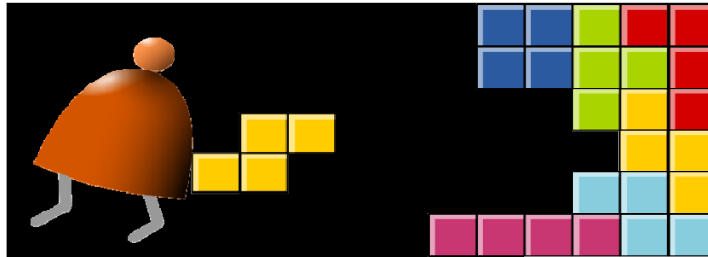


Virtual Machine

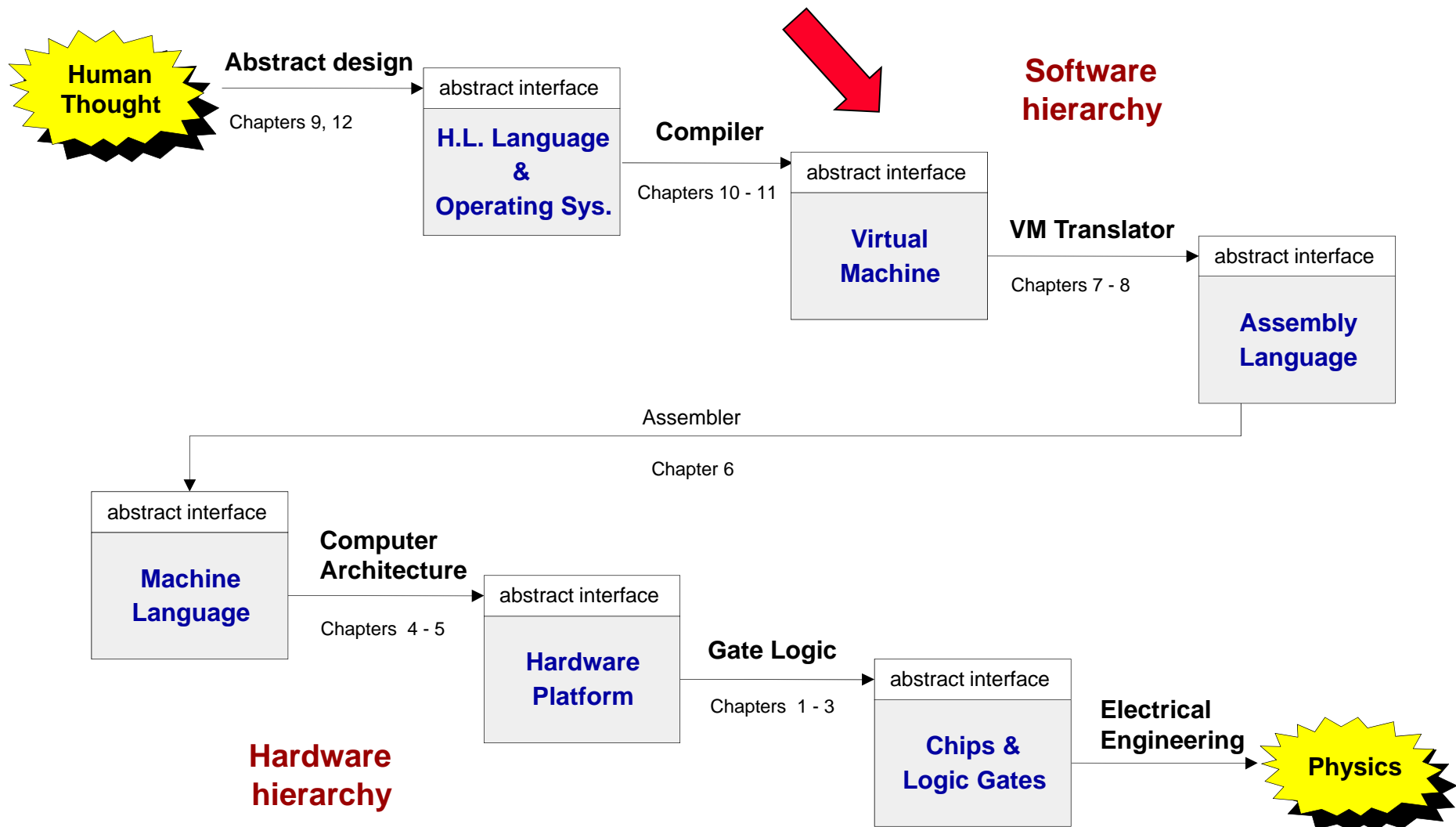
Part II: Program Control



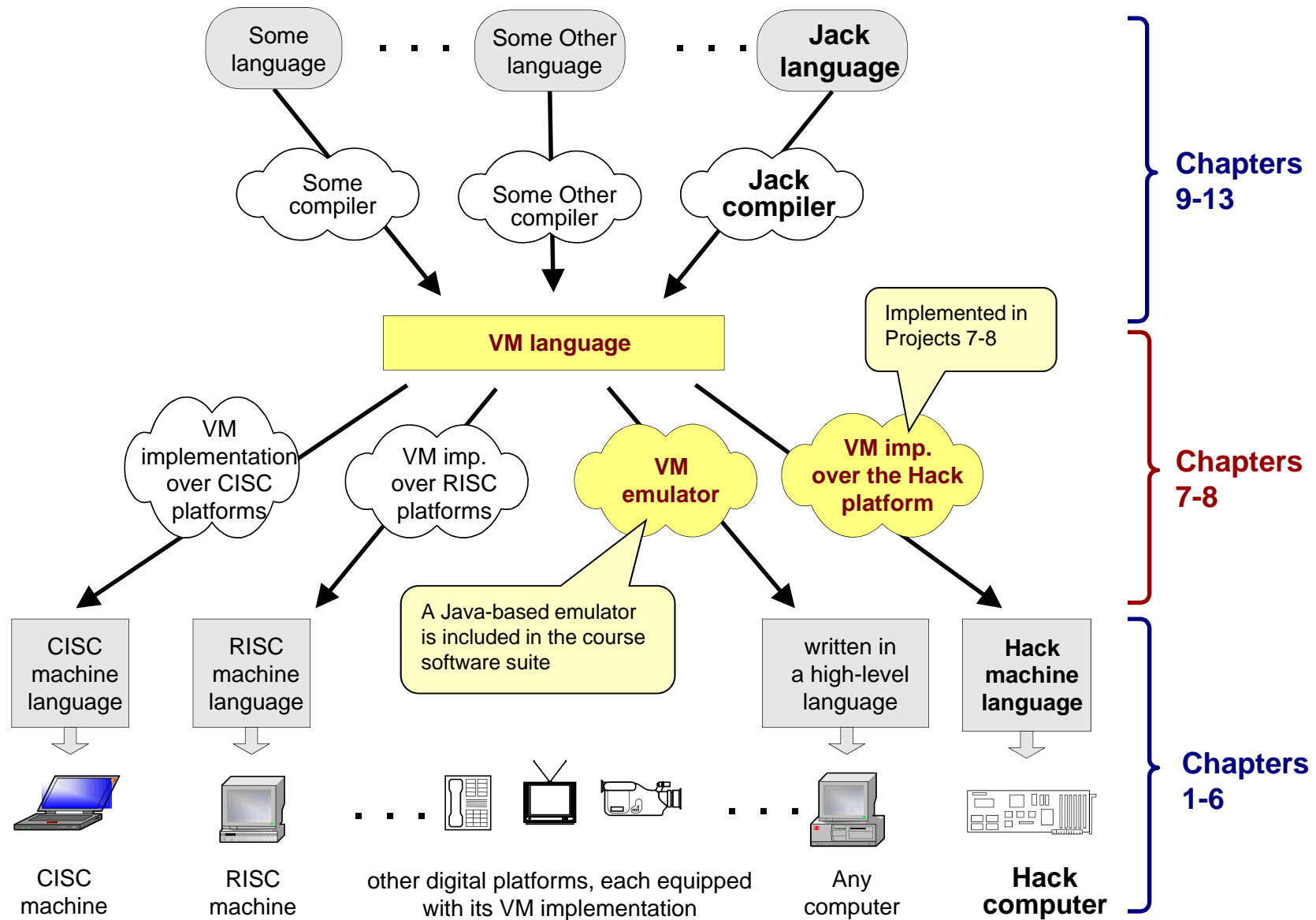
Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



The big picture



The VM language

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

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

previous
lecture

this
lecture

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 contLabel  
elseLabel:  
push c  
neg  
push b  
call div  
pop x  
contLabel:
```

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

Memory access commands

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

previous
lecture



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 three abstractions into assembly?

- ❑ 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.

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)

previous
lecture



Program flow commands

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

Function calling commands

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

Subroutines

