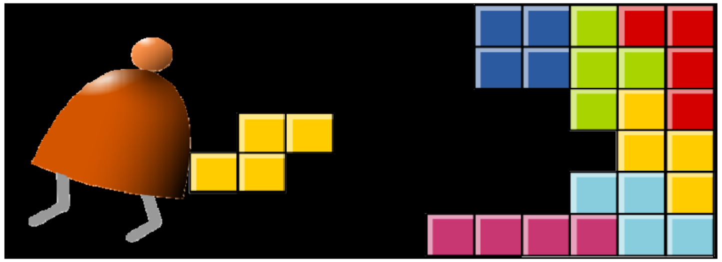
Virtual Machine

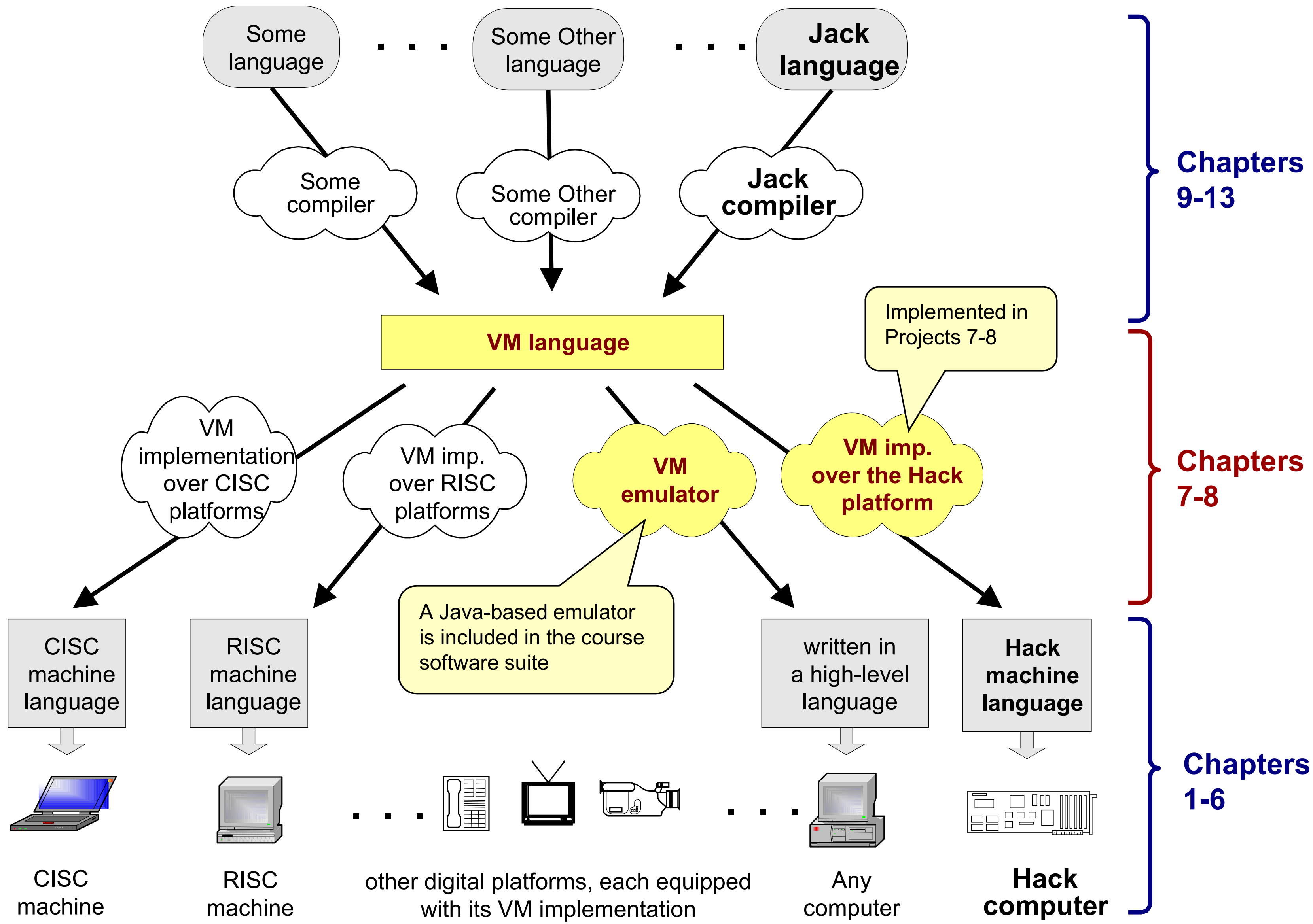
Part II: Program Control



Building a Modern Computer From First Principles

www.nand2tetris.org

The big picture



The VM language

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

Arithmetic / Boolean commands	Program flow commands
`add`	`label` (declaration)
`sub`	`goto` (label)
`neg`	`if-goto` (label)
`eq`	Chapter 8
`gt` Chapter 7	
`lt`	**Function calling commands**
`and`	`function` (declaration)
`or`	`call` (a function)
`not`	`return` (from a function)
Memory access commands	
`pop x` (pop into x, which is a variable)	
`push y` (y being a variable or a constant)	

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.

