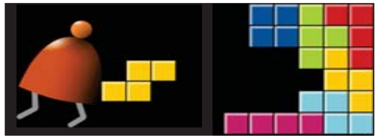
Virtual Machine

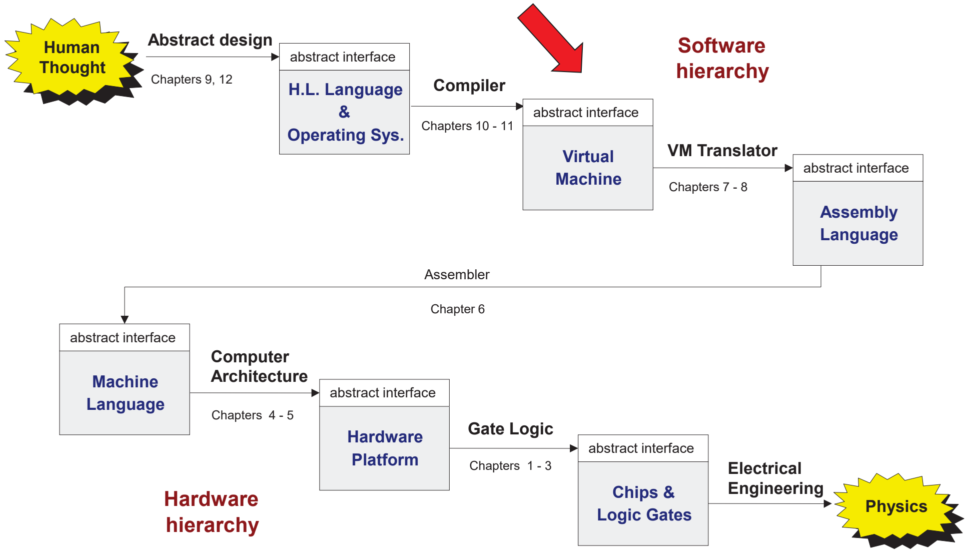
Part II: Program Control



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



Human Thought

Abstract design

Chapters 9, 12

abstract interface

H.L. Language & Operating Sys.

Compiler

Chapters 10 - 11

abstract interface

Virtual Machine

Software hierarchy

VM Translator

Chapters 7 - 8

abstract interface

Assembly Language

Assembler

Chapter 6

abstract interface

Machine Language

Computer Architecture

Chapters 4 - 5

abstract interface

Hardware Platform

Gate Logic

Chapters 1 - 3

abstract interface

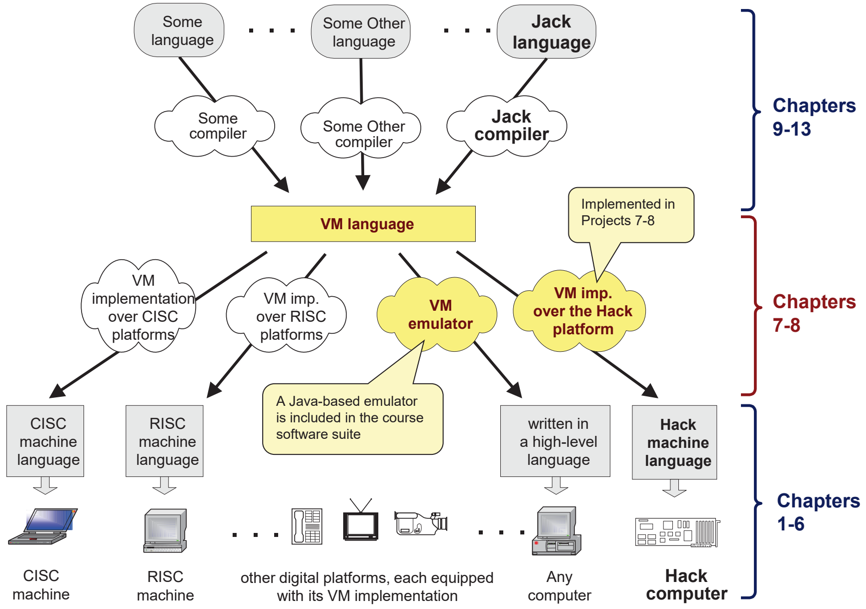
Chips & Logic Gates

Electrical Engineering

Physics

Hardware hierarchy

The big picture



Some language . . . Some Other language . . . Jack language

Some compiler

Some Other compiler

Jack compiler

VM language

Implemented in Projects 7-8

VM implementation over CISC platforms

VM imp. over RISC platforms

VM emulator

VM imp. over the Hack platform

A Java-based emulator is included in the course software suite

CISC machine language

RISC machine language

written in a high-level language

Hack machine language

CISC machine

RISC machine

other digital platforms, each equipped with its VM implementation

Any computer

Hack computer

Chapters 9-13

Chapters 7-8

Chapters 1-6

The VM language

Goal: Complete the specification and implementation of the VM model and language

Arithmetic / Boolean commands

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

Chapter 7

Memory access commands

```
pop x    (pop into x, which is a variable)
push y   (y being a variable or a constant)
```

Program flow commands

```
label     (declaration)
goto      (label)
if-goto   (label)
```

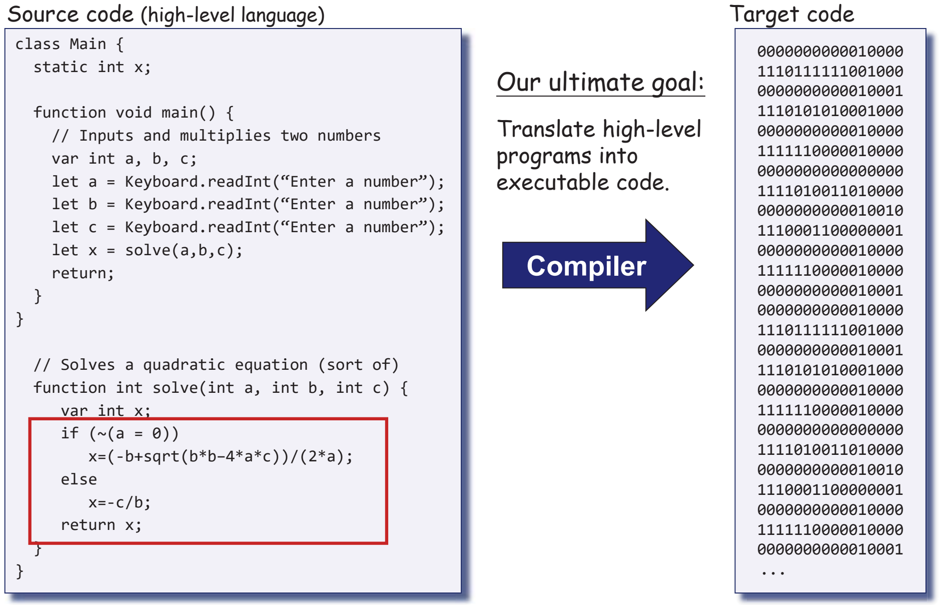
Chapter 8

Function calling commands

```
function  (declaration)
call      (a function)
return    (from a function)
```

Method: (a) specify the abstraction (model's constructs and commands)
(b) propose how to implement it over the Hack platform.

The compilation challenge



Source code (high-level language)

```
class Main {
  static int x;

  function void main() {
    // Inputs and multiplies two numbers
    var int a, b, c;
    let a = Keyboard.readInt(“Enter a number”);
    let b = Keyboard.readInt(“Enter a number”);
    let c = Keyboard.readInt(“Enter a number”);
    let x = solve(a,b,c);
    return;
  }
}

  // Solves a quadratic equation (sort of)
  function int solve(int a, int b, int c) {
     var int x;
     if (~(a = 0))
        x=(-b+sqrt(b*b–4*a*c))/(2*a);
     else
        x=-c/b;
     return x;
  }
}
```

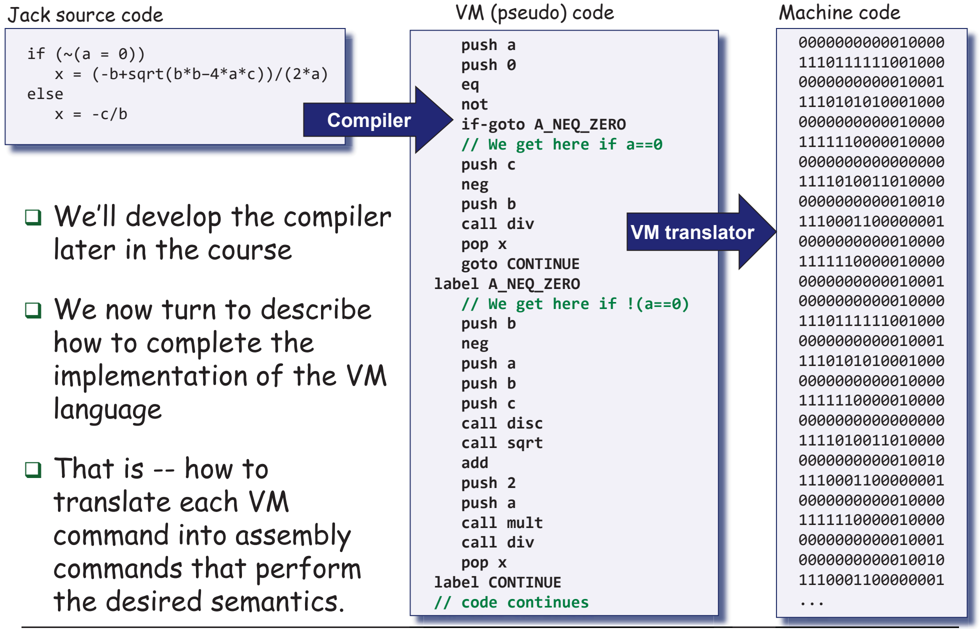
Our ultimate goal:

Translate high-level programs into executable code.

Target code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
...
```

The compilation challenge / two-tier setting



Jack source code

```
if (~(a = 0))
   x = (-b+sqrt(b*b-4*a*c))/(2*a)
else
   x = -c/b
```

VM (pseudo) code

```
   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
```

Machine code

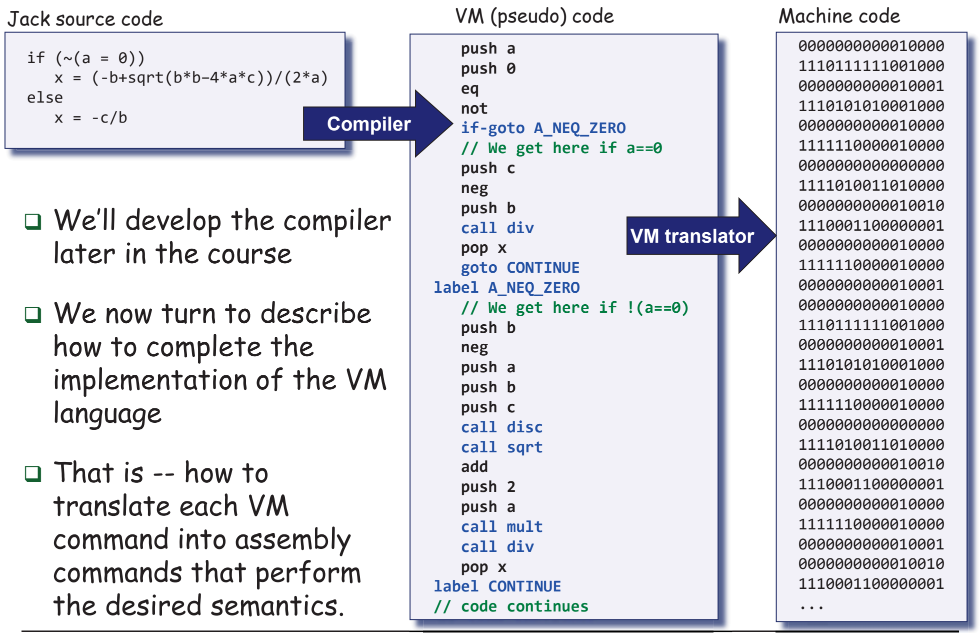
```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010010
1110001100000001
...
```

❑ We'll develop the compiler later in the course

❑ We now turn to describe how to complete the implementation of the VM language

❑ That is -- how to translate each VM command into assembly commands that perform the desired semantics.

The compilation challenge / two-tier setting



Jack source code

```
if (~(a = 0))
   x = (-b+sqrt(b*b-4*a*c))/(2*a)
else
   x = -c/b
```

VM (pseudo) code

```
   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
```

Machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010010
1110001100000001
...
```

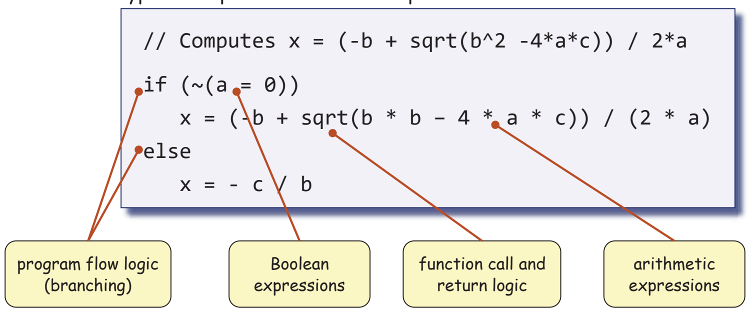
❑ We'll develop the compiler later in the course

❑ We now turn to describe how to complete the implementation of the VM language

❑ That is -- how to translate each VM command into assembly commands that perform the desired semantics.