```
// Compute  $x = (-b + \sqrt{b^2 - 4ac}) / 2a$ 
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

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

Subroutines in the VM language

Calling code (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
```

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 called command / function is responsible for removing the argument(s) from the stack, and for popping onto the stack the result of its execution.

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.

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)

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 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`.

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

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 .

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

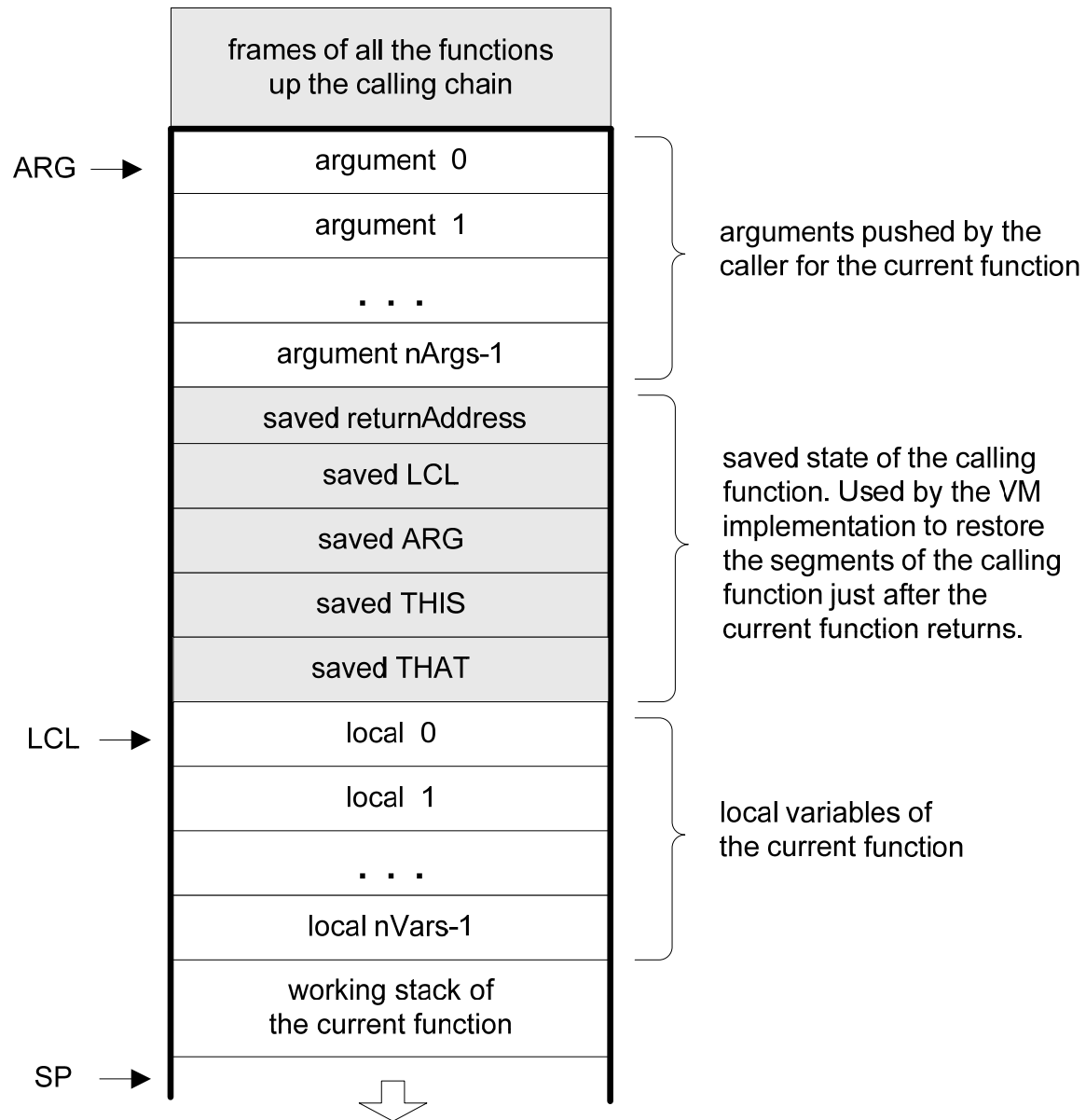
When g terminates and control should return to f , the VM implementation must:

- ❑ 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:

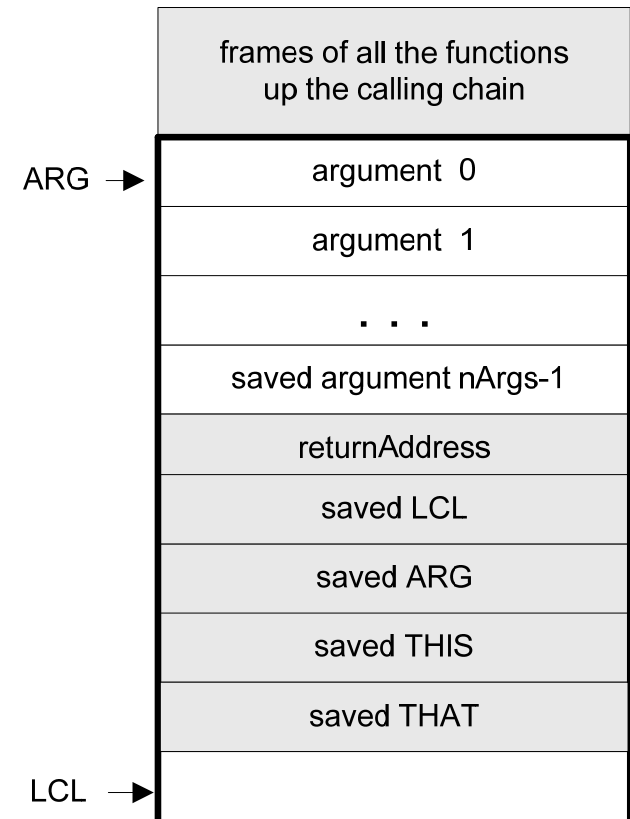
The stack that the current function sees

- 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
```