Compiler →

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
```

Compiler →

VM (pseudo) code

```
   push a
   push 0
   eq
   if-goto elseLabel
   push b
   neg
   push b
   push b
   call mult
   push 4
   push a
   call mult
   push c
   call mult
   call sqrt
   add
   push 2
   push a
   call mult
   div
   pop x
   goto contLable
elseLabel:
   push c
   neg
   push b
   call div
   pop x
contLable:
```

VM translator →

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) — **(this lecture)**
- Boolean expressions — (previous lecture)
- function call and return logic — **(this lecture)**
- arithmetic expressions — (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`

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)

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
```

<u>How to translate these abstractions into assembly?</u>

- Simple: label declarations and goto directives can be effected directly by assembly commands
- More to the point: given any one of these three VM commands, the VM Translator must emit one or more assembly commands that effects the same semantics on the Hack platform
- How to do it? see project 8.

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
   ...
```

Lecture plan

<u>Arithmetic / Boolean commands</u>

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

previous lecture

<u>Memory access commands</u>

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

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

<u>Program flow commands</u>

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

<u>Function calling commands</u>

`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 at will by user-defined commands (aka *subroutines* / *functions* / *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
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:

- 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 removes all arguments and pushes the result to the stack
- The space of the callee is recycled
- The caller's state is reinstalled
- Jump back to where is called

Stack as the facility for subroutines

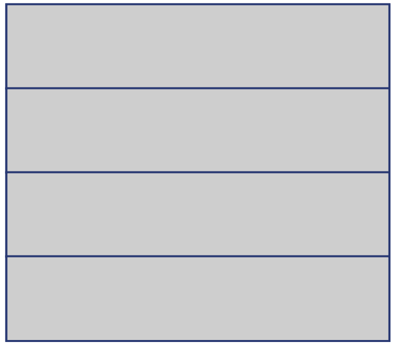
code

```
function a
   call b
   call c
   ...

function b
   call c
   call d
   ...

function c
   call d
   ...

function d
   ...
```

flow

```
start a
   start b
      start c
         start d
         end d
      end c
      start d
      end d
   end b
   start c
      start d
      end d
   end c
end a
```

stack



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 (later).

Function call-and-return conventions

Calling function

```
function demo 3
  ...
  push constant 7
  push constant 2
  add
  push constant 3
  call mult
  ...
```

called function aka "callee" (example)

```
function mult 1
  push constant 0
  pop local 0 // result (local 0) = 0
label loop
  ...          // rest of code omitted
label end
  push local 0  // push result
  return
```

Although not obvious in this example, every VM function has a private set of 5 memory segments (`local`, `argument`, `this`, `that`, `pointer`)

These resources exist as long as the function is running.

Function call-and-return conventions

Calling function

```
function demo 3
   ...
   push constant 7
   push constant 2
   add
   push constant 3
   call mult
   ...
```

called function aka "callee" (example)

```
function mult 1
   push constant 0
   pop local 0 // result (local 0) = 0
label loop
   ...          // rest of code omitted
label end
   push local 0  // push result
   return
```

Call-and-return programming convention

- The caller must push the necessary argument(s), call the callee, and wait 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, the caller's state is re-instated, execution continues from the command just after the `call`
- **Caller's net effect:** the arguments were replaced by the return value (just like with primitive commands)

Function call-and-return conventions

Calling function

```
function demo 3
  ...
  push constant 7
  push constant 2
  add
  push constant 3
  call mult
  ...
```

called function aka "callee" (example)

```
function mult 1
  push constant 0
  pop local 0 // result (local 0) = 0
label loop
  ...          // rest of code omitted
label end
  push local 0  // push result
  return
```

Behind the scene

- Recycling and re-instating subroutine resources and states is a major headache
- Some agent (either the VM or the compiler) should manage it behind the scene "like magic"
- In our implementation, the magic is VM / stack-based, and is considered a great CS gem.

The function-call-and-return protocol

```
function g nVars
call g nArgs
return
```

The caller's view:

- Before calling a function *g*, I must push onto the stack as many arguments as needed by *g*
- Next, I invoke the function using the command `call g nArgs`
- After *g* 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 (`local, argument, this, that, pointer`) are the same as before the call.

Blue = VM function writer's responsibility

Black = black box magic, delivered by the VM implementation

Thus, the VM implementation writer must worry about the "black operations" only.

The function-call-and-return protocol

```
function g nVars
call g nArgs
return
```

The callee's (*g* '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 exiting, I must push a value onto the stack and then use the command `return`.

Blue = VM function writer's responsibility

Black = black box magic, delivered by the VM implementation

Thus, the VM implementation writer must worry about the "black operations" only.

The function-call-and-return protocol: the VM implementation view

