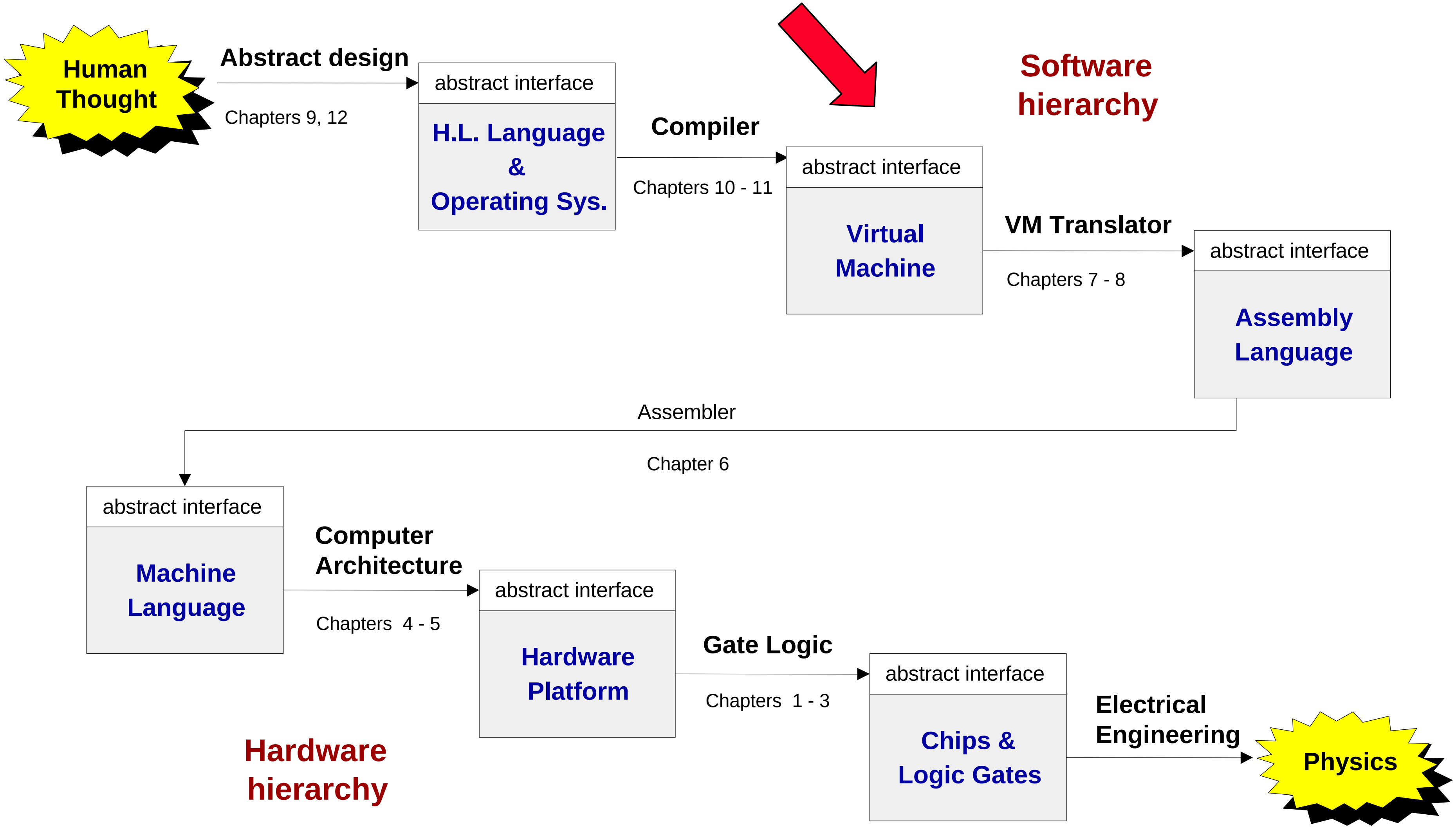
Chapter 8:

Virtual Machine II: Program Control

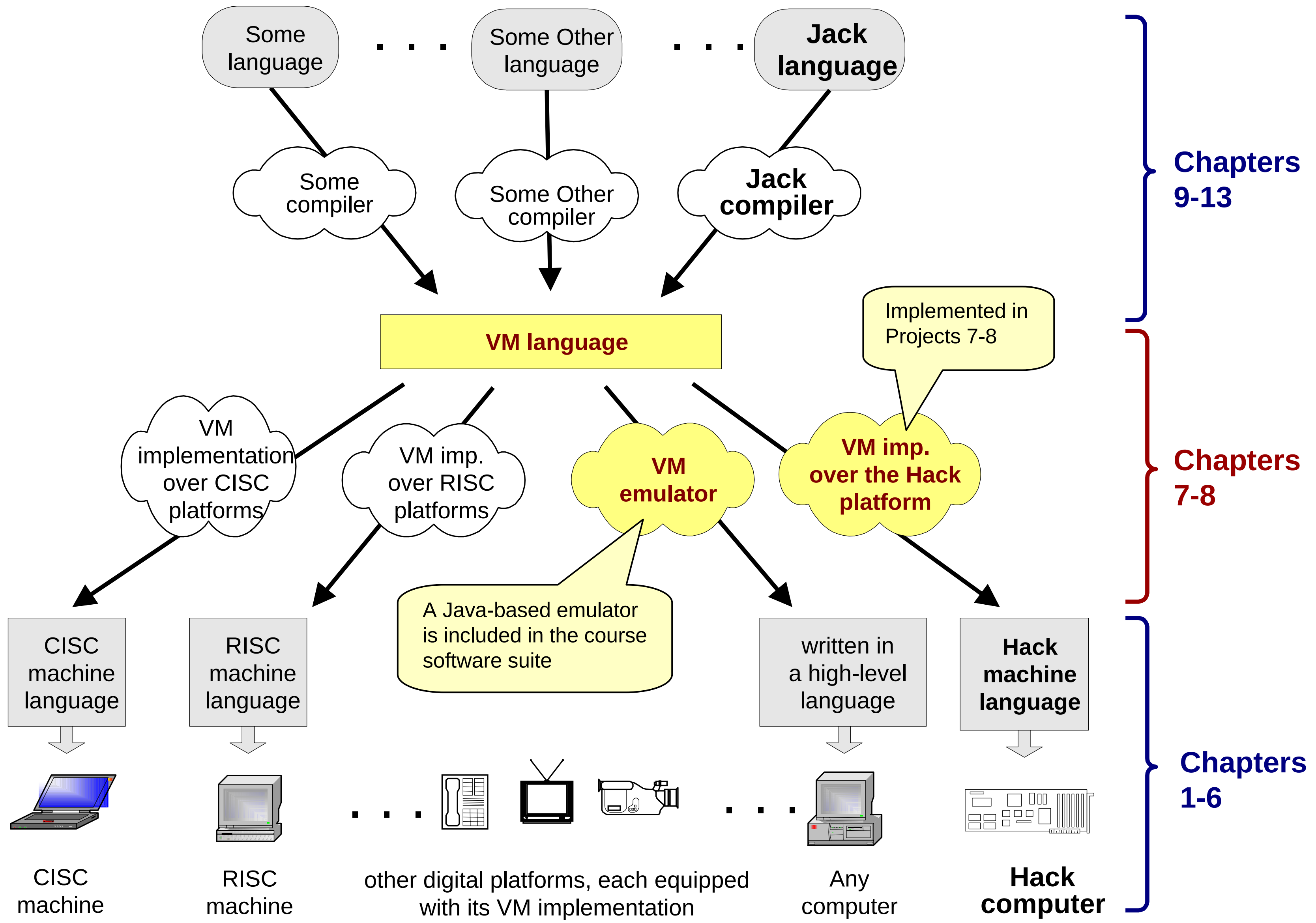
Virtual Machine

Part II: Program Control

Where we are at:



The big picture



Lecture plan

<u>Goal</u>: Specify and implement a VM model and language

Arithmetic / Boolean commands

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

Previous lecture

Memory access commands

`pop segment i`
`push segment i`

Program flow commands

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

This lecture

Function calling commands

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

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

Program structure and translation path (on the Hack-Jack platform)

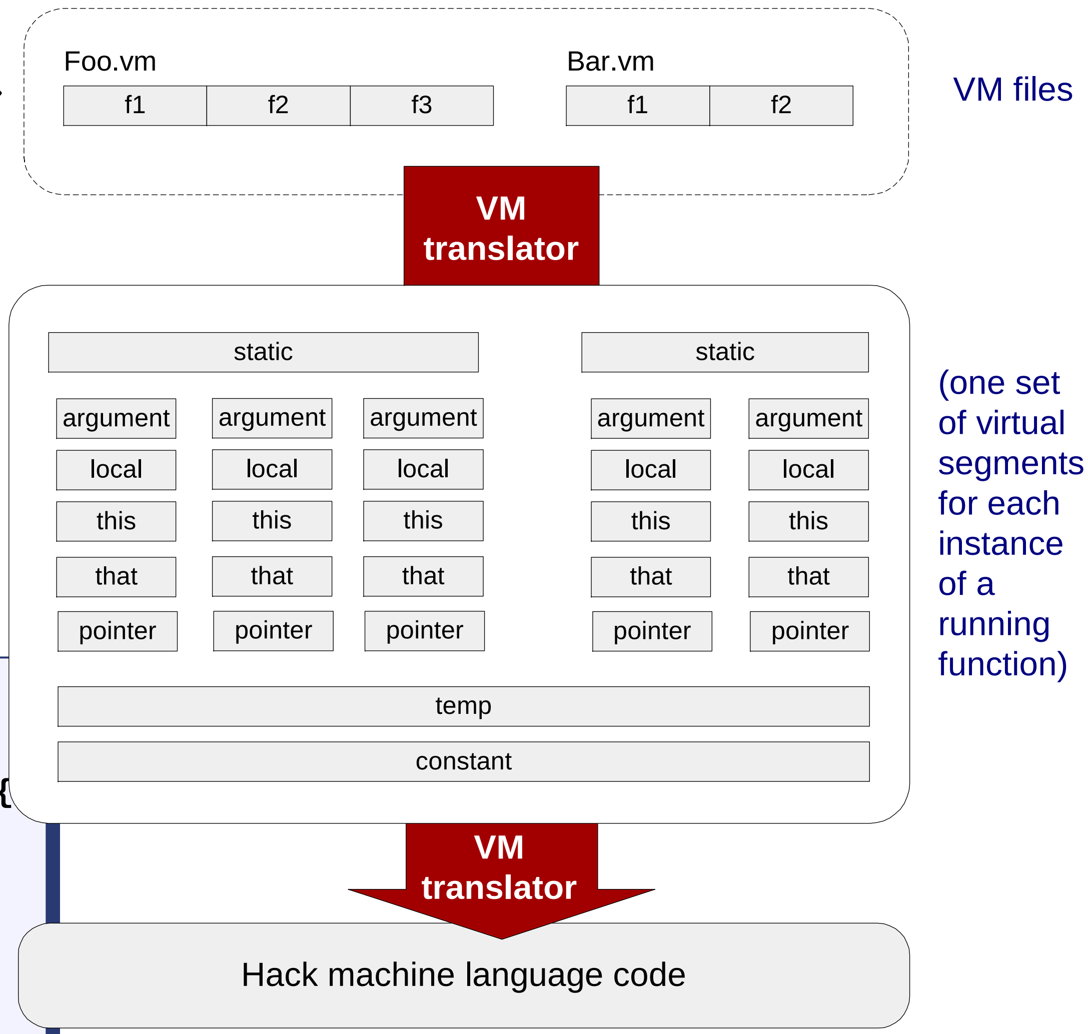
Jack source code:(example):

```
class Foo {
  static int x1, x2, x3;
  method int f1(int x) {
    var int a, b;
    ...
  }
  method void f2(int x, int y) {
    var int a, b, c;
    ...
  }
  function int f3(int u) {
    var int x;
    ...
  }
}
```

```
class Bar {
  static int y1, y2;
  function void f1(int u, int v) {
    ...
  }
  method void f2(int x) {
    var int a1, a2;
    ...
  }
}
```

