter` (developed in project 7):

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.

CodeWriter

Additional functionality:

Routine	Arguments	Returns	Function
`setFileName`	`fileName` (string)	—	Informs the *codeWriter* that the translation of a new VM file has started (called by the main program of the VM translator).
`writeInit`	—	—	Writes the assembly instructions that effect the *bootstrap code* that initializes the VM. This code must be placed at the beginning of the generated `*.asm` file.
`writeLabel`	`label` (string)	—	Writes assembly code that effects the `label` command.
`writeGoto`	`label` (string)	—	Writes assembly code that effects the `goto` command.
`writeIf`	`label` (string)	—	Writes assembly code that effects the `if-goto` command.
`writeFunction`	`functionName`(string) `numVars` (int)	—	Writes assembly code that effects the `function` command.
`writeCall`	`functionName`(string) `numArgs` (int)	—	Writes assembly code that effects the `call` command.
`writeReturn`	—	—	Writes assembly code that effects the `return` command.

The generated assembly code must follow the guidelines and symbols described in the "standard mapping of the VM on the Hack platform" contract.

Virtual machine: lecture plan

Overview

- Program control

Branching

- Abstraction
- Implementation

Functions

- Abstraction
- Implementation

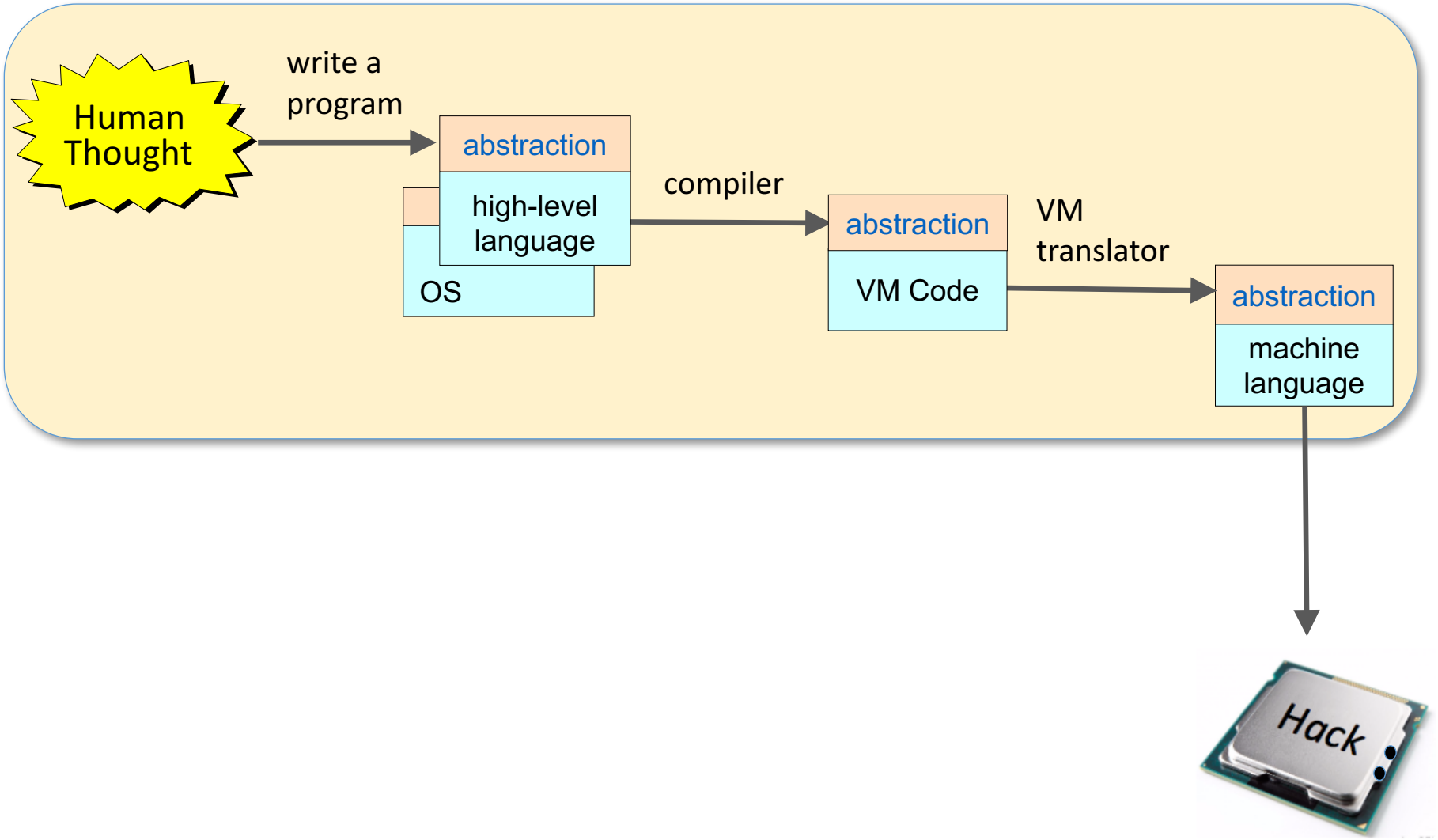
Implementing function call-and-return:

- Implementation overview
- Run-time simulation
- Detailed implementation

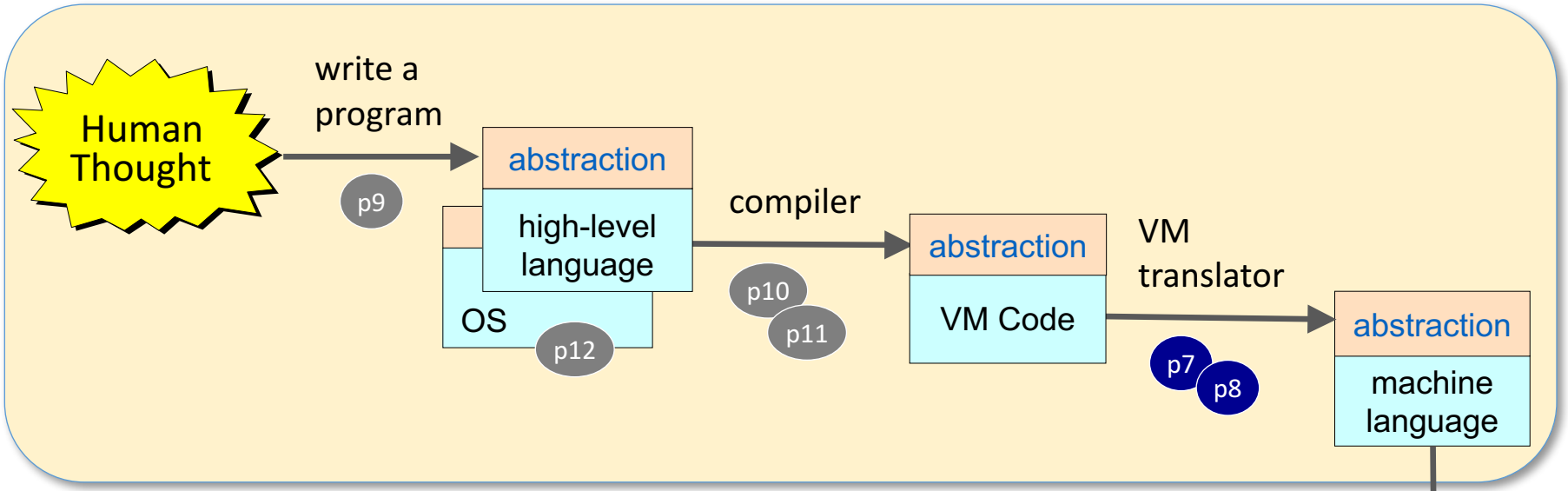
VM implementation on the Hack platform:

- Standard mapping
- VM translator:
  proposed implementation
- Project 8 overview