```
function g nVars
call g nArgs
return
```

When function *f* calls function *g*, the VM implementation must:

- Save the return address within *f* 's code:
  the address of the command just after the `call`
- Save the virtual segments 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*.

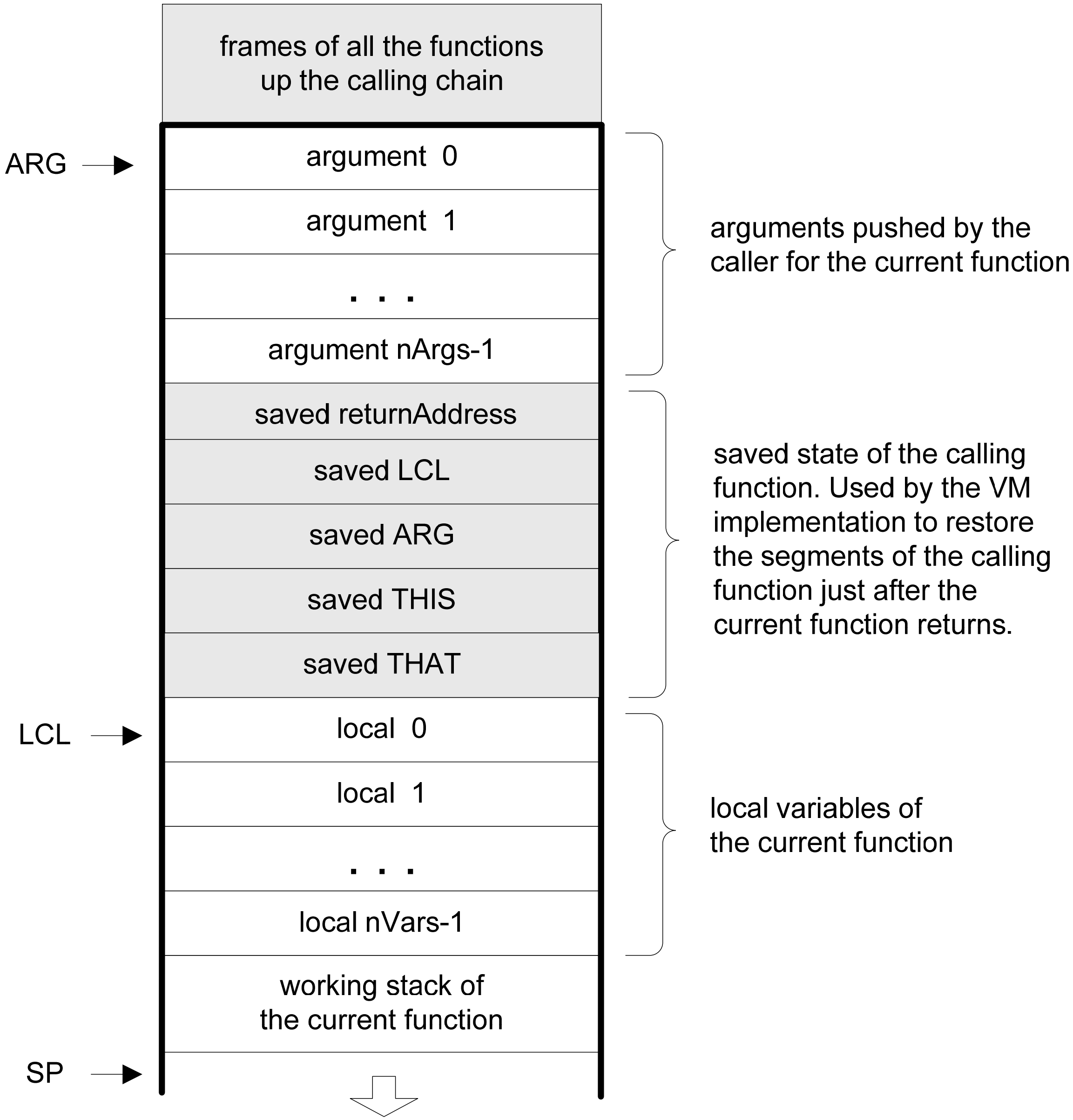
When *g* terminates and control should return to *f*, the VM implementation must:

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

Q: How should we make all this work "like magic"?

A: We'll use the stack cleverly.

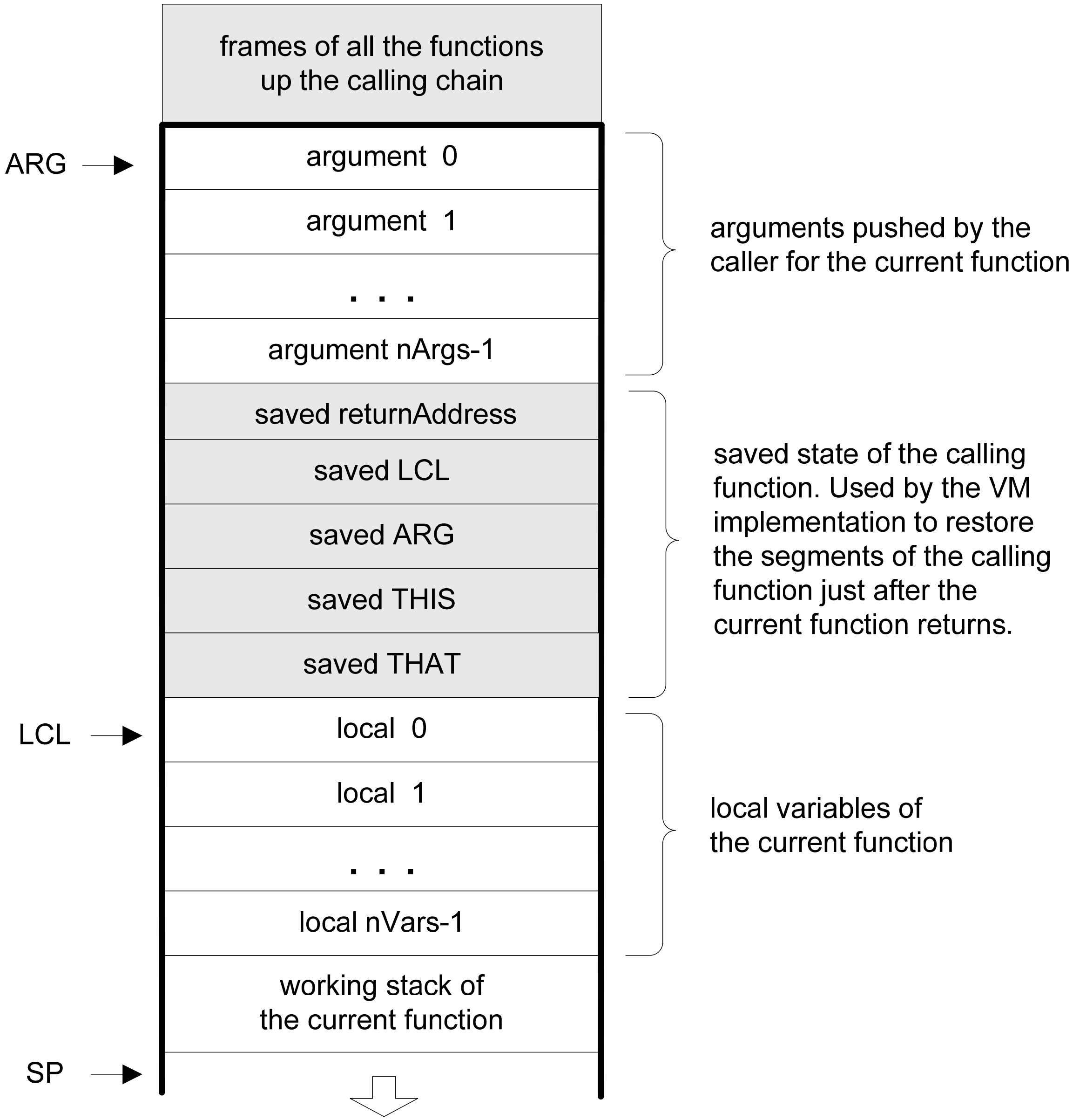
The implementation of the VM's stack on the host Hack RAM



Global stack:
the entire RAM area dedicated for holding the stack

Working stack:
The stack that the current function sees

The implementation of the VM's stack on the host Hack RAM

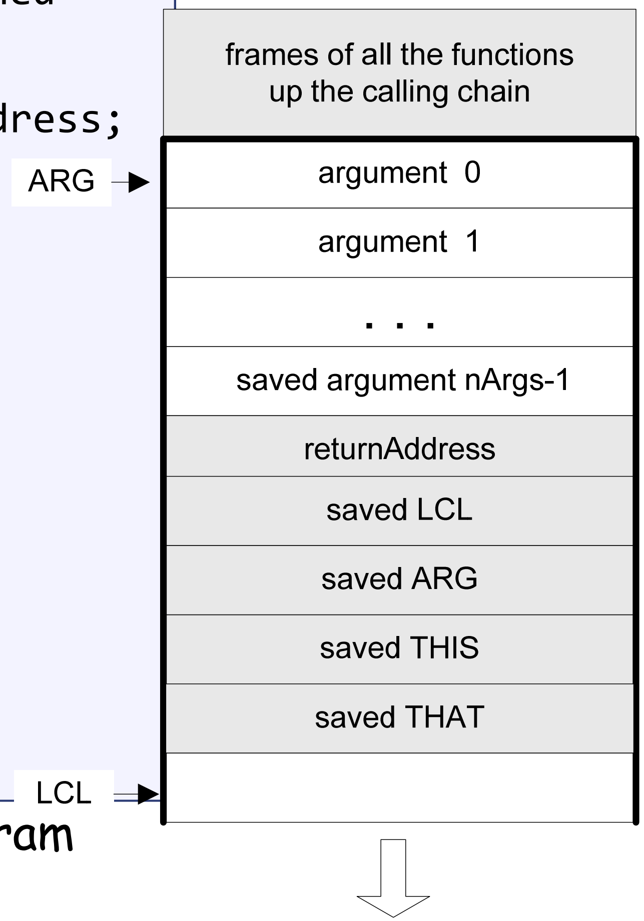


- At any point of time, only one function (the *current function*) is executing; other functions may be waiting up the calling chain
- Shaded areas: irrelevant to the current function
- The current function sees only the working stack, and has access only to its memory segments
- The rest of the stack holds the frozen states of all the functions up the calling hierarchy.

Implementing the `call g nArgs` command

`call g nArgs`