Compiler

Following compilation:



The challenge ahead

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

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

In order to enable such high-level code we have to know how to handle:

- Arithmetic operations (previous lecture)
- Boolean operations (previous lecture)
- Program flow (this lecture, *easy*)
- Subroutines (this lecture, *medium/rare*)

In the Jack/Hack platform: all these abstractions are delivered by the VM level.

Program flow

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

  (pop the topmost element from the stack. If it's not zero, jump)

<u>Implementation</u> (by translation to assembly):

Simple. label declarations and goto directives can be effected directly by assembly commands.

Example:

```
function mult 2
  push    constant 0
  pop     local 0
  push    argument 1
  pop     local 1
label     loop
  push    local 1
  push    constant 0
  eq
  if-goto end
  push    local 0
  push    argument 0
  add
  pop     local 0
  push    local 1
  push    constant 1
  sub
  pop     local 1
  goto    loop
label     end
  push    local 0
  return
```

Subroutines

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

Subroutines = a major programming artifact

- The primitive (given) language can be extended at will by user-defined commands (AKA *subroutines* / *functions* / *methods* ...)
- The primitive commands and the user-defined commands have the same look-and-feel
- Perhaps the most important abstraction delivered by programming languages. The challenge: to make the implementation of this abstraction as transparent as possible:

"A well-deigned system consists of a collection of black box modules, each executing its effect like magic"
(Steven Pinker, *How The Mind Works*)

Subroutines usage at the VM level (pseudo code)

```
// x+2
push x
push 2
add
...
```

```
// x^3
push x
push 3
call power
...
```

```
// (x^3+2)^y
push x
push 3
call power
push 2
add
push y
call power
...
```

```
// Power function
// result = first arg
// raised to the power
// of the second arg.
function power
// code omitted
push result
return
```

<u>Call-and-return convention</u>

- The caller pushes the arguments, calls the callee, then waits for it to return
- Before the callee terminates (returns), it must push a return value
- At the point of return, the callee's resources are recycled, and the caller's state is re-instated
- **Caller's net effect:** the arguments were replaced by the return value (just like with primitive operations)

<u>Behind the scene</u>

- Recycling and re-instating subroutine resources and states is a major headache
- The VM implementation should manage it "like magic"
- The magic is stack-based, and is considered a great CS gem.

Subroutine commands

- **`function`** ***g nVars***

  (Here starts a function called ***g***, which has ***nVars*** local variables)

- **`call`** ***g nArgs***

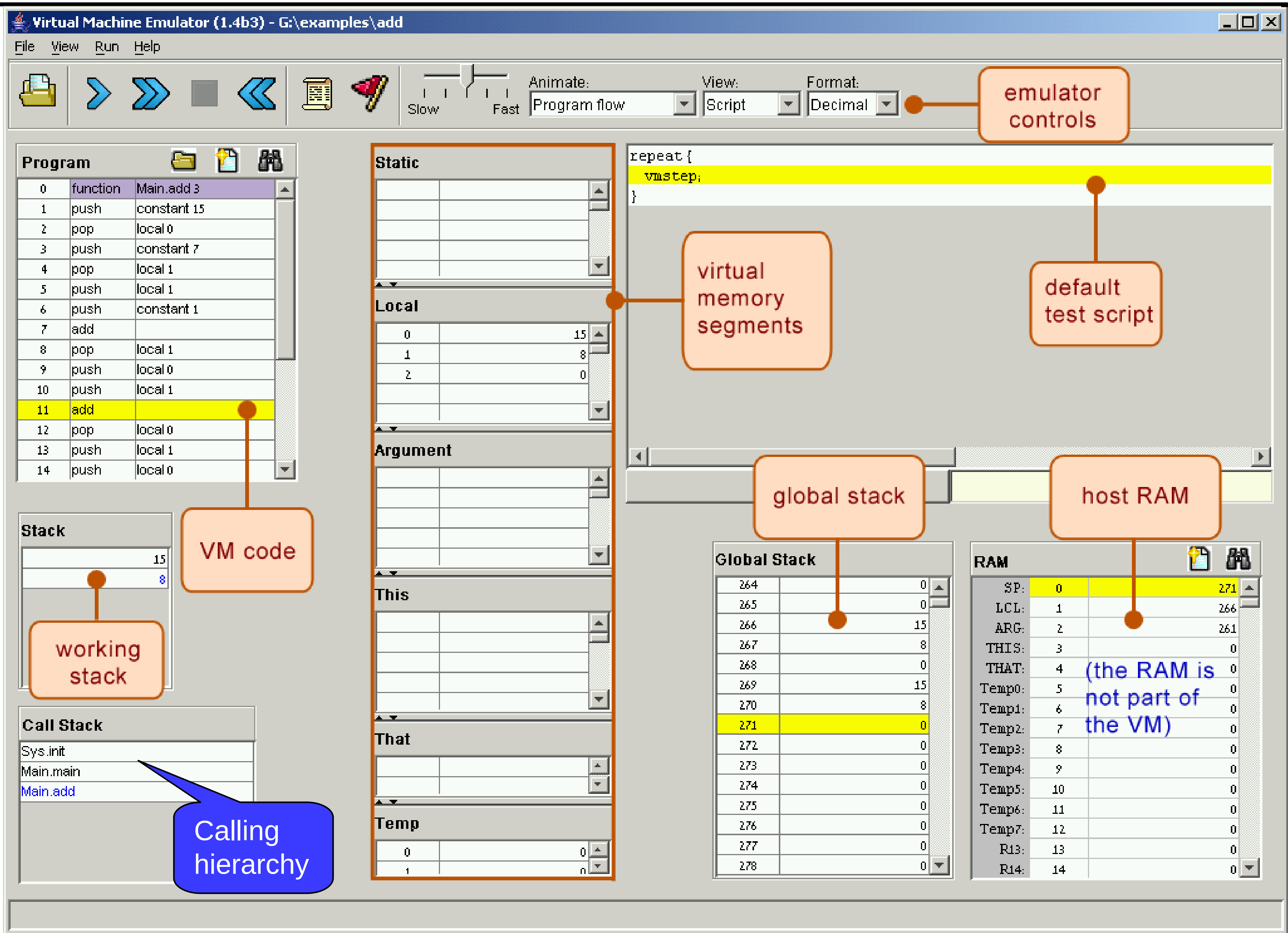
  (Invoke function ***g*** for its effect; ***nArgs*** arguments have been pushed onto the stack)