The compilation challenge

Typical compiler's source code input:



```
// Computes x = (-b + sqrt(b^2 -4*a*c)) / 2*a

if (~(a = 0))
   x = (-b + sqrt(b * b – 4 * a * c)) / (2 * a)
else
   x = - c / b
```

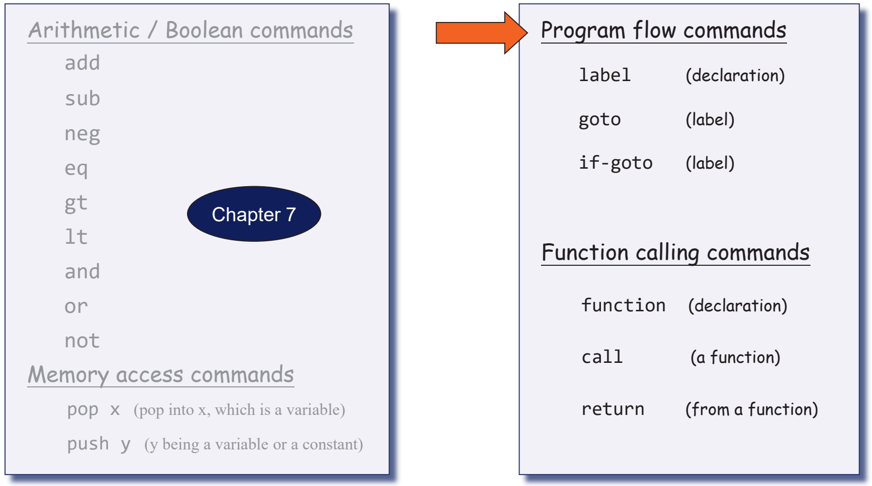
program flow logic (branching)
Boolean expressions
function call and return logic
arithmetic expressions

(this lecture) (previous lecture) **(this lecture)** (previous lecture)

How to translate such high-level code into machine language?

- In a two-tier compilation model, the overall translation challenge is broken between a *front-end* compilation stage and a subsequent *back-end* translation stage
- In our Hack-Jack platform, all the above sub-tasks (handling arithmetic / Boolean expressions and program flow / function calling commands) are done by the back-end, i.e. by the VM translator.

Lecture plan



Arithmetic / Boolean commands

add

sub

neg

eq

gt

lt

and

or

not

Memory access commands

pop x (pop into x, which is a variable)

push y (y being a variable or a constant)

Chapter 7

Program flow commands

`label`	(declaration)
`goto`	(label)
`if-goto`	(label)

Function calling commands

`function`	(declaration)
`call`	(a function)
`return`	(from a function)

Program flow commands in the VM language

VM code example:

```
function mult 1
  push constant 0
  pop local 0
label loop
  push argument 0
  push constant 0
  eq
  if-goto end
  push argument 0
  push 1
  sub
  pop argument 0
  push argument 1
  push local 0
  add
  pop local 0
  goto loop
label end
  push local 0
  return
```

In the VM language, the program flow abstraction is delivered using three commands:

```
label c      // label declaration

goto c       // unconditional jump to the
             // VM command following the label c

if-goto c    // pops the topmost stack element;
             // if it's not zero, jumps to the
             // VM command following the label c
```

How to translate these abstractions into assembly?

Flow of control

pseudo code

```
if (cond)
    statement1
else
    statement2
```

VM code

```
   ~cond
   if-goto elseLabel
   statement1
   goto contLabel
label elseLabel
   statement2
label contLabel
```

Flow of control

pseudo code

```
while (cond)
    statement
...
```

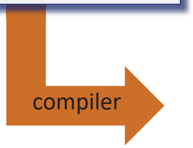
VM code

```
label contLabel
   ~(cond)
   if-goto exitLabel
   statement
   goto contLabel
label exitLabel
   ...
```

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 = 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
```

Label:

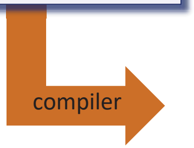
`label` *label*

defines a *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 = 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
```

Label:

`label` *label*

defines a *label.*

Unconditional branching:

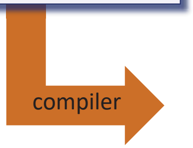
`goto` *label*

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 = 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
```

Label:

`label` *label*

defines a *label.*

Unconditional branching:

`goto` *label*

Jumps to execute the command just after *label.*

Conditional branching:

`if-goto` *label*

VM logic:

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

Lecture plan

Arithmetic / Boolean commands

add
sub
neg
eq
gt
lt
and
or
not

previous lecture

Memory access commands

`pop x` (pop into x, which is a variable)

`push y` (y being a variable or a constant)

Program flow commands

label	(declaration)
goto	(label)
if-goto	(label)

Function calling commands

function	(declaration)
call	(a function)
return	(from a function)

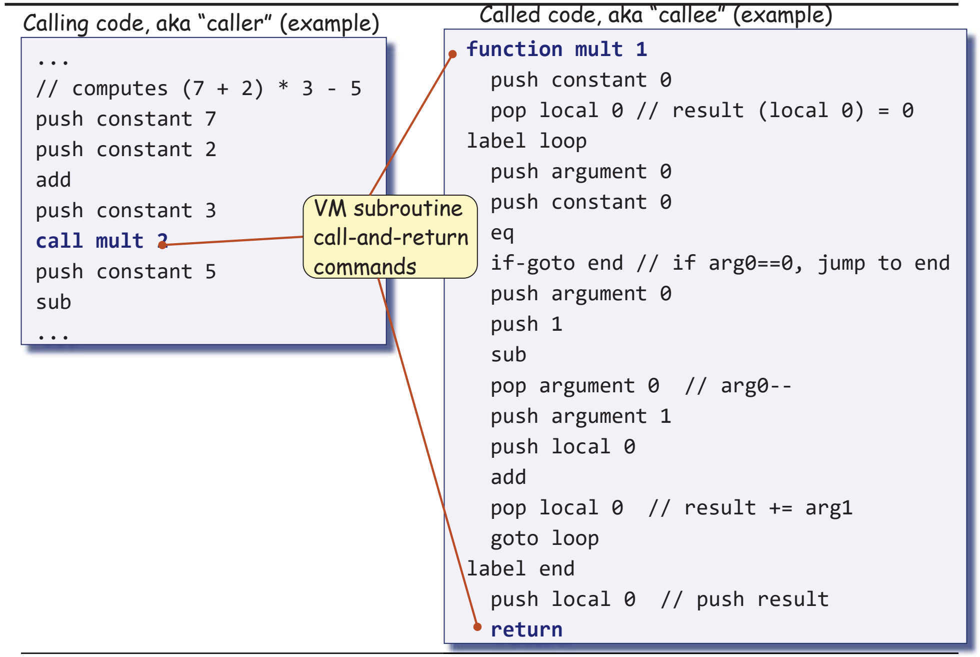
Subroutines

```
// Compute x = (-b + sqrt(b^2 -4*a*c)) / 2*a
if (~(a = 0))
   x = (-b + sqrt(b * b – 4 * a * c)) / (2 * a)
else
   x = - c / b
```

Subroutines = a major programming artifact

- Basic idea: the given language can be extended by user-defined commands (aka *subroutines/functions/procedures/methods* ...)
- Important: the language's primitive commands and the user-defined commands have the same look-and-feel
- This transparent extensibility is the most important abstraction delivered by high-level programming languages
- The challenge: implement this abstraction, i.e. allow the program control to flow effortlessly between one subroutine to the other

Subroutines in the VM language



Calling code, aka "caller" (example)

```
...
// computes (7 + 2) * 3 - 5
push constant 7
push constant 2
add
push constant 3
call mult 2
push constant 5
sub
...
```

VM subroutine call-and-return commands

Called code, aka "callee" (example)