```
  // In the course of implementing the code of f
  // (the caller), we arrive to the command call g nArgs.
  // we assume that nArgs arguments have been pushed
  // onto the stack. What do we do next?
  // We generate a symbol, let's call it returnAddress;
  // Next, we effect the following logic:
  push returnAddress // saves the return address
  push LCL           // saves the LCL of f
  push ARG           // saves the ARG of f
  push THIS          // saves the THIS of f
  push THAT          // saves the THAT of f
  ARG = SP-nArgs-5   // repositions SP for g
  LCL = SP           // repositions LCL for g
  goto g             // transfers control to g
returnAddress:       // the generated symbol
```

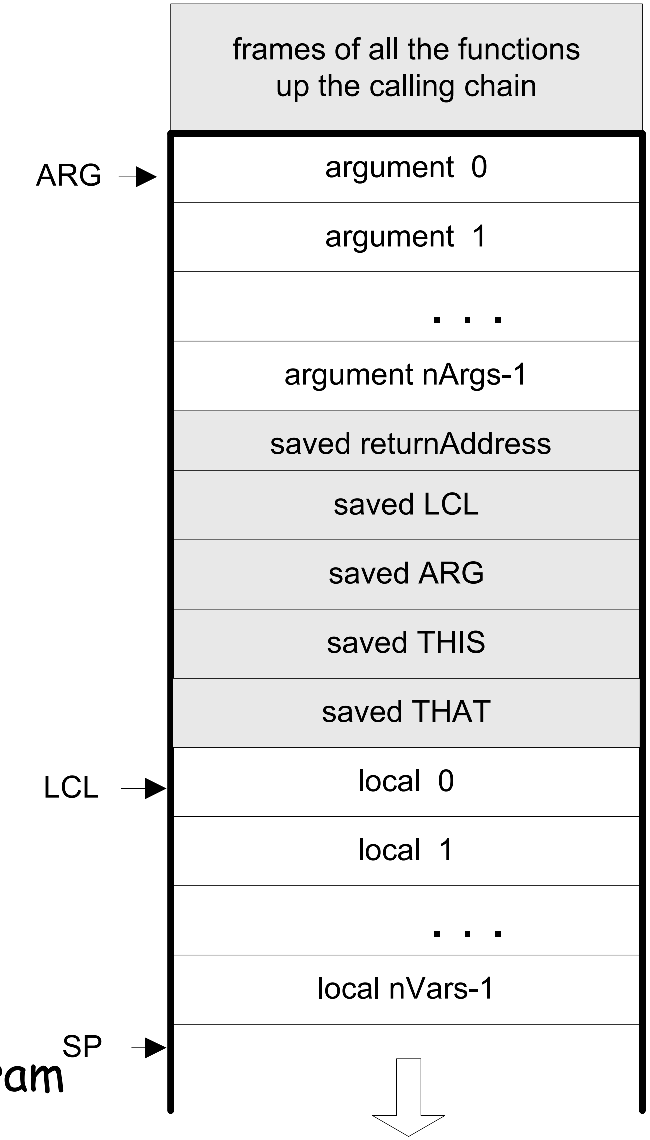


<u>Implementation:</u> If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Implementing the `function g nVars` command