- **`Return`**

  (Terminate execution and return control to the calling function)

Implementation: Next few slides.

Aside: The VM emulator (Java-based, included in the course software suite)



The calling protocol

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

The caller's view:

- Before calling the function, I must push as many arguments as necessary onto the stack
- Next, I invoke the function using the **`call`** command
- After the called function returns:
  - The arguments 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
  - All my memory segments (**argument**, **local**, **static**, …) are the same as before the call.

Blue = function writer's responsibility

Black = black box magic, supplied by the VM implementation.

The callee's view:

- When I start executing, my **argument** segment has been initialized with actual argument values passed by the caller
- My **local** variables segment has been allocated and initialized to zero
- The **static** segment that I see has been set to the **static** segment of the VM file to which I belong, and the working stack that I see is empty
- Before returning, I must push a value onto the stack.

VM implementation view of the calling protocol

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

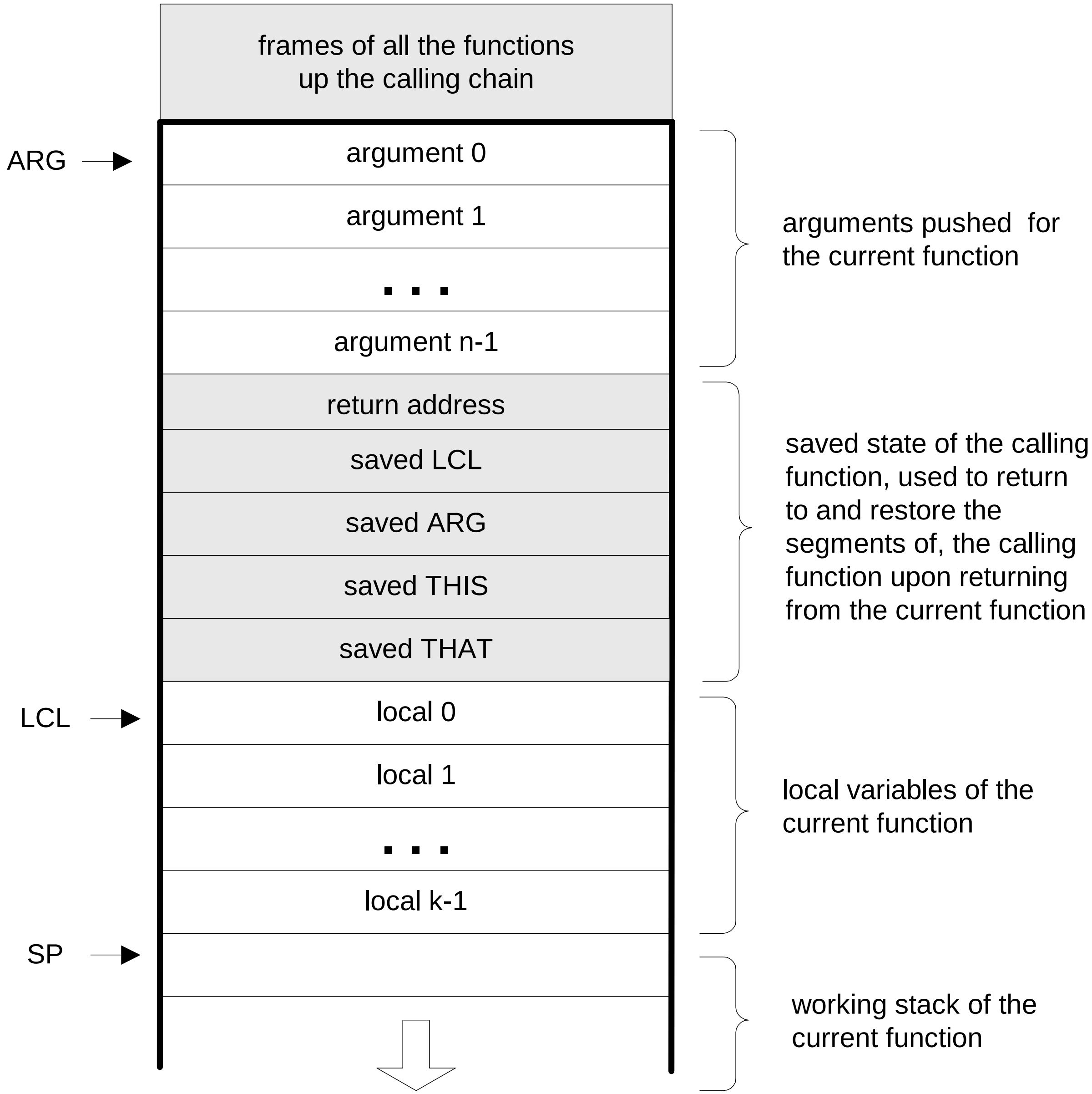
When function *f* calls function *g*, I must:

- Save the return address
- Save the segment pointers of *f*
- Allocate, and initialize to 0, as many local variables as needed by *g*
- Set the local and argument segment pointers of *g*
- Transfer control to *g*.

When g terminates and control should return to f, I must:

- Clear the arguments and other junk from the stack
- Restore the segments of *f*
- Transfer control back to *f*
  (jump to the saved return address).

The VM implementation housekeeping storage = the stack



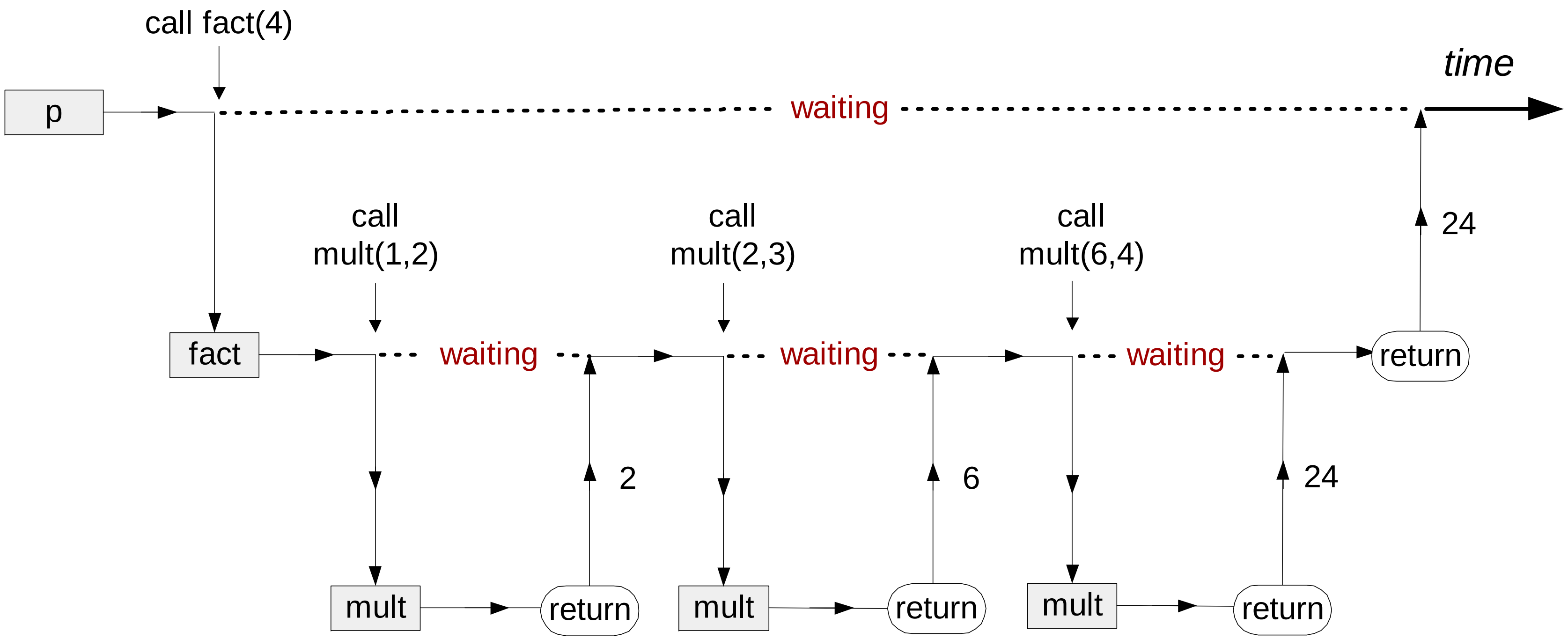
- Remember: at any typical point of time, some functions are waiting, and only the current function is running
- Shaded areas: irrelevant to the current function
- The current function sees only the top of the stack (AKA *working stack*)
- The rest of the stack holds the frozen states of all the functions up the calling hierarchy
- Physical storage details depend on the VM implementation.