```
function mult 1
  push constant 0
  pop local 0 // result (local 0) = 0
label loop
  push argument 0
  push constant 0
  eq
  if-goto end // if arg0==0, jump to end
  push argument 0
  push 1
  sub
  pop argument 0  // arg0--
  push argument 1
  push local 0
  add
  pop local 0  // result += arg1
  goto loop
label end
  push local 0  // push result
  return
```

Subroutines in the VM language

The invocation of the VM's primitive commands and subroutines follow exactly the same rules (consistent with other stack operations):

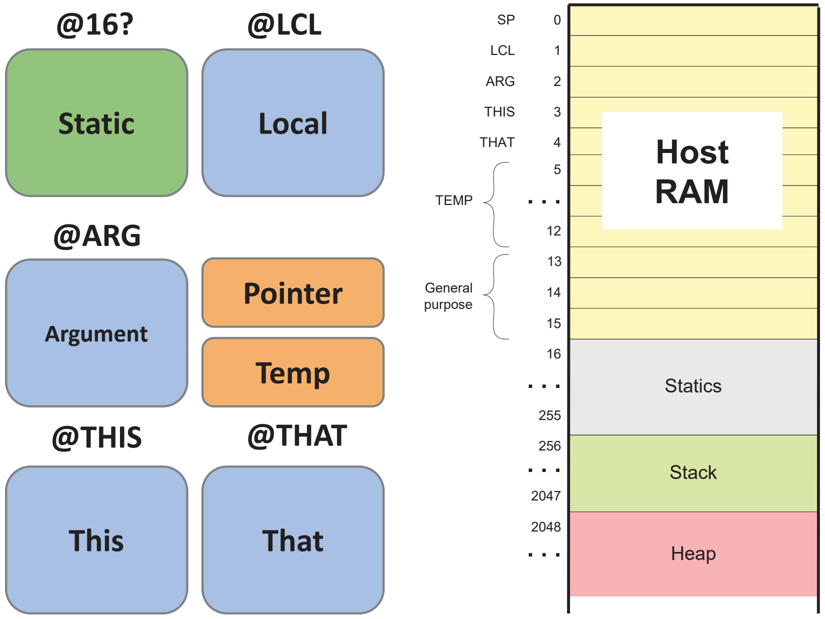
- The caller pushes the necessary argument(s) and calls the command / function for its effect
- The callee is responsible for removing the argument(s) from the stack, and for popping onto the stack the result of its execution.

What behind subroutines

The following scenario happens

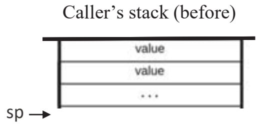
- The caller pushes the necessary arguments and call callee
- The state of the caller is saved
- The space of callee's local variables is allocated
- The callee executes what it is supposed to do
- The callee pushes the result to the stack
- Removes all arguments
- The space of the callee is recycled
- The caller's state is reinstalled
- Jump back to where is called

Memory Segments



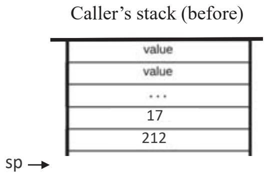
Function call and return: abstraction

Example: computing `mult(17,212)`



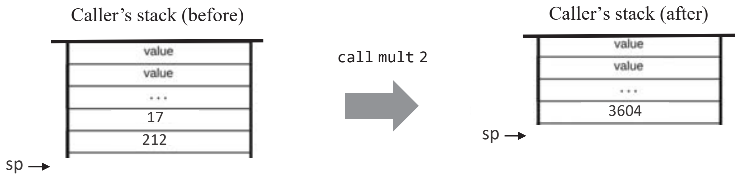
Function call and return: abstraction

Example: computing `mult(17,212)`



Function call and return: abstraction

Example: computing `mult(17,212)`

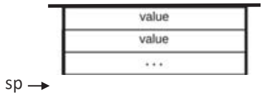


Net effect:

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

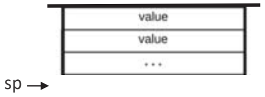
Function call and return: implementation

The function is running,
doing something



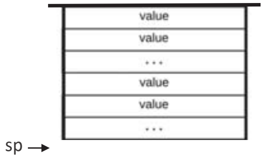
Function call and return: implementation

The function prepares
to call another function:



Function call and return: implementation

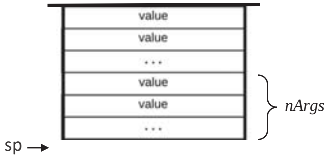
The function pushes arguments:



Function call and return: implementation

The function says:

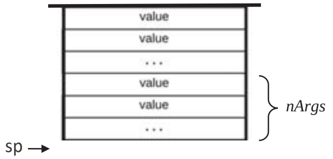
`call` *foo nArgs*



Function call and return: implementation

The function says:

`call` *foo nArgs*

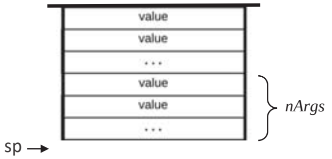


VM implementation (handling `call`):

Function call and return: implementation

The function says:

`call` *foo nArgs*



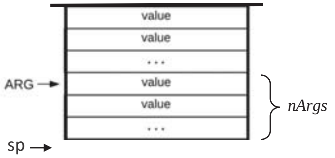
VM implementation (handling `call`):

1. Sets `ARG`

Function call and return: implementation

The function says:

`call` *foo nArgs*



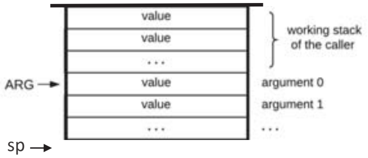
VM implementation (handling `call`):

1. Sets `ARG`

Function call and return: implementation

The function says:

`call` *foo nArgs*



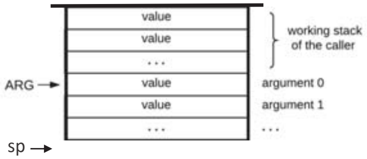
VM implementation (handling `call`):

1. Sets `ARG`

Function call and return: implementation

The function says:

`call` *foo nArgs*



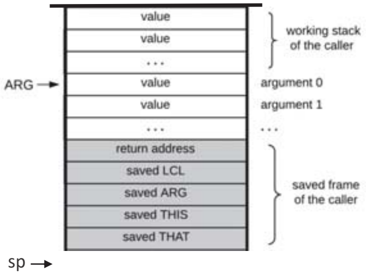
VM implementation (handling `call`):

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

Function call and return: implementation

The function says:

`call` *foo nArgs*



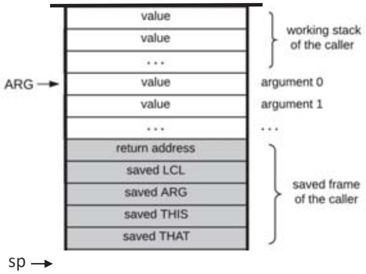
VM implementation (handling `call`):

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

Function call and return: implementation

The function says:

`call` *foo nArgs*



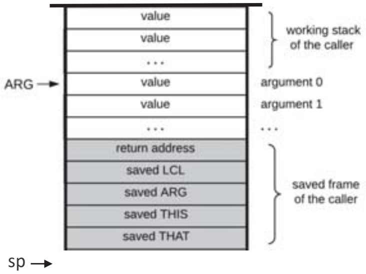
VM implementation (handling `call`):

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*



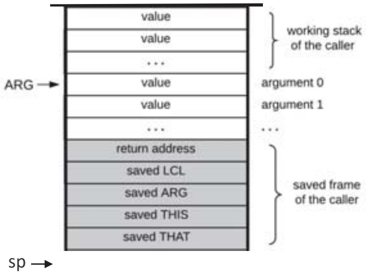
VM implementation (handling `call`):

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

Function call and return: implementation

The called function is entered:

`function` *foo nVars*



VM implementation (handling `call`):

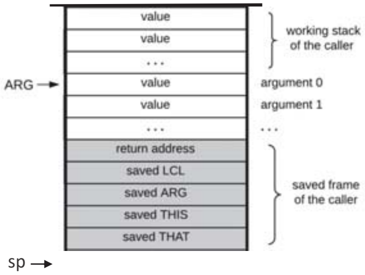
1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

VM implementation (handling `function`):

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*

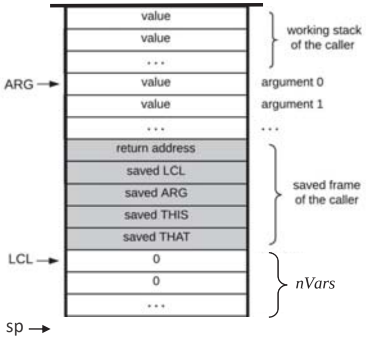
VM implementation (handling `function`):

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*

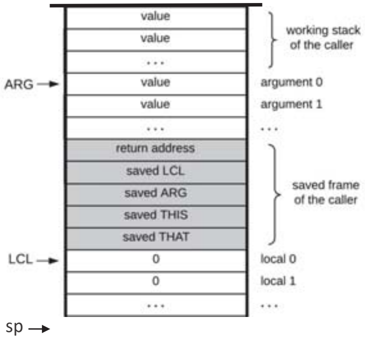
VM implementation (handling `function`):

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`):

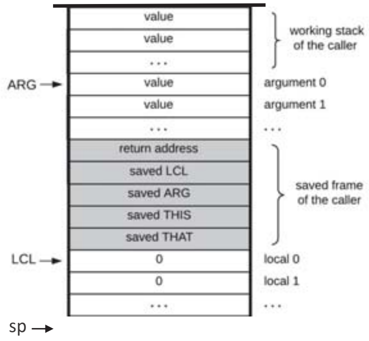
1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

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

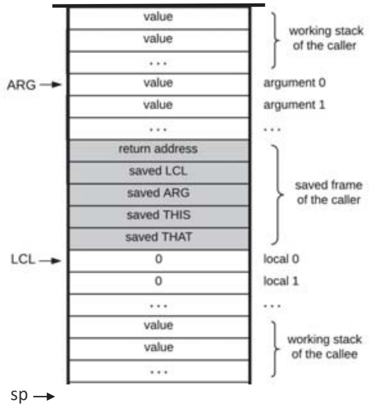

value
value
...
ARG → value — argument 0
value — argument 1
... ...
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0 — local 0
0 — local 1
... ...
sp →

working stack of the caller

saved frame of the caller

Function call and return: implementation

The called function is running, doing something


value
value
...
ARG → value — argument 0
value — argument 1
... ...
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0 — local 0
0 — local 1
... ...
value
value
...
sp →

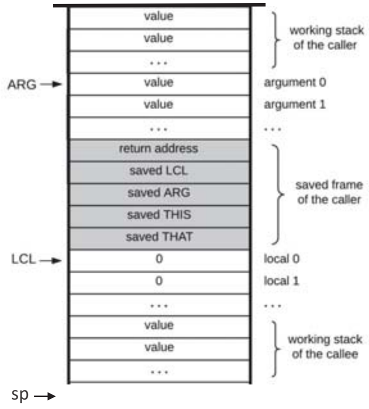
working stack of the caller

saved frame of the caller

working stack of the callee

Function call and return: implementation

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


value
value
...
ARG → value — argument 0
value — argument 1
... ...
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0 — local 0
0 — local 1
... ...
value
value
...
sp →

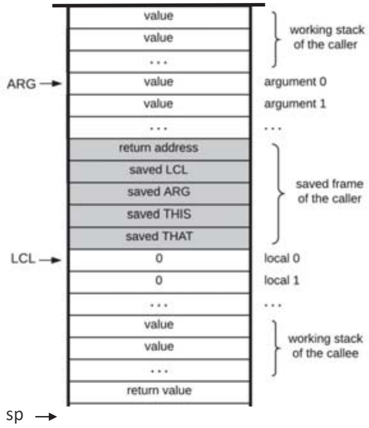
working stack of the caller

saved frame of the caller

working stack of the callee

Function call and return: implementation

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


value
value
...
ARG → value — argument 0
value — argument 1
... ...
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0 — local 0
0 — local 1
... ...
value
value
...
return value
sp →

working stack of the caller

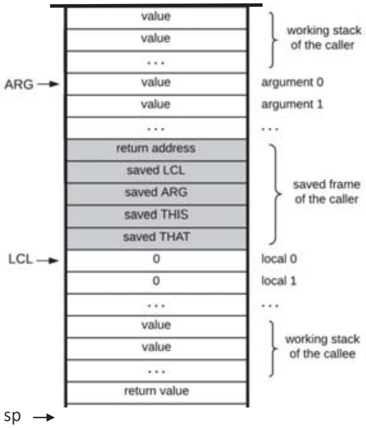
saved frame of the caller

working stack of the callee

Function call and return: implementation

The called function says:

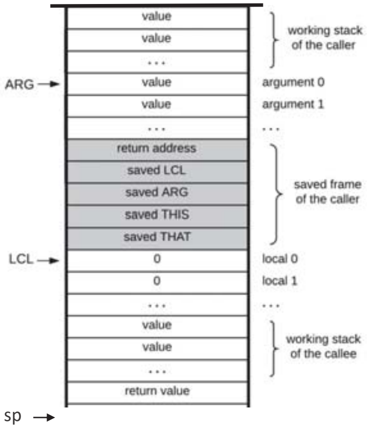
`return`



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 `function`):

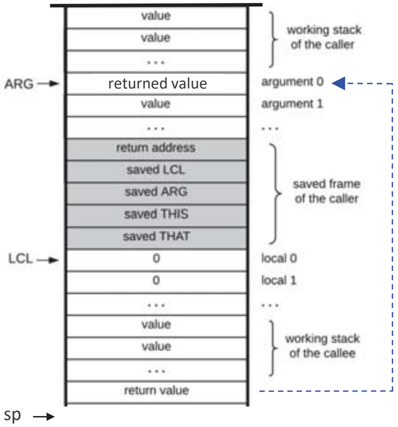
Sets up the `local` segment
of the called function

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 `function`):

Sets up the `local` segment
of the called function

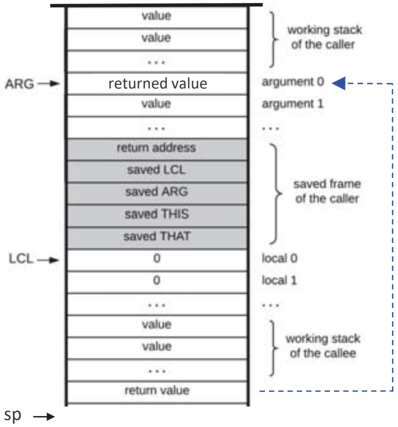
VM implementation (handling `return`):

1. Copies the return value onto `argument 0`

Function call and return: implementation

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 `function`):

Sets up the `local` segment
of the called function

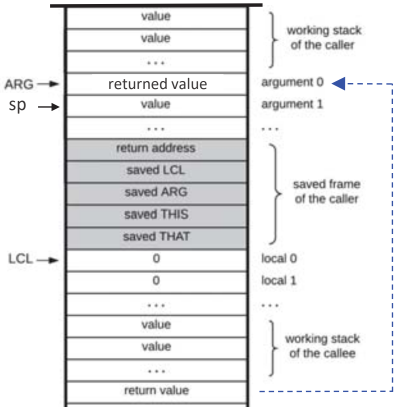
VM implementation (handling `return`):

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 `function`):

Sets up the `local` segment of the called function

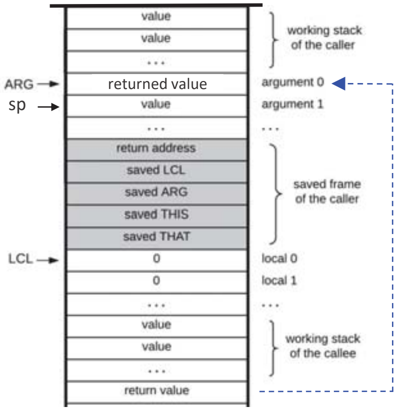
VM implementation (handling `return`):

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 `function`):

Sets up the `local` segment of the called function

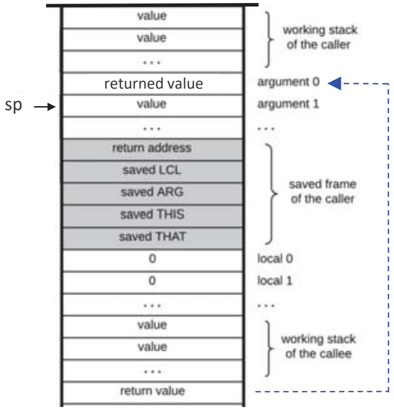
VM implementation (handling `return`):

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 `function`):

Sets up the `local` segment of the called function

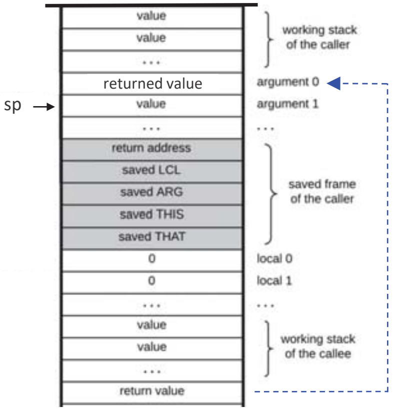
VM implementation (handling `return`):

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 `function`):

Sets up the `local` segment of the called function

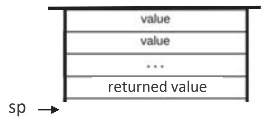
VM implementation (handling `return`):

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 `function`):

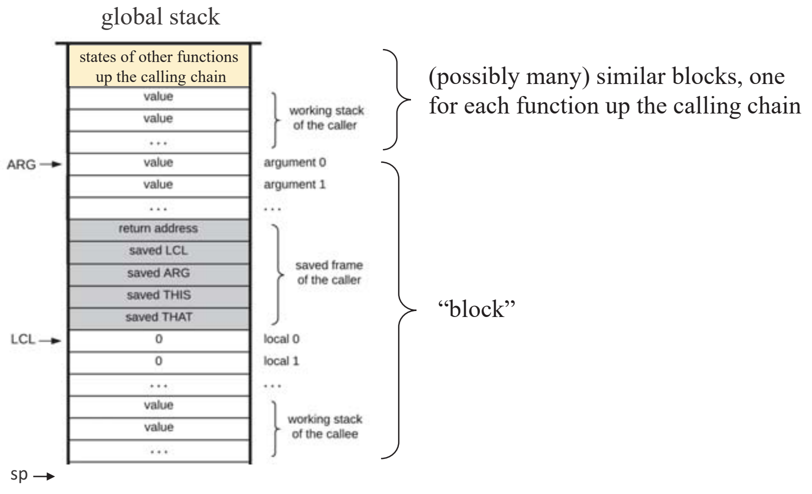
Sets up the `local` segment
of the called function

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


global stack
states of other functions up the calling chain
value
value
. . .
working stack of the caller
ARG
value
value
. . .
argument 0
argument 1
. . .
return address
saved LCL
saved ARG
saved THIS
saved THAT
saved frame of the caller
LCL
0
0
. . .
local 0
local 1
. . .
value
value
. . .
working stack of the callee
sp
(possibly many) similar blocks, one for each function up the calling chain
"block"

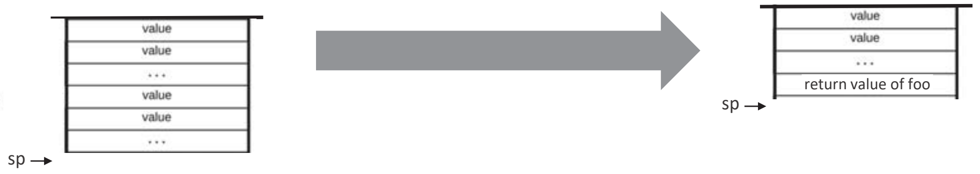
Recap: function call and return

The caller says:

`call` *foo nArgs*

Abstraction:

The caller resumes its execution



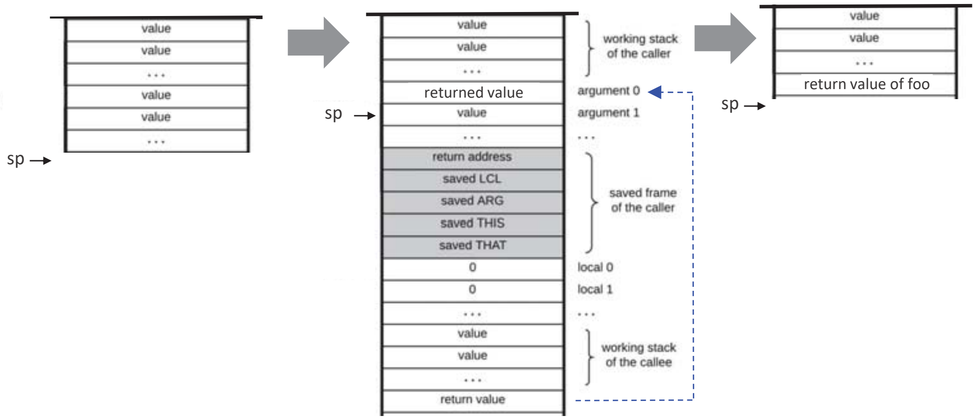
Recap: function call and return

The caller says:

`call` *foo nArgs*

Implementation:

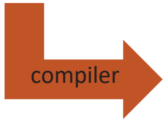
The caller resumes its execution



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
```