```
function g nVars
```

```
// to implement the command function g nVars,
// we effect the following logic:

g:
  repeat nVars times:
  push 0
```

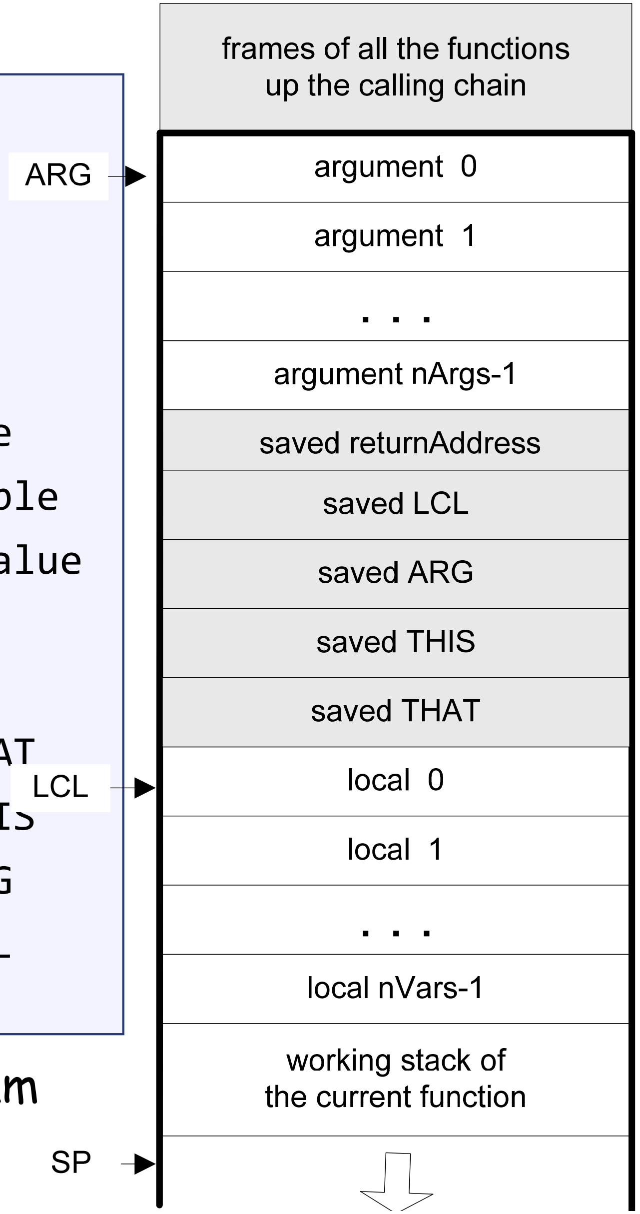


<u>Implementation:</u> If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Implementing the `return` command

return