Example: a typical calling scenario

```
function p(...) {
...
   ... fact(4) ...
}
```

```
function fact(n) {
    vars result,j;
    result=1; j=1;
    while j<=n {
      result=mult(result,j);
    }
     return result;
}
```

```
function mult(x,y) {
    vars sum,j;
    sum=0; j=y;
    while j>0 {
      sum=sum+x;
    }
    return sum;
}
```

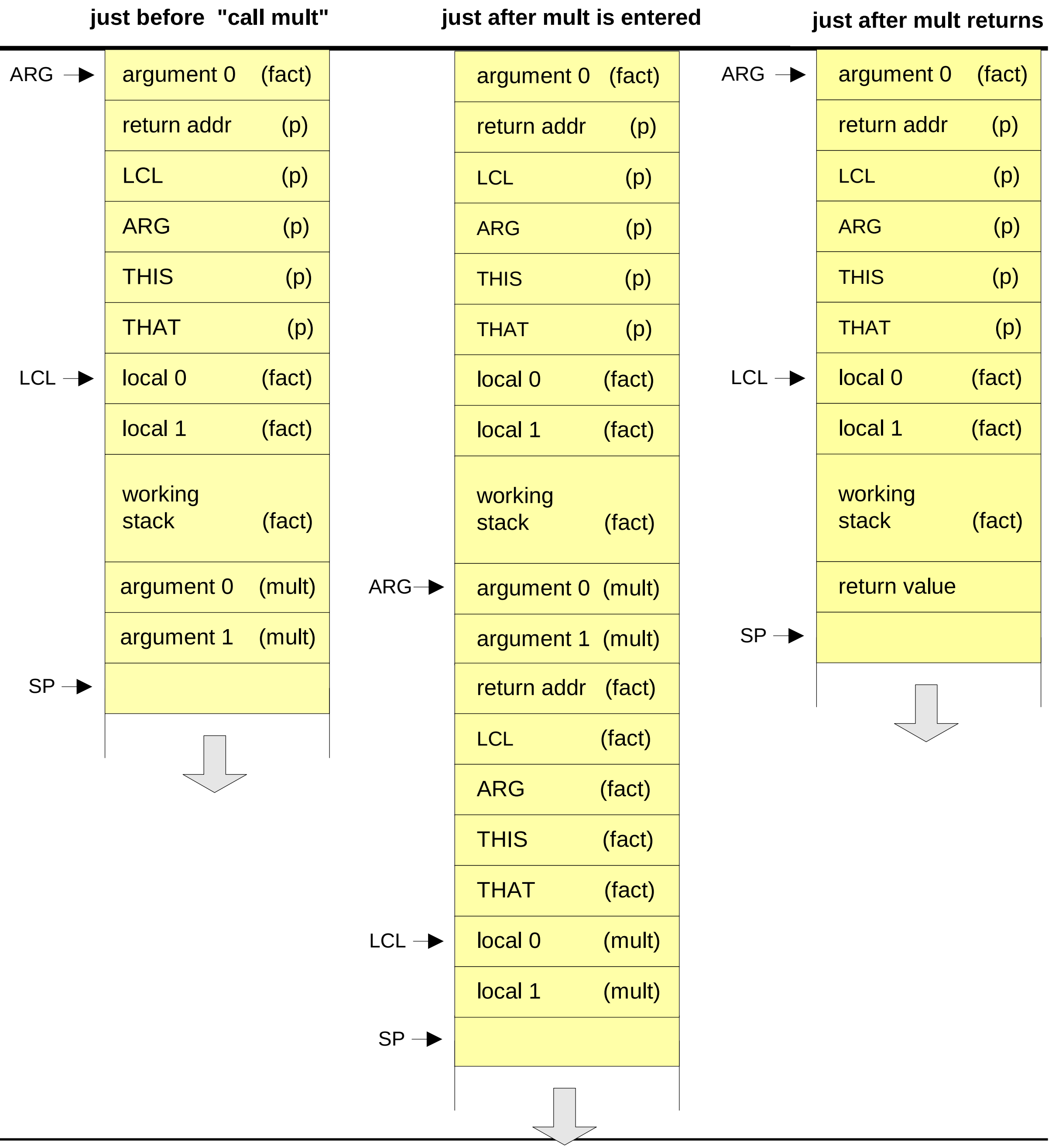


Behind the scene:

```
function p(...) {
...
   ... fact(4) ...
}
```

```
function fact(n) {
    vars result,j;
    result=1; j=1;
    while j<=n {
      result=mult(result,j);
    }
     return result;
}
```

```
function mult(x,y) {
    vars sum,j;
    sum=0; j=y;
    while j>0 {
      sum=sum+x;
    }
    return sum;
}
```