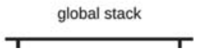
VM program

```
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
```

Example: `factorial`

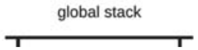
VM program

```
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
```



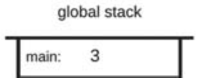
Example: `factorial`

```
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
```



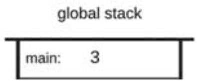
Example: `factorial`

```
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
```



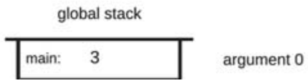
Example: factorial

```
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
```



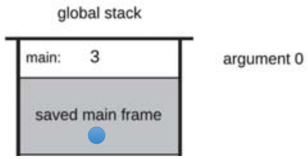
Example: factorial

```
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
```



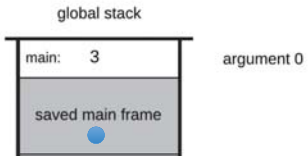
Example: factorial

```
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
```



Example: factorial

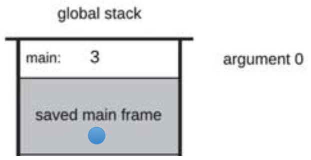
```
function main 0
   push constant 3
   call factorial 1
   return

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
```



Example: `factorial`

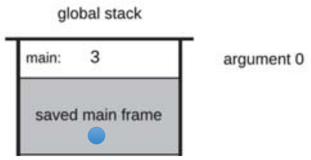
```
function main 0
   push constant 3
   call factorial 1
   return

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



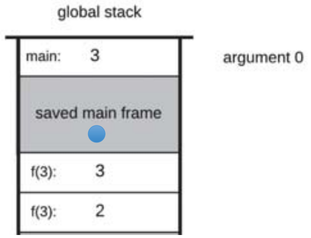
Example: `factorial`

```
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
```



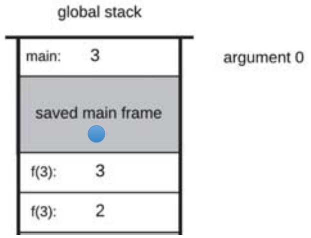
Example: `factorial`

```
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
```



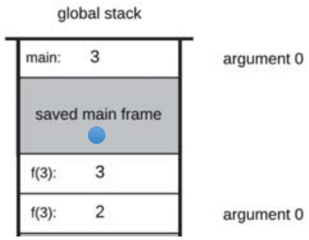
Example: `factorial`

```
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
```



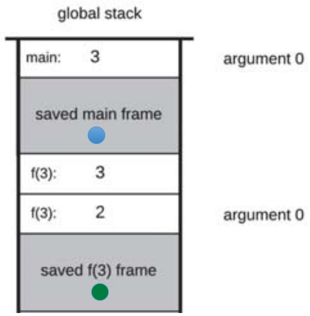
Example: factorial

```
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
```



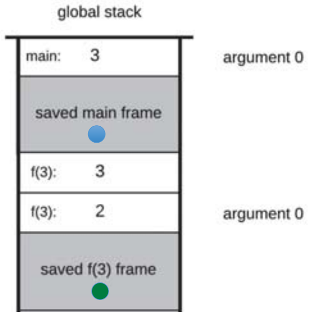
Example: factorial

```
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
```



Example: factorial

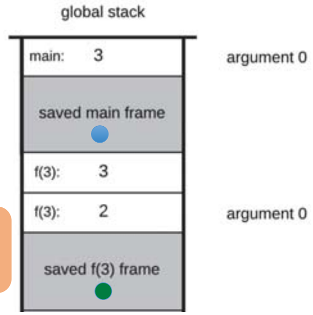
```
function main 0
   push constant 3
   call factorial 1
   return

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
```



Example: factorial

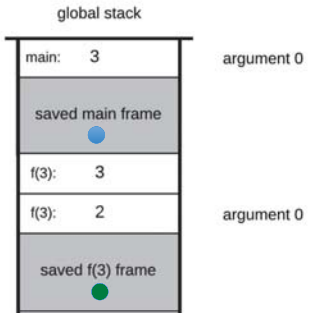
```
function main 0
   push constant 3
   call factorial 1
   return

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



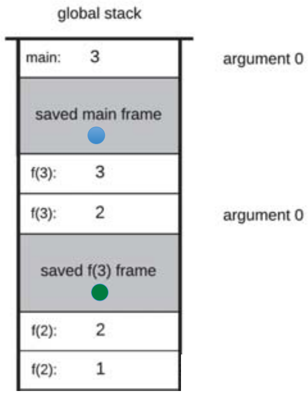
Example: factorial

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

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

function mult 2
   // Code omitted
```


global stack
main: 3
argument 0
saved main frame
f(3): 3
f(3): 2
argument 0
saved f(3) frame

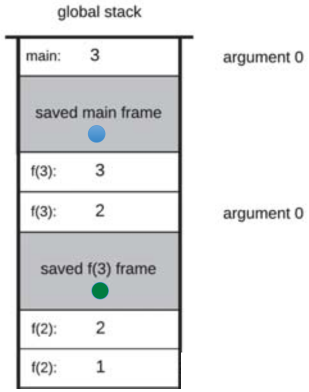
Example: factorial

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

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

function mult 2
   // Code omitted
```


global stack
main: 3
argument 0
saved main frame
f(3): 3
f(3): 2
argument 0
saved f(3) frame
f(2): 2
f(2): 1

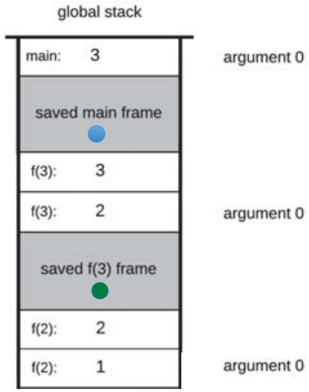
Example: factorial

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

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

function mult 2
   // Code omitted
```


global stack
main: 3
argument 0
saved main frame
f(3): 3
f(3): 2
argument 0
saved f(3) frame
f(2): 2
f(2): 1

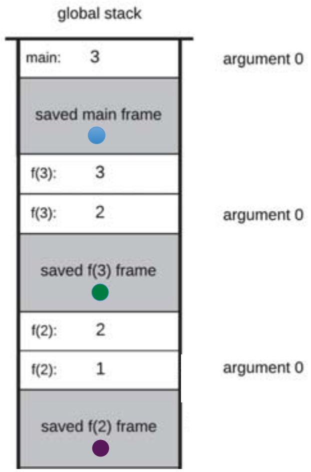
Example: factorial

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

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

function mult 2
   // Code omitted
```


global stack
main: 3
argument 0
saved main frame
f(3): 3
f(3): 2
argument 0
saved f(3) frame
f(2): 2
f(2): 1
argument 0

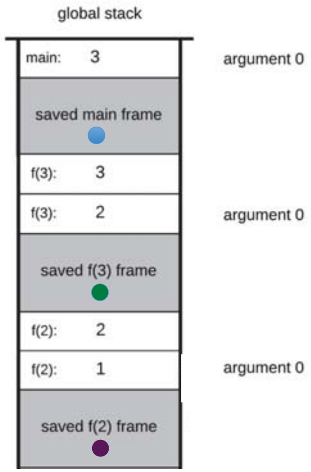
Example: factorial

```
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
```