```
// In the course of implementing the code of g,
// we arrive to the command return.
// We assume that a return value has been pushed
// onto the stack.
// We effect the following logic:
   frame = LCL          // frame is a temp. variable
   retAddr = *(frame-5) // retAddr is a temp. variable
   *ARG = pop           // repositions the return value
                        // for the caller
   SP=ARG+1             // restores the caller's SP
   THAT = *(frame-1)    // restores the caller's THAT
   THIS = *(frame-2)    // restores the caller's THIS
   ARG = *(frame-3)     // restores the caller's ARG
   LCL = *(frame-4)     // restores the caller's LCL
   goto retAddr         // goto returnAddress
```



Implementation: If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Example: factorial

High-level code

```
function fact (n) {
  int result, j;
  result = 1;
  j = 1;
  while ((j=j+1) <= n) {
     result = result * j;
  }
  return result;
}
```

Pseudo code

```
...
loop:
  if ((j=j+1) > n) goto end
  result=result*j
  goto loop
end:
...
```

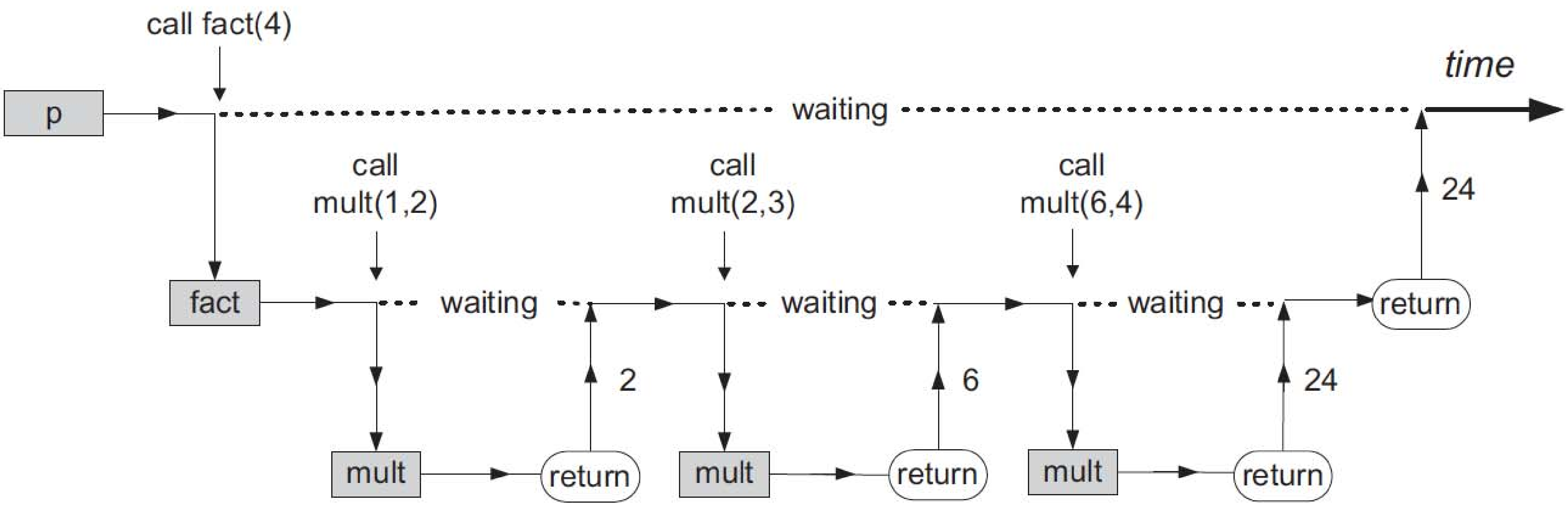
VM code (first approx.)

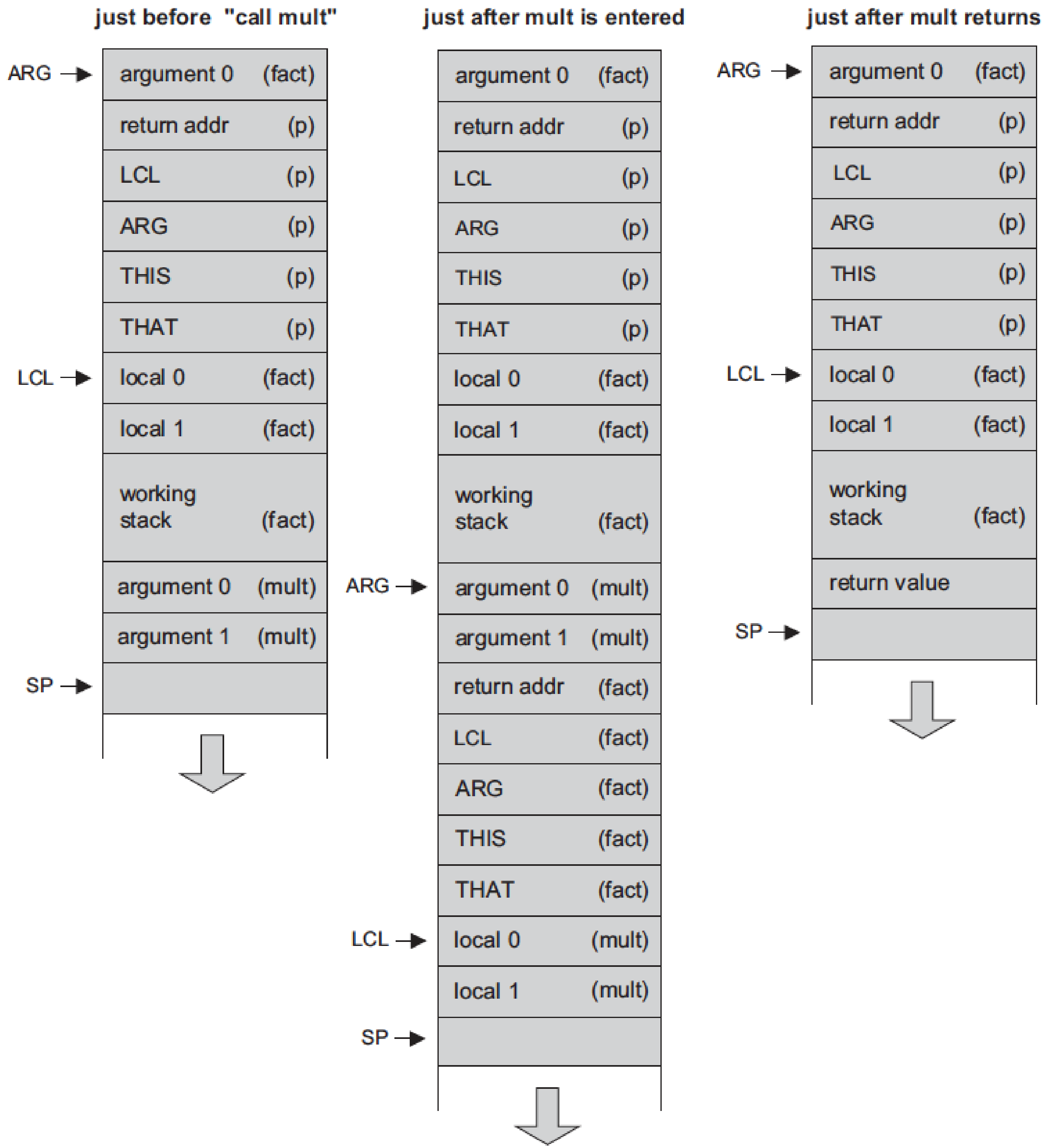
```
function fact(n)
    push 0
    pop result
    push 1
    pop j
label loop
    push 1
    push j
    add
    pop j
    push n
    gt
    if-goto end
    push result
    push j
    mult
    pop result
    goto loop
label end
    push result
    return
```

VM code