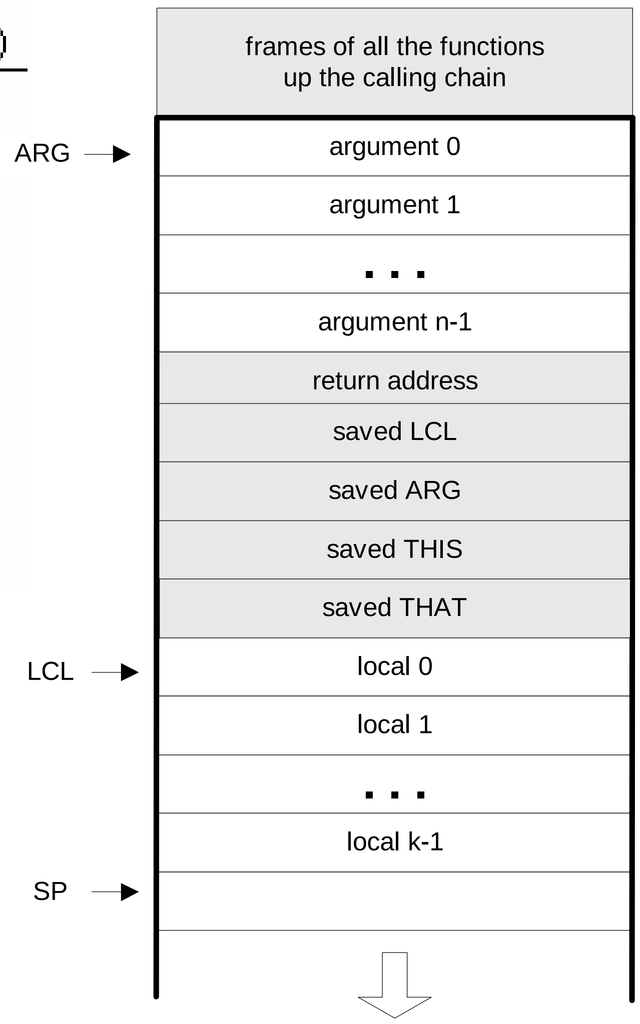


Implementing the `call f n` command

`call f n`

(calling a function `f` after `n` arguments have been pushed onto the stack)

```
    push return-address // (Using the label declared below)
    push LCL            // Save LCL of the calling function
    push ARG            // Save ARG of the calling function
    push THIS           // Save THIS  of the calling function
    push THAT           // Save THAT of the calling function
    ARG = SP-n-5        // Reposition ARG (n = number of args)
    LCL = SP            // Reposition LCL
    goto f              // Transfer control
(return-address)        // Declare a label for the return-address
```

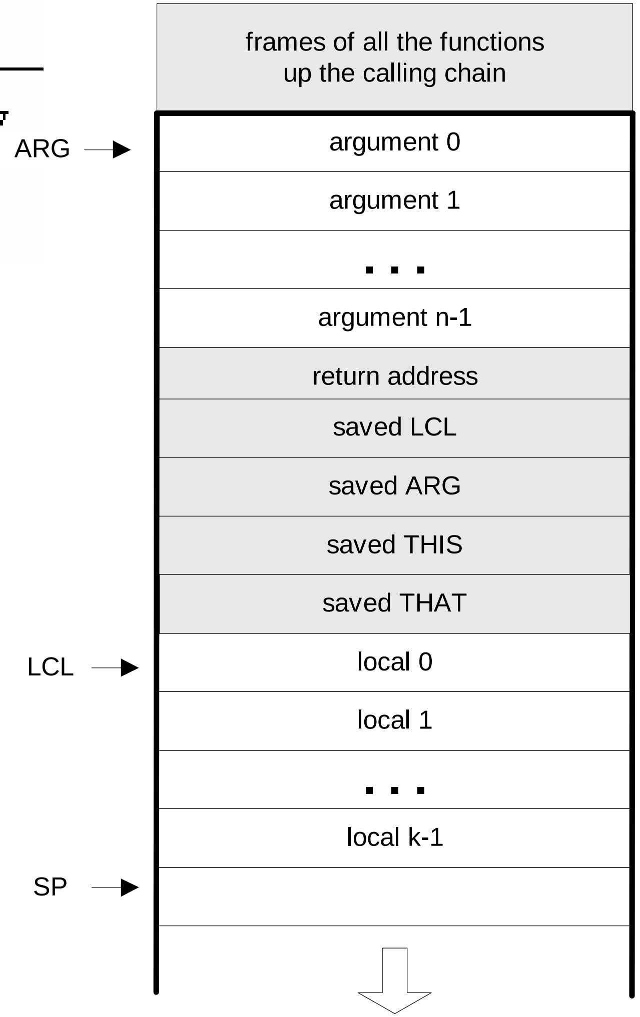


Implementing the **function f k** command

function f k

(declaring a function f that has k local variables)

```
(f)                      // Declare a label for the function entry
   repeat k times:       // k = number of local variables
   PUSH 0                // Initialize all of them to 0
```

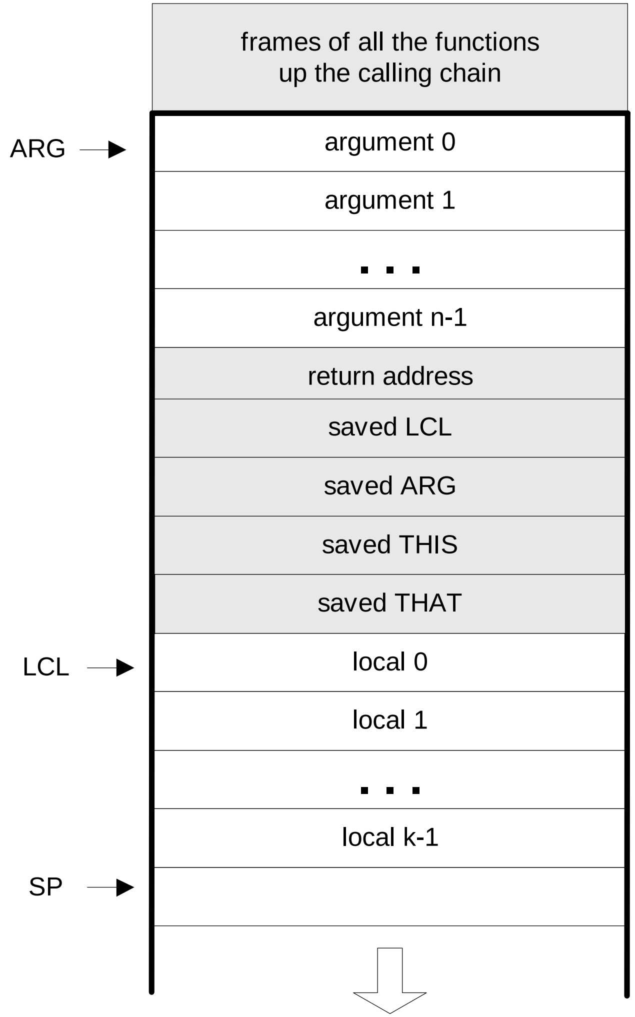


Implementing the `return` command

`return`

(from a function)

```
FRAME=LCL          // FRAME is a temporary variable
RET=*(FRAME-5)     // Put the return-address in a temp. variable
*ARG=pop()         // Reposition the return value for the caller
SP=ARG+1           // Restore SP of the caller
THAT=*(FRAME-1)    // Restore THAT of the caller
THIS=*(FRAME-2)    // Restore THIS of the caller
ARG=*(FRAME-3)     // Restore ARG  of the caller
LCL=*(FRAME-4)     // Restore LCL of the caller
goto RET           // Goto return-address (in the caller's code)
```