Example: factorial

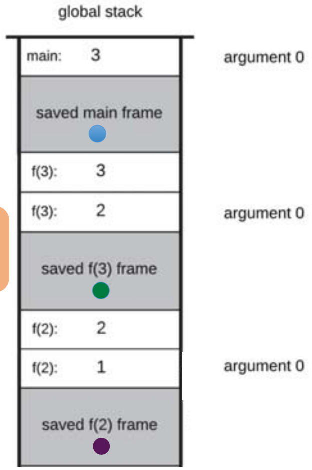
```
function main 0
    push constant 3
    call factorial 1
    return

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
```



Example: factorial

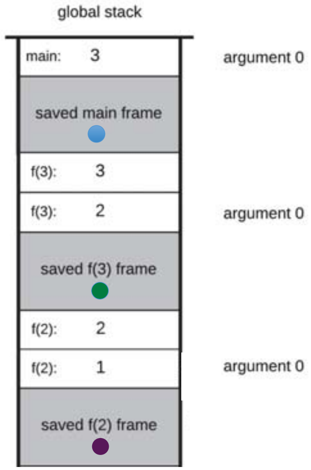
```
function main 0
    push constant 3
    call factorial 1
    return

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



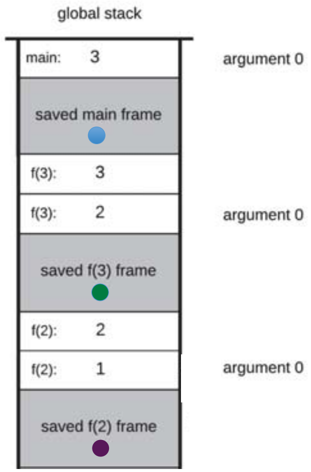
Example: factorial

```
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
```



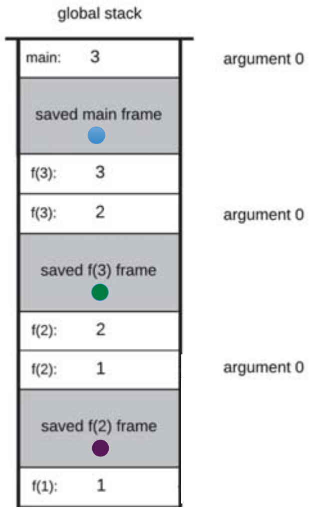
Example: factorial

```
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
```



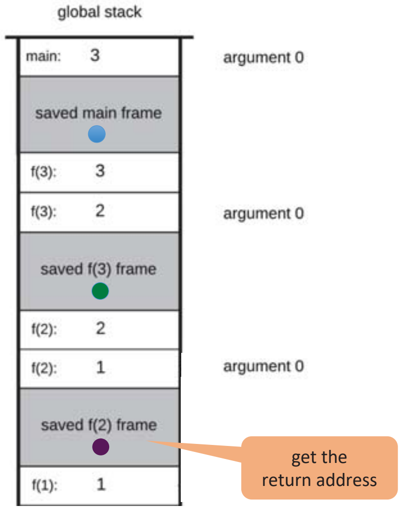
Example: factorial

```
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
```



Example: factorial

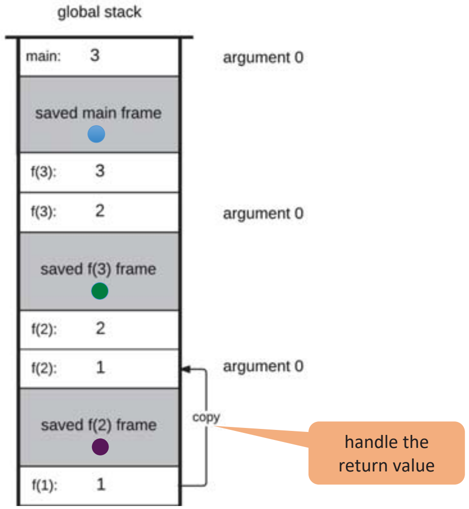
```
function main 0
   push constant 3
   call factorial 1
   return

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
```



get the return address

Example: factorial

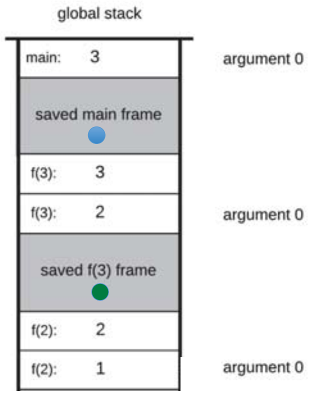
```
function main 0
   push constant 3
   call factorial 1
   return

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
```



handle the return value

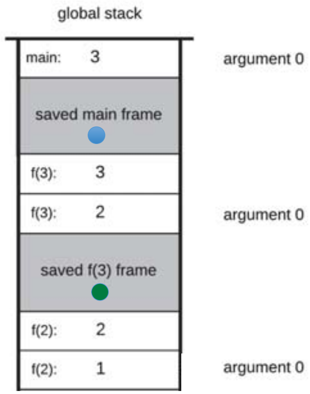
Example: factorial

```
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
```



Example: factorial

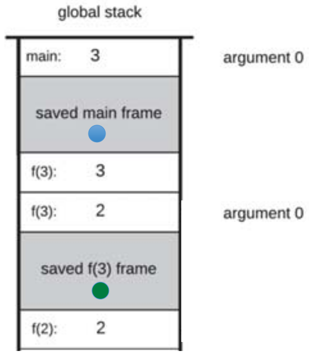
```
function main 0
   push constant 3
   call factorial 1
   return

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)

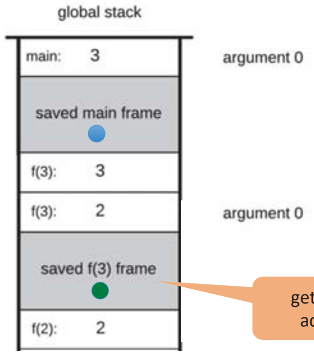
Example: factorial

```
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
```



Example: factorial

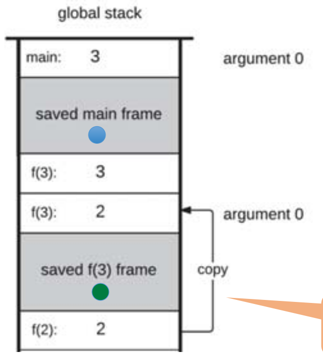
```
function main 0
   push constant 3
   call factorial 1
   return

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
```



get return address

Example: `factorial`

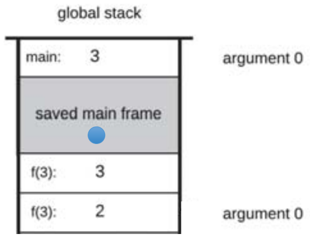
```
function main 0
   push constant 3
   call factorial 1
   return

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
```



handle the return value

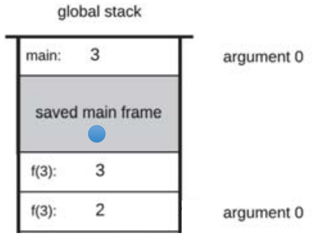
Example: `factorial`

```
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
```



Example: `factorial`

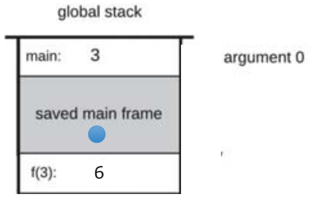
```
function main 0
   push constant 3
   call factorial 1
   return

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)

Example: `factorial`

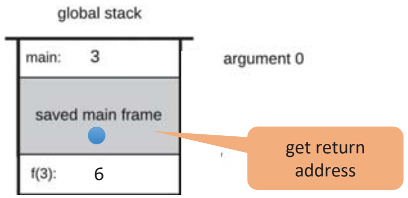
```
function main 0
   push constant 3
   call factorial 1
   return

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)

Example: `factorial`

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

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

function mult 2
   // Code omitted
```


global stack

main: 3

saved main frame

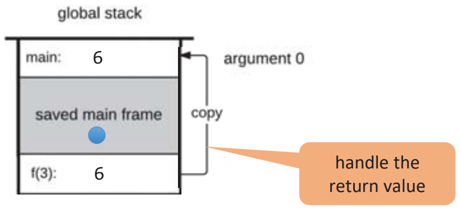
f(3): 6

argument 0

get return address

Example: `factorial`

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

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

function mult 2
   // Code omitted
```


global stack

main: 6

saved main frame

f(3): 6

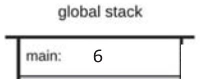
argument 0

copy

handle the return value

Example: `factorial`

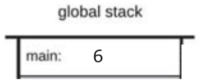
```
function main 0
   push constant 3
   call factorial 1
   return

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

function mult 2
   // Code omitted
```


global stack

main: 6

Example: `factorial`

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

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

function mult 2
   // Code omitted
```

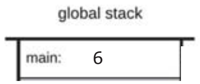

global stack

main: 6

Example: `factorial`

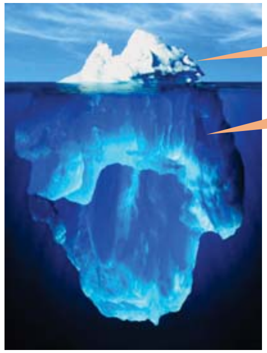
```
function main 0
   push constant 3
   call factorial 1
   return

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

function mult 2
   // Code omitted
```

global stack



The caller (`main` function) wanted to compute 3!

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



abstraction

implementation

Function commands in the VM language

```
function g nVars   // here starts a function called g,
                   // which has nVars local variables

call g nArgs       // invoke function g for its effect;
                   // nArgs arguments have already been pushed
                   // onto the stack

return             // terminate execution and return control
                   // to the caller
```

Q: Why this particular syntax?

A: Because it simplifies the VM implementation.

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).

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.

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

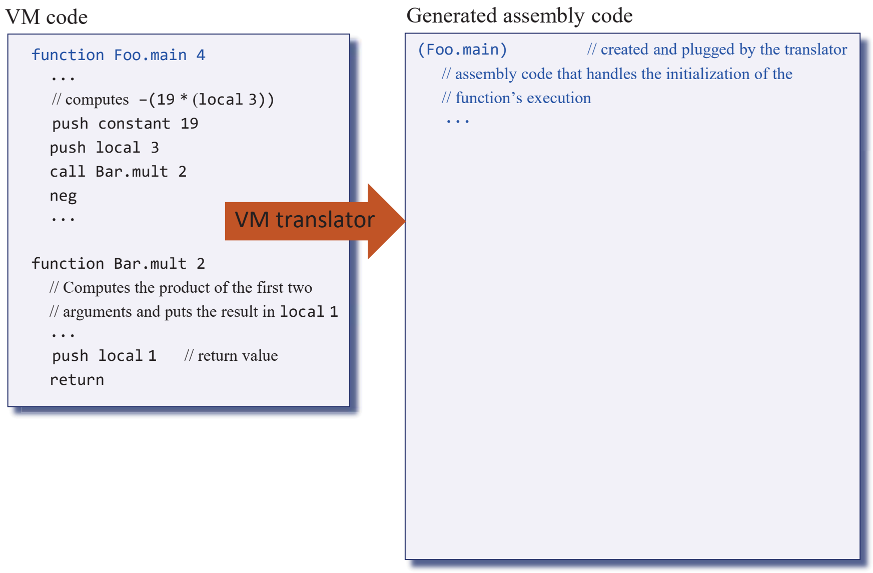
VM code

```
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
```

VM translator

Generated assembly code

The VM implementation view



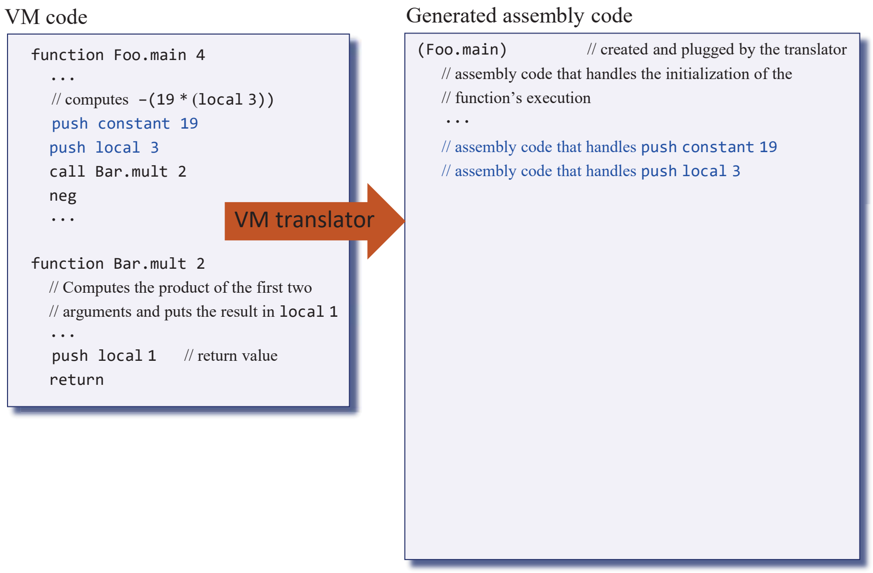
VM code