```
function fact 2
    push constant 0
    pop local 0
    push constant 1
    pop local 1
label loop
    push constant 1
    push local 1
    add
    pop local 1
    push argument 0
    gt
    if-goto end
    push local 0
    push local 1
    call mult 2
    pop local 0
    goto loop
label end
    push local 0
    return
```

```
function p
  ...
  push constant 4
  call fact 1
  ...
```




just before "call mult"
just after mult is entered
just after mult returns
ARG
LCL
SP
argument 0 (fact)
return addr (p)
LCL (p)
ARG (p)
THIS (p)
THAT (p)
local 0 (fact)
local 1 (fact)
working stack (fact)
argument 0 (mult)
argument 1 (mult)
return addr (fact)
LCL (fact)
ARG (fact)
THIS (fact)
THAT (fact)
local 0 (mult)
local 1 (mult)
return value

Example: factorial

High-level code

```
function fact (n) {
  int r;
  if (n!=1)
    r = n * fact(n-1);
  else
    r = 1;
  return r;
}
```

VM code (first approx.)

```
function fact(n)
   push n
   push 1
   eq
   if-goto else
   push n
   push 1
   sub
   fact
   push n
   mult
   ~~pop r~~
   goto cont
label else
   push 1
   ~~pop r~~
label cont
   ~~push r~~
   return
```

Example: factorial

High-level code

```
function fact (n) {
  int r;
  if (n!=1)
    r = n * fact(n-1);
  else
    r = 1;
  return r;
}
```

VM code (first approx.)