None of this code is executed yet ...
At this point we are just *generating code* (or simulating the VM code on some platform)

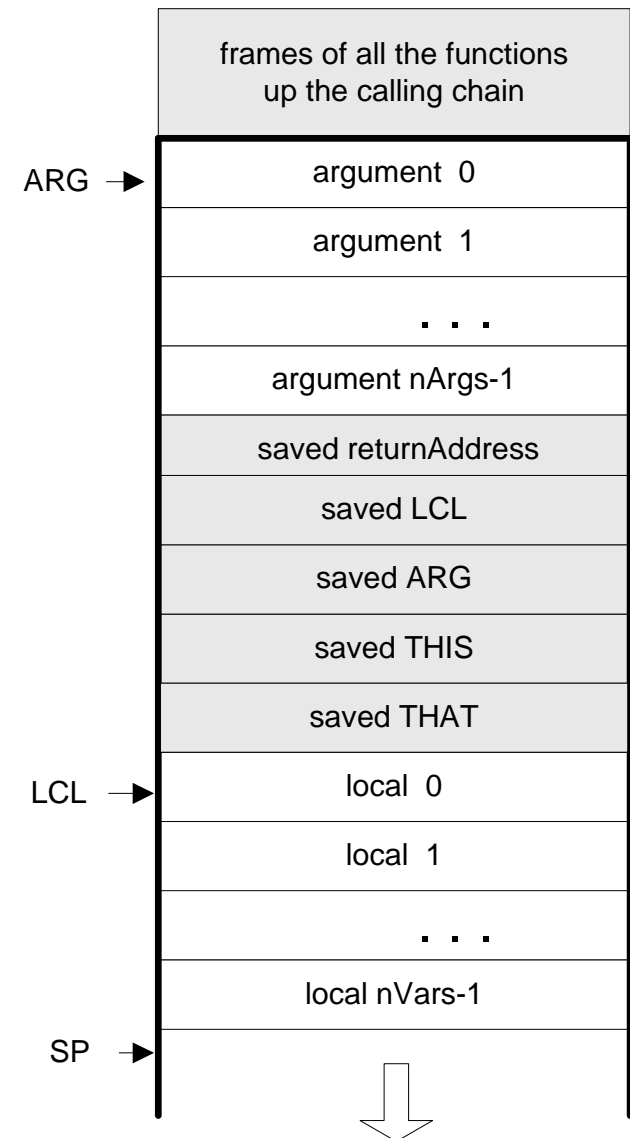
Implementation: If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