```
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
   ...
```

The VM implementation view



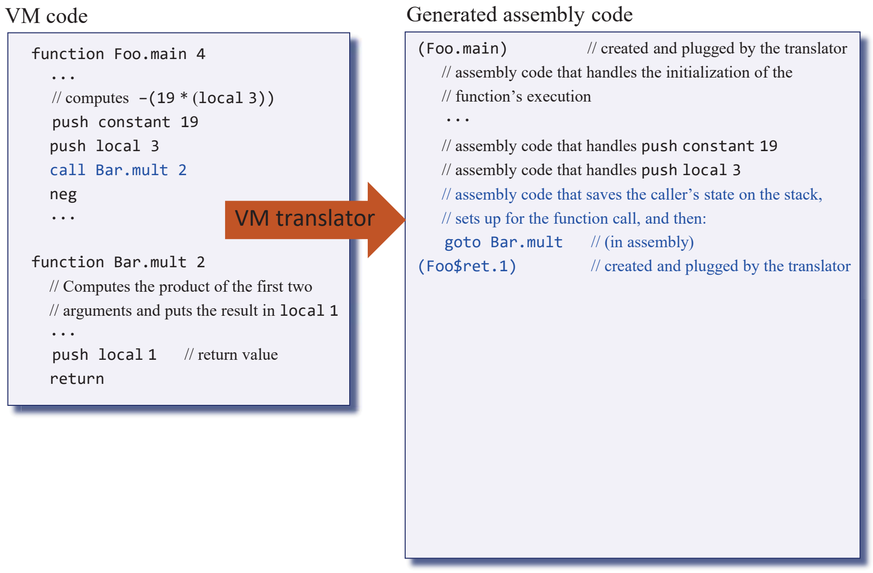
VM code

```
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
```

The VM implementation view



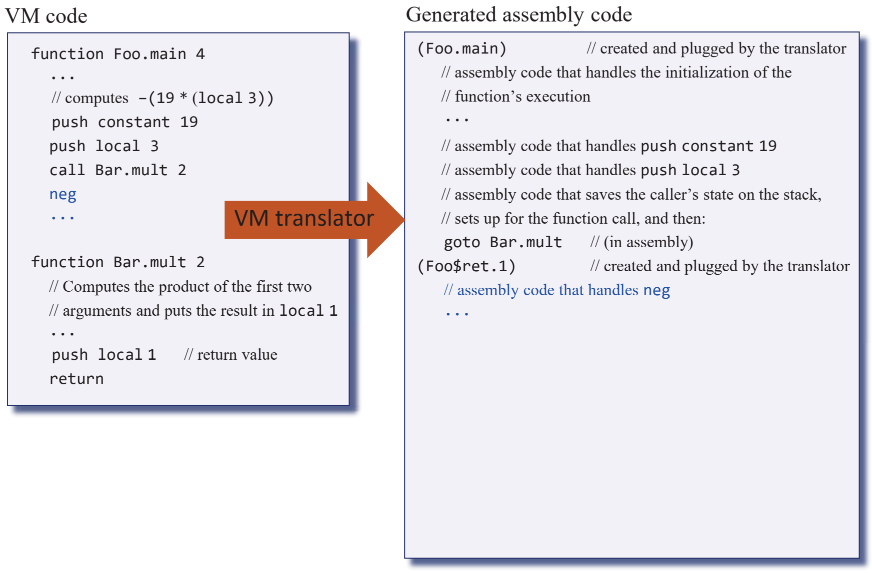
VM code

```
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
```

The VM implementation view



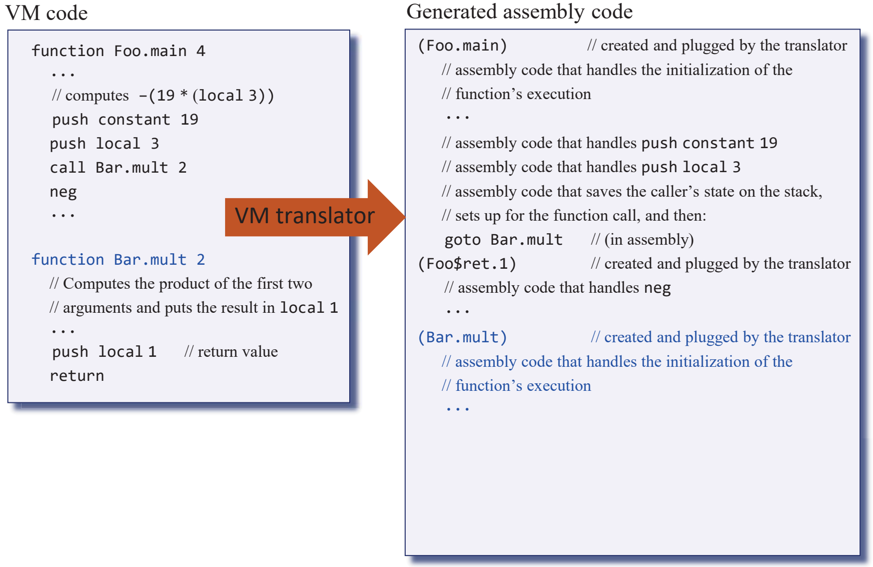
VM code

```
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
   ...
```

The VM implementation view



VM code

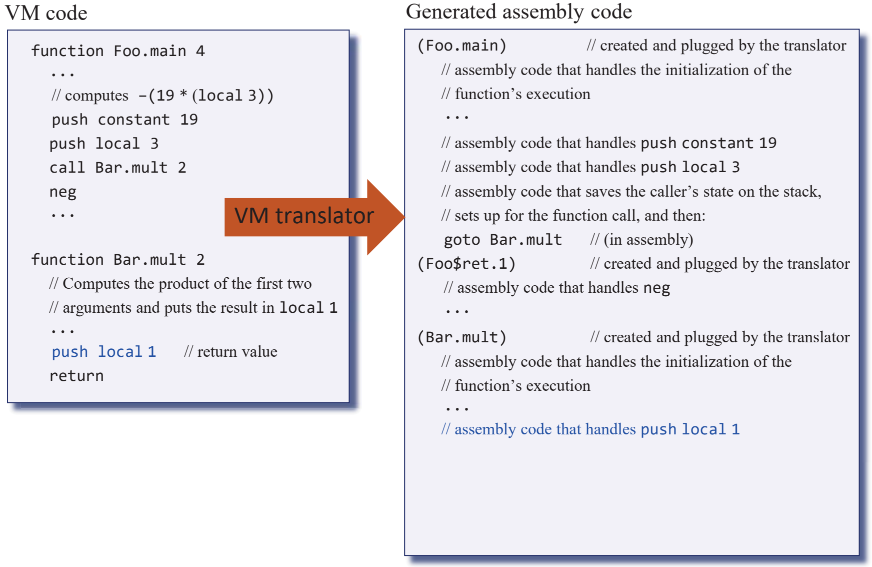
```
function Foo.main 4
  ...
  // computes -(19 * (local 3))
  push constant 19
  push local 3
  call Bar.mult 2
  neg
  ...

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

VM translator

Generated assembly code

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

The VM implementation view



VM code

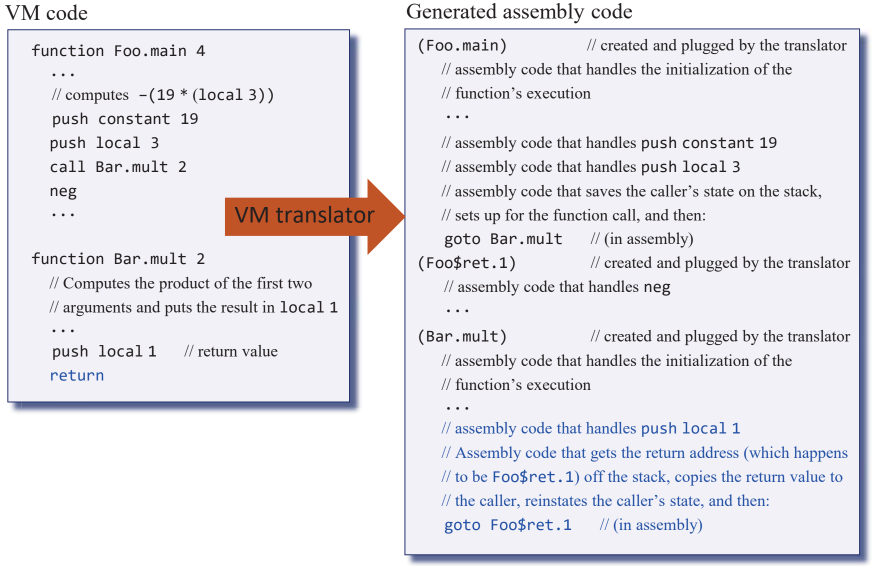
```
function Foo.main 4
  ...
  // computes -(19 * (local 3))
  push constant 19
  push local 3
  call Bar.mult 2
  neg
  ...

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

VM translator

Generated assembly code

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

The VM implementation view



VM code

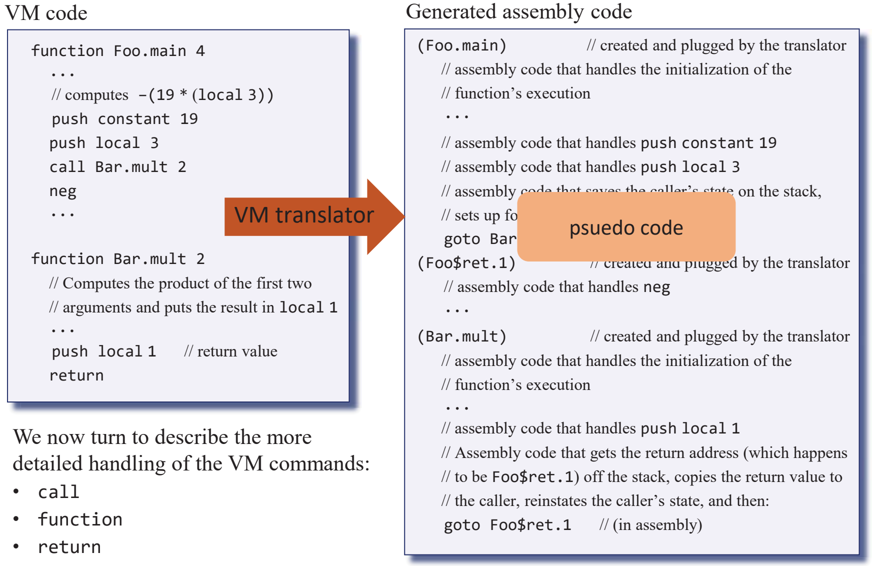
```
function Foo.main 4
  ...
  // computes -(19 * (local 3))
  push constant 19
  push local 3
  call Bar.mult 2
  neg
  ...

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

VM translator

Generated assembly code

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

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
```

VM translator

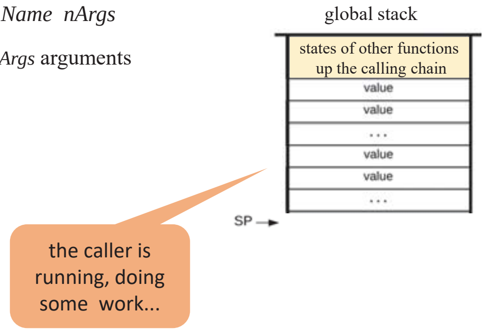
Generated assembly code

```
(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 `call`

VM command: `call` *functionName nArgs*

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

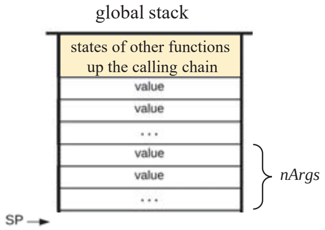


the caller is running, doing some work...