The big picture



The big picture

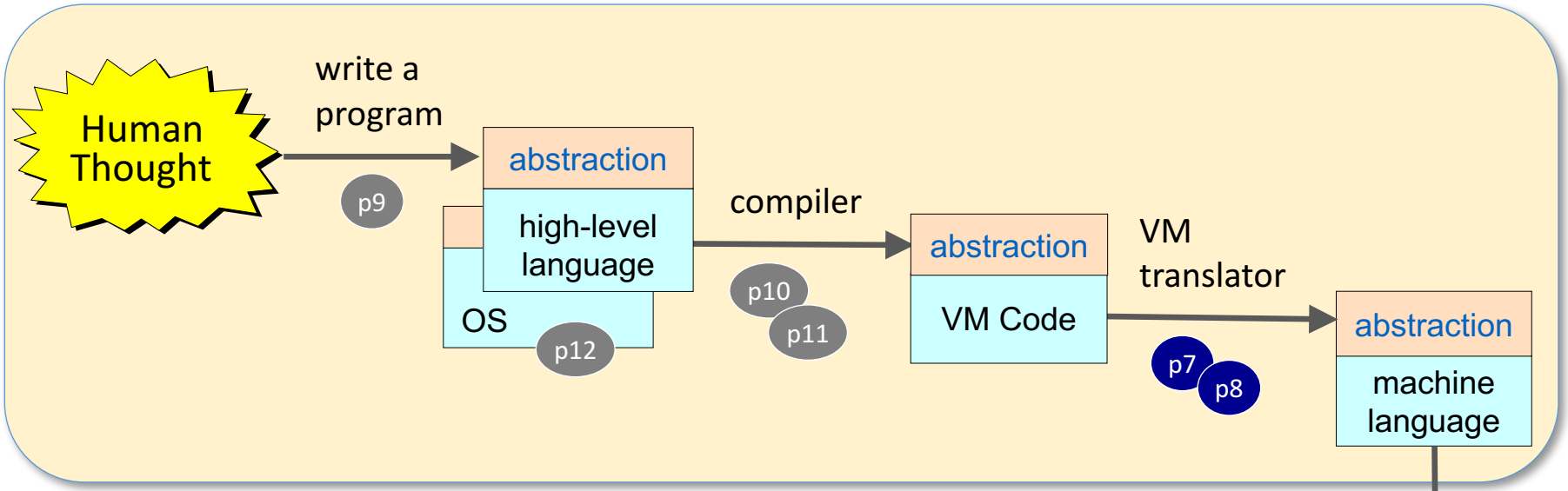


Objective: build a VM translator that translates programs written in the VM language into programs written in the Hack assembly language

To test the translator:

Run the generated nachine-level code on the target platform:

The big picture



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

To test the translator:

Run the generated nachine-level code on the target platform:

Test programs

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Test programs: BasicLoop

ProgramFlow:

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

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Testing routine

- Load and run XxxVME.tst on the VM emulator;
  This script loads Xxx.vm into the VM emulator, allowing you to experiment with its code

Test programs: BasicLoop

ProgramFlow:

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

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Testing routine

- Load and run XxxVME.tst on the VM emulator;
  This script loads Xxx.vm into the VM emulator, allowing you to experiment with its code
- Use your VM translator to translate Xxx.vm;
  The result will be a file named Xxx.asm

Test programs: BasicLoop

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

FunctionCalls:
- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Testing routine

- Load and run XxxVME.tst on the VM emulator;
  This script loads Xxx.vm into the VM emulator, allowing you to experiment with its code
- Use your VM translator to translate Xxx.vm;
  The result will be a file named Xxx.asm
- Load and run Xxx.tst on the CPU emulator;
  This script is designed to load Xxx.asm, execute it, and compares its output to Xxx.cmp

Test programs: BasicLoop

ProgramFlow:

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

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Typical loop logic:

```
while (counter > 0) {
      sum += counter
      counter-- }
```