One more detail: bootstrapping

- A high-level jack program (AKA *application*) is a set of class files. By convention, one class must be called **Main**, and this class must have at least one function called **main**. The contract is such that when we tell the computer to execute the program, the function **Main.main** starts running

<u>Implementation:</u>

- After the program is compiled, each class file is translated into a **.vm** file
- From the host platform's standpoint, the operating system is also a set of **.vm** files (AKA "libraries") that co-exist alongside the user's **.vm** files
- One of the OS libraries is called **Sys.vm**, which includes a function called **init.** This function starts with some OS initialization code (explained in Ch. 12), then it does **call** *f* and enters an infinite loop; if the code originates from Jack, *f* is **Main.main**
- Thus, to bootstrap, the VM implementation has to effect (e.g. in assembly), the following operations:

```
SP = 256          // initialize the stack pointer to 0x0100
call Sys.init     // the initialization function
```

VM implementation over the Hack platform

- Extends the VM implementation proposed in the last lecture (Chapter 7)
- The result: a big assembly program with lots of agreed-upon symbols:

Symbol	*Usage*
`SP`, `LCL`, `ARG`, `THIS`, `THAT`	These predefined symbols point, respectively, to the stack top and to the base addresses of the virtual segments `local`, `argument`, `this`, and `that`.
`R13 - R15`	These predefined symbols can be used for any purpose.
`Xxx.j`	Each static variable `j` in a VM file `Xxx.vm` is translated into the assembly symbol `Xxx.j`. In the subsequent assembly process, these symbolic variables will be allocated RAM space by the Hack assembler.
`functionName$label`	Each `label b` command in a VM function `f` should generate a globally unique symbol "`f$b`" where "`f`" is the function name and "`b`" is the label symbol within the VM function's code. When translating `goto b` and `if-goto b` VM commands into the target language, the full label specification "`f$b`" must be used instead of "`b`".
`(FunctionName)`	Each VM function f should generates a symbol "`f`" that refers to its entry point in the instruction memory of the target computer.
return-address	Each VM function call should generate and insert into the translated code a unique symbol that serves as a return address, namely the memory location (in the target platform's memory) of the command following the function call.

Proposed API

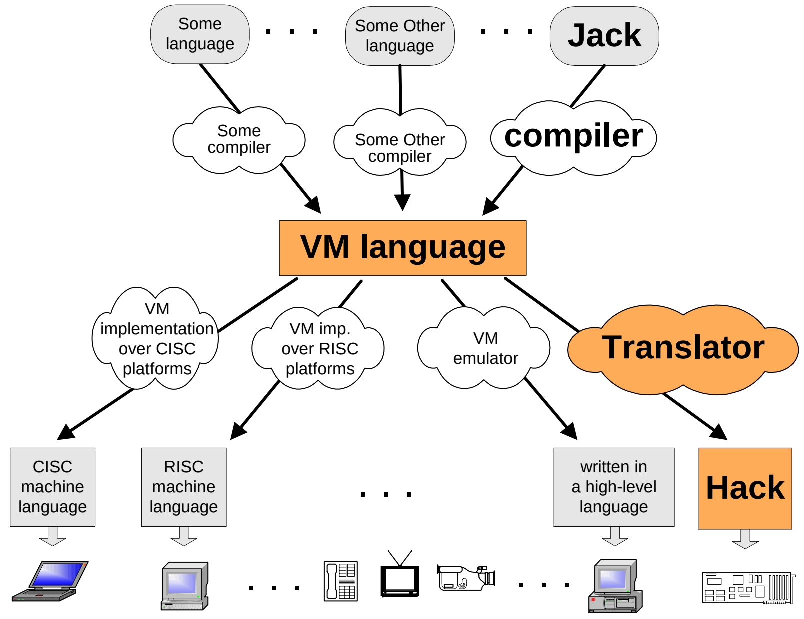
CodeWriter: Translates VM commands into Hack assembly code. The routines listed here should be added to the `CodeWriter` module API given in chapter 7.

Routine	Arguments	Returns	Function
`writeInit`	--	--	Writes the assembly code that effects the VM initialization, also called *bootstrap code*. This code must be placed at the beginning of the output file.
`writeLabel`	`label` (string)	--	Writes the assembly code that is the translation of the `label` command.
`writeGoto`	`label` (string)	--	Writes the assembly code that is the translation of the `goto` command.
`writeIf`	`label` (string)	--	Writes the assembly code that is the translation of the `if-goto` command.
`writeCall`	`functionName` (string) `numArgs` (int)	--	Writes the assembly code that is the translation of the `call` command.
`writeReturn`	--	--	Writes the assembly code that is the translation of the `return` command.
`writeFunction`	`functionName` (string) `numLocals` (int)	--	Writes the assembly code that is the trans. of the given `function` command.

Perspective

Benefits of the VM approach

- Code transportability: compiling for different platforms require replacing only the VM implementation
- Language inter-operability: code of multiple languages can be shared using the same VM
- Common software libraries
- Code mobility: Internet
- Modularity:
  - Improvements in the VM implementation are shared by all compilers above it
  - Every new digital device with a VM implementation gains immediate access to an existing software base
  - New programming languages can be implemented easily using simple compilers



Benefits of managed code:

- Security
- Array bounds, index checking, …
- Add-on code
- Etc.

VM Cons

- Performance.