Handling `call`

VM command: `call` *functionName nArgs*

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

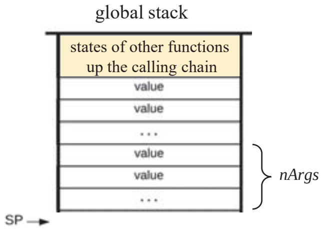


Handling `call`

VM command: `call` *functionName nArgs*

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

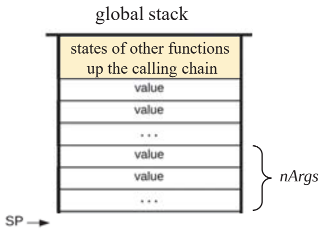
Assembly code (generated by the translator):



Handling `call`

VM command: `call` *functionName nArgs*

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



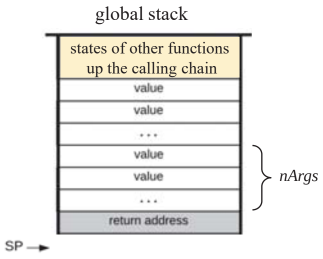
Assembly code (generated by the translator):

```
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)



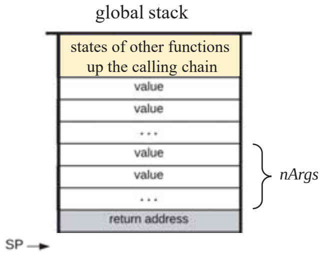
Assembly code (generated by the translator):

```
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)



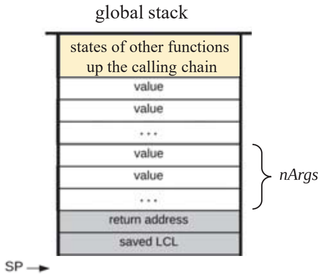
Assembly code (generated by the translator):

```
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
```

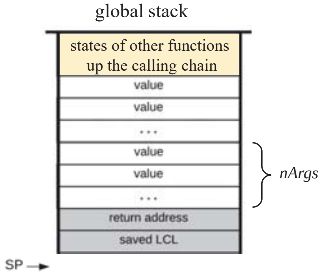
Handling `call`

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
```



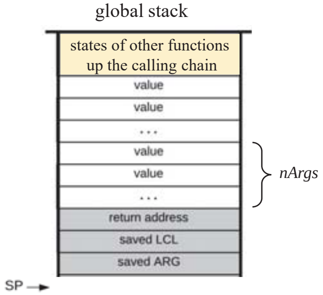
Handling `call`

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
```



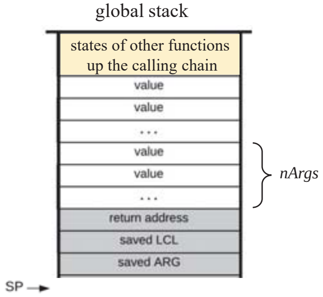
Handling `call`

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
```



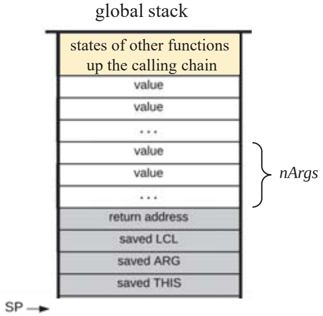
Handling `call`

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
```



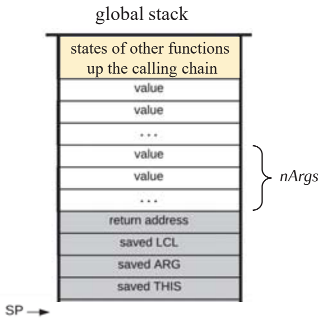
Handling call

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
```



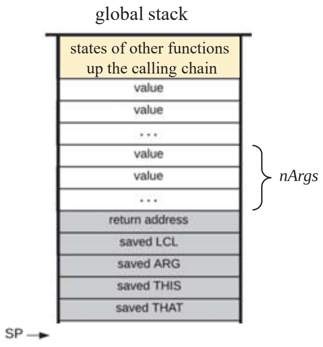
Handling call

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
```



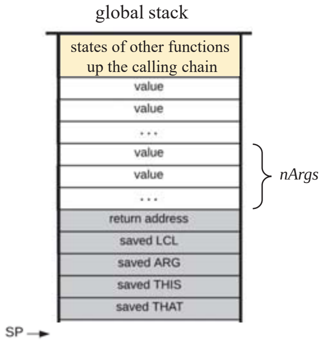
Handling call

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
```



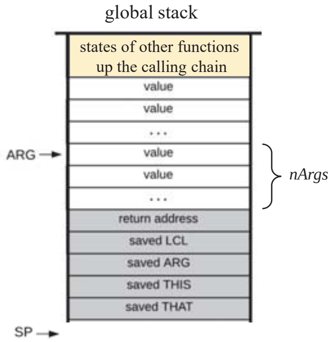
Handling call

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
```



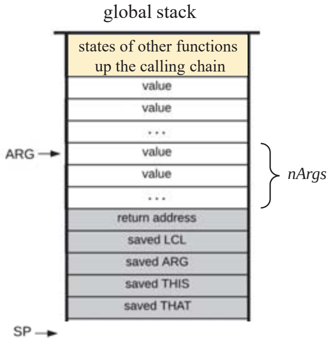
Handling call

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
```



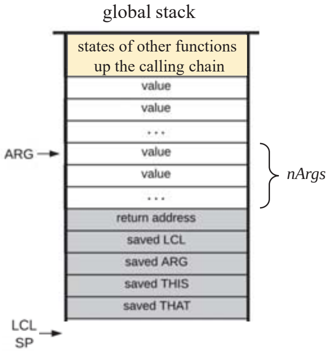
Handling call

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
```



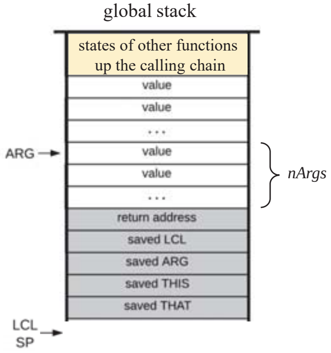
Handling call

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
```



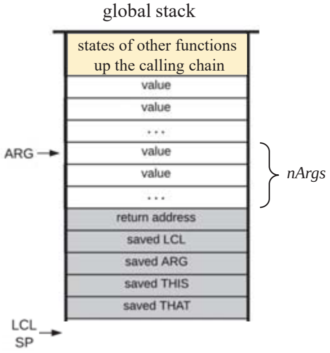
Handling call

VM command: call *functionName nArgs*

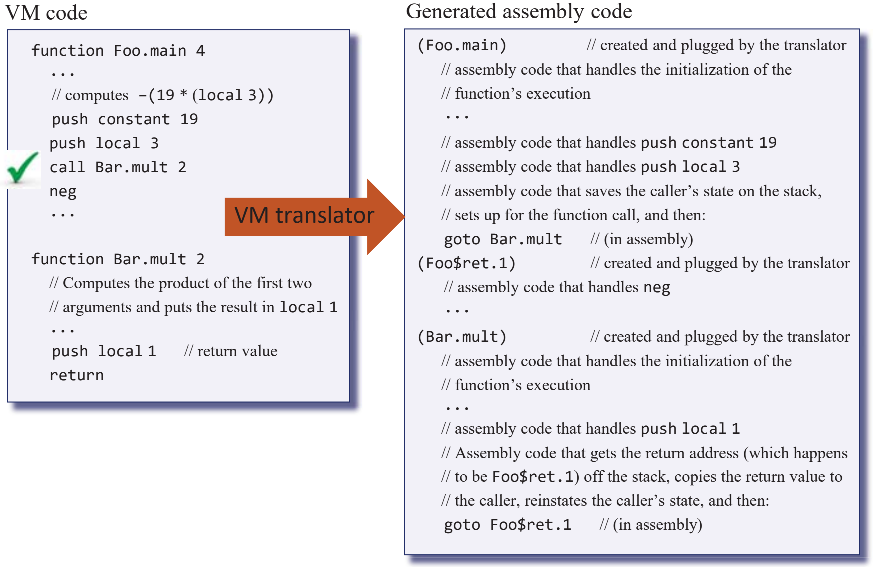
(calls a function, informing that *nArgs* arguments have been pushed onto the stack)

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

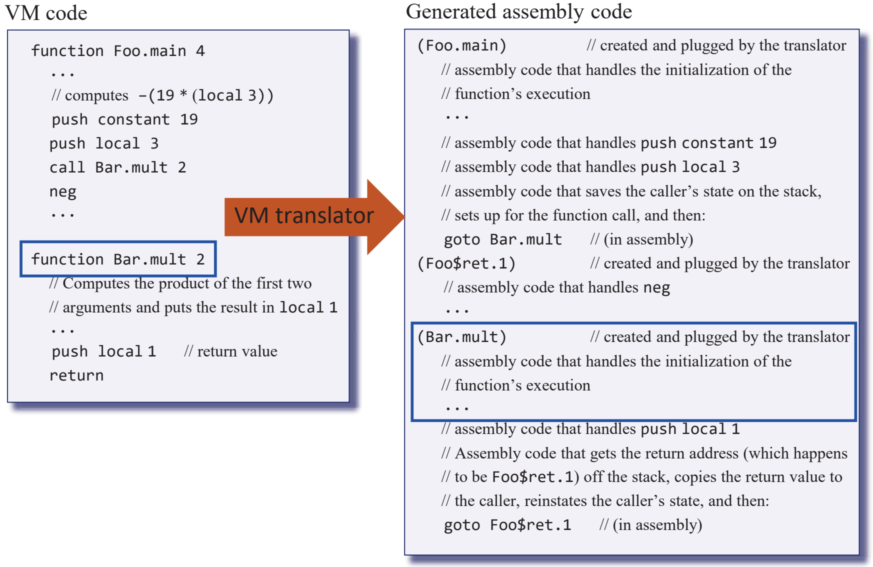
```
function Foo.main 4
   ...
   // computes  -(19 * (local 3))
   push constant 19
   push local 3
✓  call Bar.mult 2
   neg
   ...

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
```

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
```

VM translator

Generated assembly code

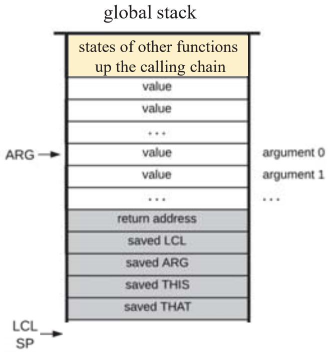
```
(Foo.main)             // created and plugged by the translator
   // assembly code that handles the initialization of the
   // function's execution
   ...
   // assembly code that handles push constant 19
   // assembly code that handles push local 3
   // assembly code that saves the caller's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



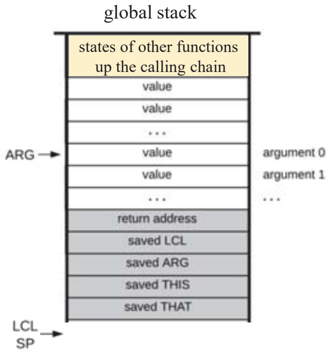
Handling `function`

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
```



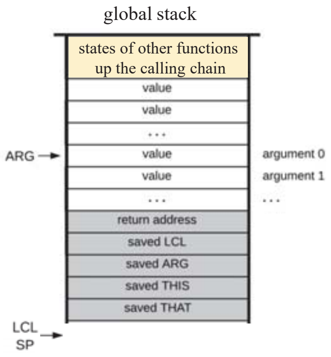
Handling `function`

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
```



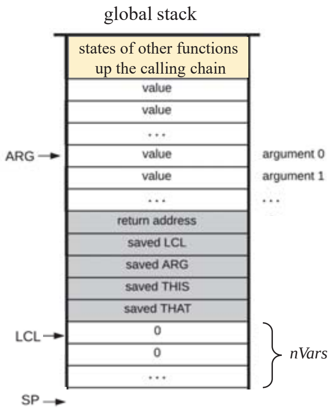
Handling `function`

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
```