BasicLoop.vm

```
// Computes the sum 1 + 2 + ... + argument[0],
// and pushes the  result onto the stack.
// The translated version, BasicLoop.asm (not shown here), can be tested on
// the CPU emulator.
// This can be done using BasicLoop.tst. This script initializes the VM
// memory segments as well as argument[0], and then loads and executes
// BasicLoop.asm.
   push constant 0
   pop local 0          // initialize sum = 0
label LOOP_START
   push argument 0
   push local 0
   add
   pop local 0          // sum = sum + counter
   push argument 0
   push constant 1
   sub
   pop argument 0       // counter--
   push argument 0
if-goto LOOP_START  // if counter > 0, goto LOOP_START
   push local 0
```

Test programs: BasicLoop

ProgramFlow:

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

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Typical loop logic:

```
while (counter > 0) {
      sum += counter
      counter-- }
```

Designed to test the handling of the VM commands:

- label
- if-goto

BasicLoop.vm

```
// Computes the sum 1 + 2 + ... + argument[0],
// and pushes the  result onto the stack.
// The translated version, BasicLoop.asm (not shown here), can be tested on
// the CPU emulator.
// This can be done using BasicLoop.tst. This script initializes the VM
// memory segments as well as argument[0], and then loads and executes
// BasicLoop.asm.
   push constant 0
   pop local 0          // initialize sum = 0
label LOOP_START
   push argument 0
   push local 0
   add
   pop local 0          // sum = sum + counter
   push argument 0
   push constant 1
   sub
   pop argument 0       // counter--
   push argument 0
if-goto LOOP_START  // if counter > 0, goto LOOP_START
   push local 0
```

Test programs

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

Test programs: FibonacciSeries

- ProgramFlow:
  - BasicLoop
  - FibonacciSeries
    - FibSeries.vm
    - FibSeriesVME.tst
    - FibSeries.tst
    - FibSeries.cmp
- FunctionCalls:
  - SimpleFunction
  - NestedCall
  - FibonacciElement
  - StaticsTest

typical array processing logic:

`arr[i] = arr[i-1] + arr[i-2]`

FibSeries.vm

```
// Computes the first argument[0] elements of the Fibonacci series.
// Puts the elements in the RAM, starting at the address given in argument[1].

// FibSeries.tst initializes the VM memory segments as well as
// argument[0] and argument[1], and then executes the translated
// machine code, stored in FibSeries.asm, on the CPU emulator.

   push argument 1
   pop pointer 1           // that = argument[1]
   push constant 0
   pop that 0              // first series element = 0
   push constant 1
   pop that 1              // second series element = 1
   ...
label MAIN_LOOP_START
   push argument 0
   if-goto COMPUTE_ELEMENT // if ... goto COMPUTE_ELEMENT
   goto END_PROGRAM        // otherwise, goto END_PROGRAM

label COMPUTE_ELEMENT
   push that 0
   push that 1
   add
   ...
   goto MAIN_LOOP_START

label END_PROGRAM
```

Test programs: FibonacciSeries

ProgramFlow:

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

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

typical array processing logic:

`arr[i] = arr[i-1] + arr[i-2]`

Designed to perform a more elaborate test of handling the VM commands:

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

FibSeries.vm

```
// Computes the first argument[0] elements of the Fibonacci series.
// Puts the elements in the RAM, starting at the address given in argument[1].

// FibSeries.tst initializes the VM memory segments as well as
// argument[0] and argument[1], and then executes the translated
// machine code, stored in FibSeries.asm, on the CPU emulator.

   push argument 1
   pop pointer 1           // that = argument[1]
   push constant 0
   pop that 0              // first series element = 0
   push constant 1
   pop that 1              // second series element = 1
   ...
label MAIN_LOOP_START
   push argument 0
   if-goto COMPUTE_ELEMENT // if ... goto COMPUTE_ELEMENT
   goto END_PROGRAM        // otherwise, goto END_PROGRAM

label COMPUTE_ELEMENT
   push that 0
   push that 1
   add
   ...
   goto MAIN_LOOP_START

label END_PROGRAM
```

Test programs: SimpleFunction

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

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

Test programs: SimpleFunction

ProgramFlow:
- BasicLoop
- FibonacciSeries

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

SimpleFunction.vm

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

// SimpleFunction.tst initializes the VM memory segments as well
// as some argument values, and then executes the translated
// machine code, stored in SimpleFunction.asm,  in the CPU emulator.