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
```

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.

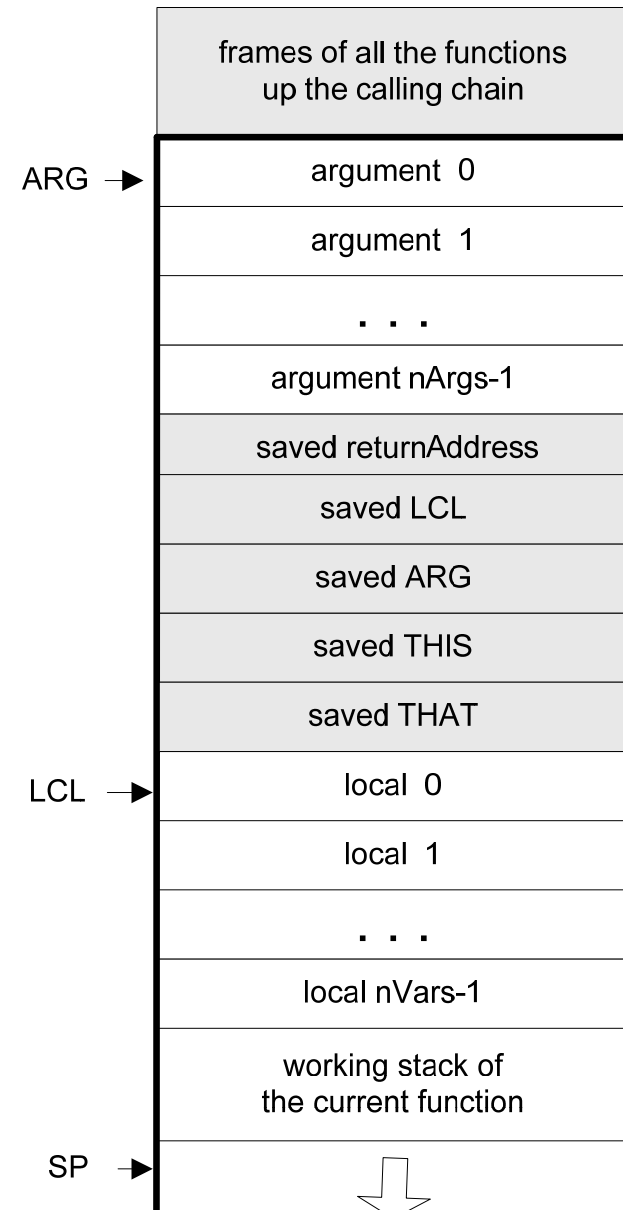


Implementing the `return` command

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.



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
```

VM implementation over the Hack platform

- Extends the VM implementation described in the last lecture (chapter 7)
- The result: a single assembly program file with lots of agreed-upon symbols:

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

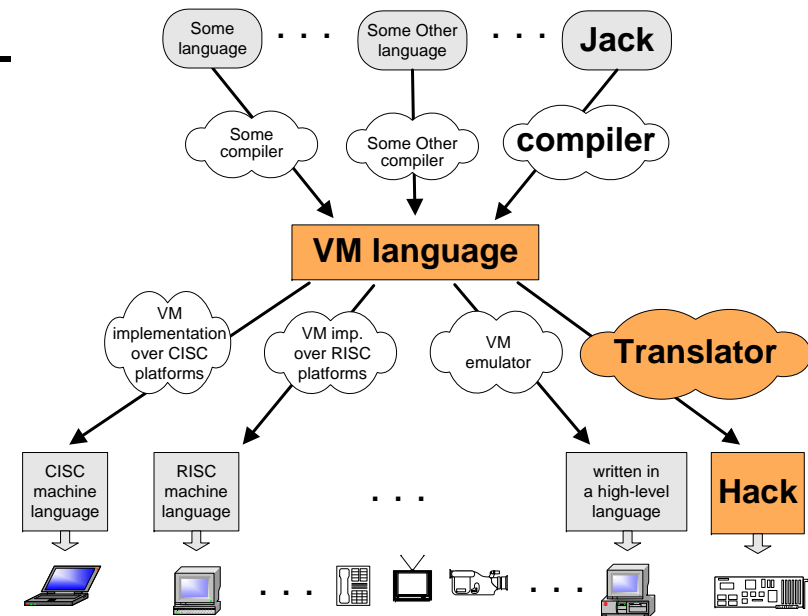
CodeWriter: Translates VM commands into Hack assembly code. The routines listed here should be added to the CodeWriter module API given in chapter 7.

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