```
function fact(n)
   push n
   push 1
   eq
   if-goto else
   push n
   push 1
   sub
   fact
   push n
   mult

   goto cont
label else
   push 1

label cont

   return
```

VM code

```
function fact 1
   push argument 0
   push constant 1
   eq
   if-goto else
   push argument 0
   push constant 1
   sub
   call fact 1
   push argument 0
   call mult 2

   goto cont
label else
   push constant 1

label cont

   return
```

Calling stack for fact(4)

High-level code

```
function fact (n) {
  int r;
  if (n!=1)
    r = n * fact(n-1);
  else
    r = 1;
  return r;
}
```

stack

stack
frame fact(4)
frame fact(3)
frame fact(2)
frame fact(1)

Calling stack for fact(4)

High-level code

```
function fact (n) {
  int r;
  if (n!=1)
    r = n * fact(n-1);
  else
    r = 1;
  return r;
}
```

stack

frame fact(4)
frame fact(3)
frame fact(2)
frame mult(2,1)

Calling stack for fact(4)

High-level code

```
function fact (n) {
  int r;
  if (n!=1)
    r = n * fact(n-1);
  else
    r = 1;
  return r;
}
```

stack

frame fact(4)
frame fact(3)
frame mult(3,2)

Calling stack for fact(4)

High-level code

```
function fact (n) {
  int r;
  if (n!=1)
    r = n * fact(n-1);
  else
    r = 1;
  return r;
}
```

stack

frame fact(4)
frame mult(4,6)

Bootstrapping

A high-level jack *program* (aka *application*) is a set of class files.
By a Jack convention, one class must be called `Main`, and this class must have at least one function, called `main`.

The contract: when we tell the computer to execute a Jack program, the function `Main.main` starts running

Implementation:

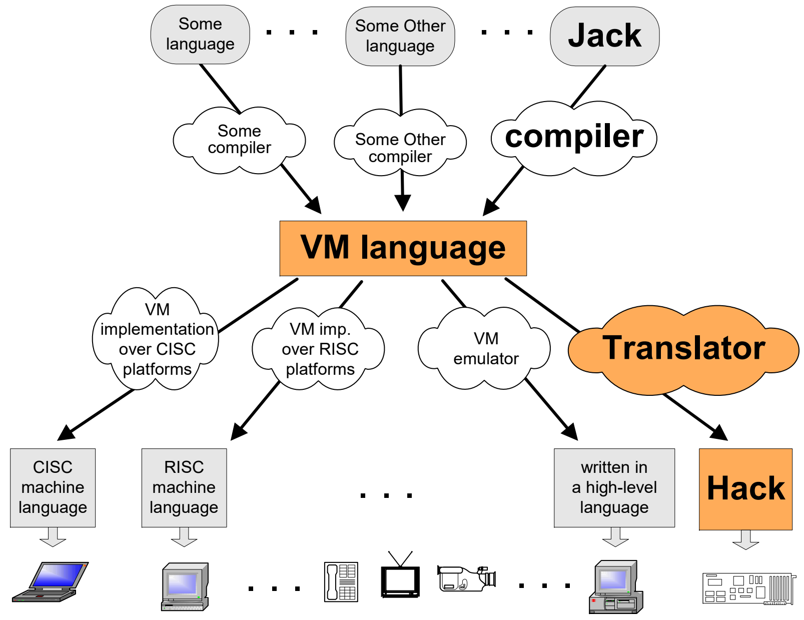
- After the program is compiled, each class file is translated into a `.vm` file
- The operating system is also implemented as a set of `.vm` files (aka "libraries") that co-exist alongside the program's `.vm` files
- One of the OS libraries, called `Sys.vm`, includes a method called `init`. The `Sys.init` function starts with some OS initialization code (we'll deal with this later, when we discuss the OS), then it does `call 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    // call the function that calls Main.main
```

Perspective

Benefits of the VM approach

- Code transportability: compiling for different platforms requires 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, cloud



Benefits of managed code:

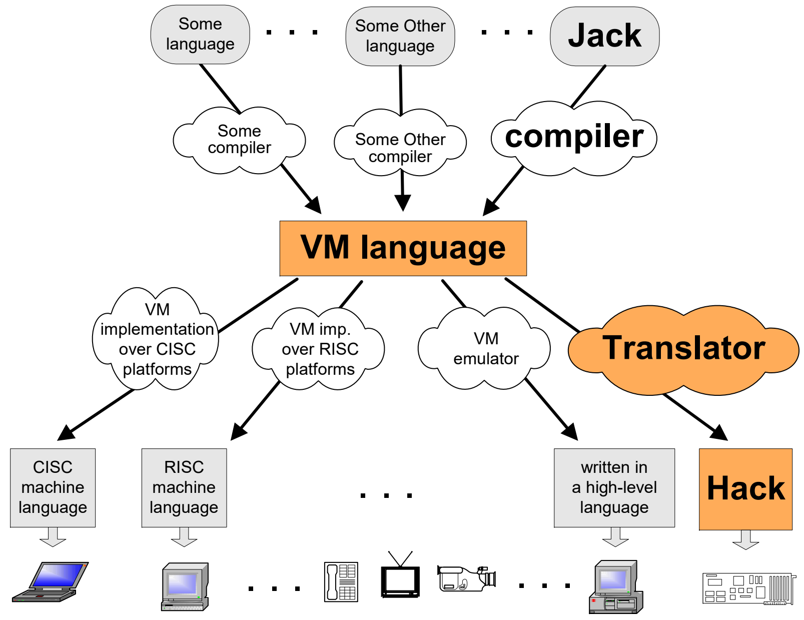
- Security
- Array bounds, index checking, …
- Add-on code
- Etc.

VM Cons

- Performance.

Perspective

- Some virtues of the modularity implied by the VM approach to program translation:
  - 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.