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

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

Simple test, since it involves no caller

Test programs: FibonacciElement

ProgramFlow:
- BasicLoop
- FibonacciSeries

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

> VMTranslator FibonacciElement

Should yield a single output file:

FibonacciElement.asm

Test programs: FibonacciElement

ProgramFlow:
- BasicLoop
- FibonacciSeries

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

typical recursive logic:

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

Main.vm

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

Test programs: FibonacciElement

ProgramFlow:
- BasicLoop
- FibonacciSeries

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

- Tests the handling of `function`, `return`, and `call` (and many other VM commands)
- Tests that the VM implementation (your translator) initializes the memory segments
- Tests that the the bootsrap code of the VM implementation initializes the stack and calls `Sys.init`

`Main.vm` (abbreviated)

```
// Main.fibonacci: computes the n'th element of the Fibonacci series,
// recursively. The n value is supplied by the caller, and stored in
// argument 0.
function Main.fibonacci 0
   ...
   call Main.fibonacci 1  // computes fib(n-2)
   push argument 0
   push constant 1
   sub
   call Main.fibonacci 1  // computes fib(n-1)
   add                    // returns fib(n-1) + fib(n-2)
   return
```

`Sys.vm`

Normally, `Sys.init` should call the function `Main.main`.

In Project 8, we use it to call various test functions.

```
function Sys.init 0
   push constant 4
   call Main.fibonacci 1 // test: computes the 4'th Fib. element
label WHILE
   goto WHILE            // loops forever
```

Test programs: NestedCall

ProgramFlow:
- BasicLoop
- FibonacciSeries

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

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

Test programs: StaticsTest

ProgramFlow:

- BasicLoop
- FibonacciSeries

FunctionCalls:

- SimpleFunction
- NestedCall
- FibonacciElement
- StaticsTest

  Class1.vm
  Class2.vm
  Sys.vm
  StaticsTestVME.tst
  StaticsTest.tst
  StaticsTest.cmp

> VMTranslator StaticsTest

Should yield a single output file:

StaticsTest.asm

Test programs: StaticsTest

ProgramFlow:
- BasicLoop
- FibonacciSeries

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

class1.vm

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

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

Test programs: StaticsTest

ProgramFlow:
- BasicLoop
- FibonacciSeries

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

Class1.vm

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

// Returns (st
function C
    push st
    push st
    sub
    return
```

Class2.vm

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

// Returns (static 0) – (static 1)
function Class2.get 0
    push static 0
    push static 1
    sub
    return
```

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

Test programs: StaticsTest

ProgramFlow:
- BasicLoop
- FibonacciSeries

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

Tests that the static segments of different files are handled correctly.

Class1.vm (abbreviated)

```
// Stores two supplied arguments in static 0 and static 1
function Class1.set 0
...
// Returns (static 0) – (static 1)
function Class1.get 0
...
```

Class2.vm

```
// Stores two supplied arguments in static 0 and static 1
function Class1.set 0
```

Sys.vm

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

Tools and resources

- Test programs and compare files: `nand2tetris/projects/08`
- Reference: chapter 8 in *The Elements of Computing Systems*

Same as in project 7:

- Experimenting with the test VM programs: the supplied *VM emulator*
- Translating the test VM programs into assembly: your *VM translator*
- Testing the resulting assembly code: the supplied *CPU emulator*
- Programming language for implementing your VM translator: Java, Python, ...
- Tutorials: VM emulator, CPU emulator (`nand2tetris` web site).

Chapter 8

Virtual Machine, Part II

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