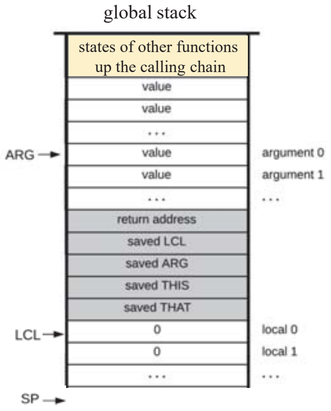
Handling `function`

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 `return`

VM code

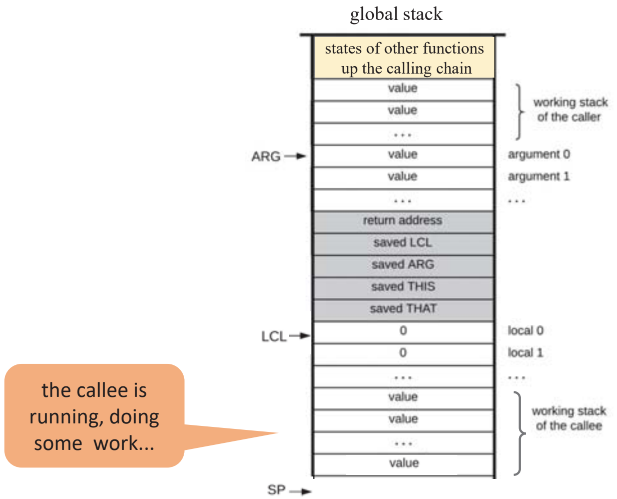
```
function Foo.main 4
  ...
  // computes -(19 * (local 3))
  push constant 19
  push local 3
  call Bar.mult 2
  neg
  ...

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

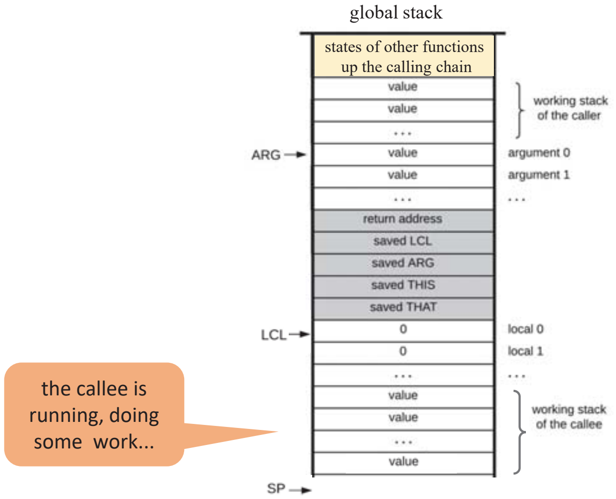
```
(Foo.main)          // created and plugged by the translator
  // assembly code that handles the initialization of the
  // function's execution
  ...
  // assembly code that handles push constant 19
  // assembly code that handles push local 3
  // assembly code that saves the caller's state on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



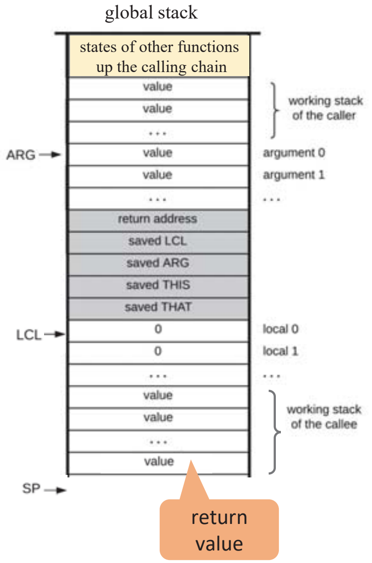
Handling `return`

VM command: `return`



Handling `return`

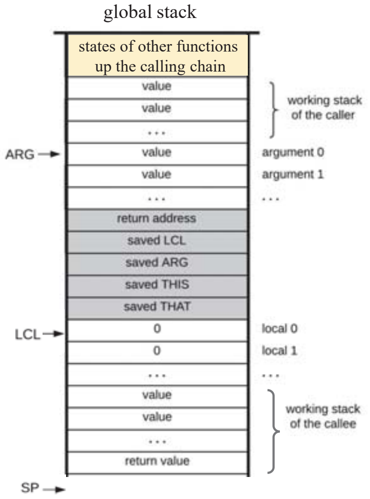
VM command: `return`



Handling `return`

VM command: `return`

Assembly code (generated by the translator):

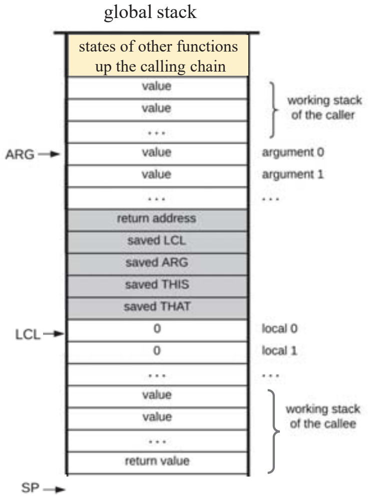

global stack
states of other functions up the calling chain
value
value
. . .
ARG → value — argument 0
value — argument 1
. . .
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0 — local 0
0 — local 1
. . .
value
value
. . .
return value
SP →
working stack of the caller
working stack of the callee

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
```

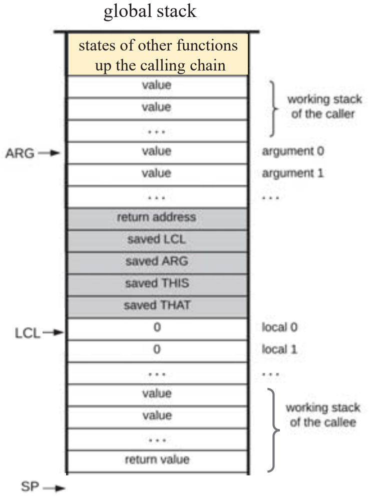

global stack
states of other functions up the calling chain
value
value
. . .
ARG → value — argument 0
value — argument 1
. . .
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0 — local 0
0 — local 1
. . .
value
value
. . .
return value
SP →
working stack of the caller
working stack of the callee

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
```

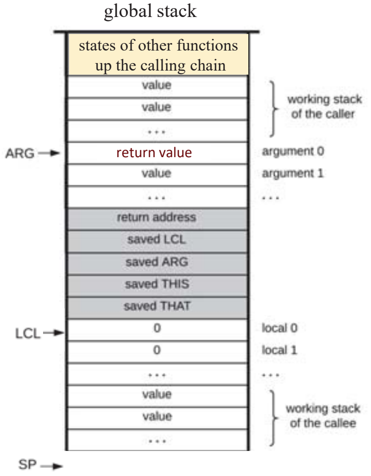

global stack
states of other functions up the calling chain
value
value
. . .
ARG → value — argument 0
value — argument 1
. . .
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0 — local 0
0 — local 1
. . .
value
value
. . .
return value
SP →
working stack of the caller
working stack of the callee

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
```

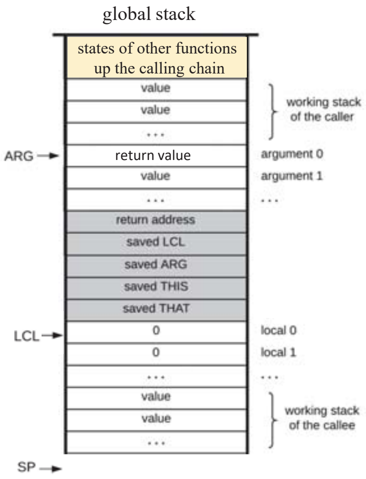

global stack
states of other functions up the calling chain
value
value
. . .
ARG → return value — argument 0
value — argument 1
. . .
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0 — local 0
0 — local 1
. . .
value
value
. . .
SP →
working stack of the caller
working stack of the callee

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
```



global stack
states of other functions up the calling chain
value
value
...
ARG → return value
value
...
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0
0
...
value
value
...
SP →

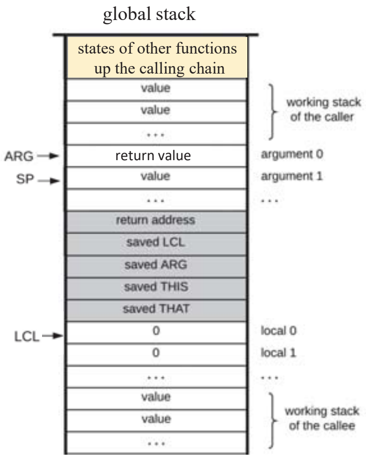
working stack of the caller
argument 0
argument 1
...
local 0
local 1
...
working stack of the callee

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
```



global stack
states of other functions up the calling chain
value
value
...
ARG → return value
SP → value
...
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0
0
...
value
value
...

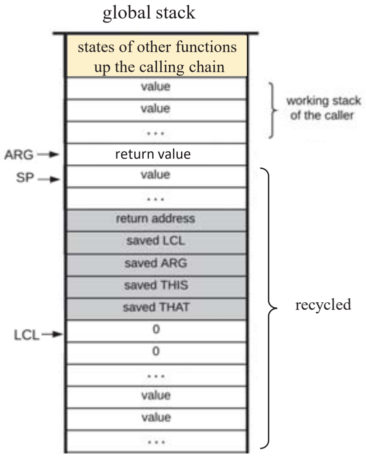
working stack of the caller
argument 0
argument 1
...
local 0
local 1
...
working stack of the callee

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
```



global stack
states of other functions up the calling chain
value
value
...
ARG → return value
SP → value
...
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0
0
...
value
value
...

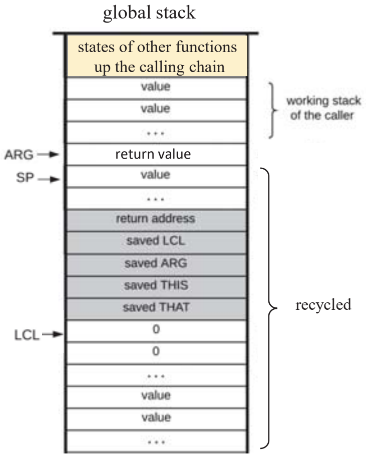
working stack of the caller
recycled

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
```



global stack
states of other functions up the calling chain
value
value
...
ARG → return value
SP → value
...
return address
saved LCL
saved ARG
saved THIS
saved THAT
LCL → 0
0
...
value
value
...

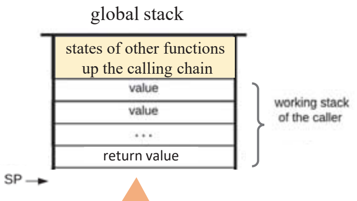
working stack of the caller
recycled

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
```


global stack
states of other functions up the calling chain
value
value
...
return value
working stack of the caller
SP

Net impact: the caller is back in business, with the return value at the top of the stack

Handling `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
```

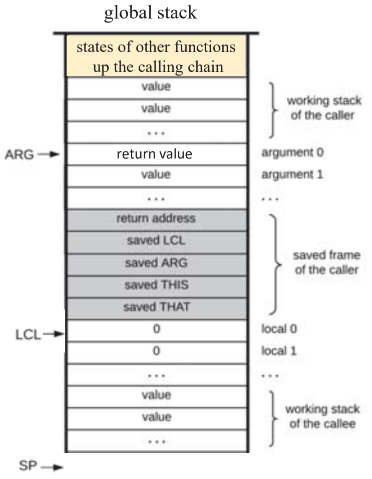
VM translator

Generated assembly code

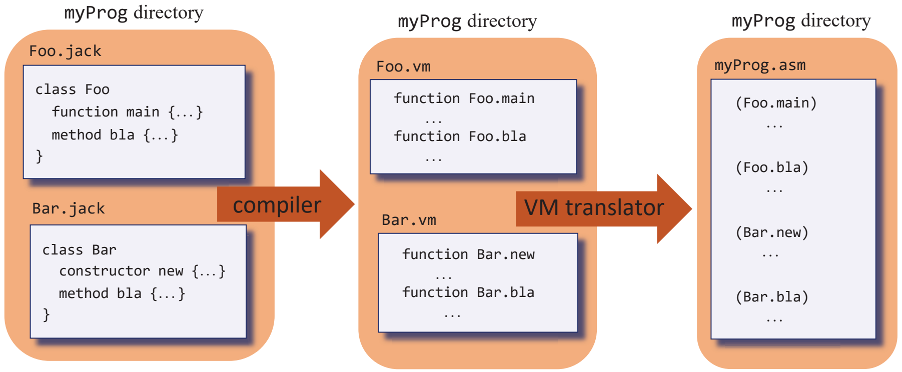
```
(Foo.main)
   // assembly code that handles the setting up of
   // a function's execution
   ...
   // assembly code that handles push constant 19
   // assembly code that handles push local 3
   // assembly code that saves the caller's state,
   // handles some pointers, and then:
   goto Bar.mult    // (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.



The big picture: program compilation and translation



```
class Foo
  function main {...}
  method bla {...}
}
```

```
class Bar
  constructor new {...}
  method bla {...}
}
```

```
function Foo.main
  ...
function Foo.bla
  ...
```

```
function Bar.new
  ...
function Bar.bla
  ...
```

```
(Foo.main)
  ...

(Foo.bla)
  ...

(Bar.new)
  ...

(Bar.bla)
  ...
```

- 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