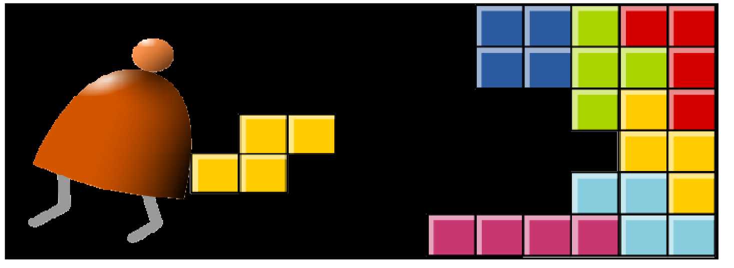
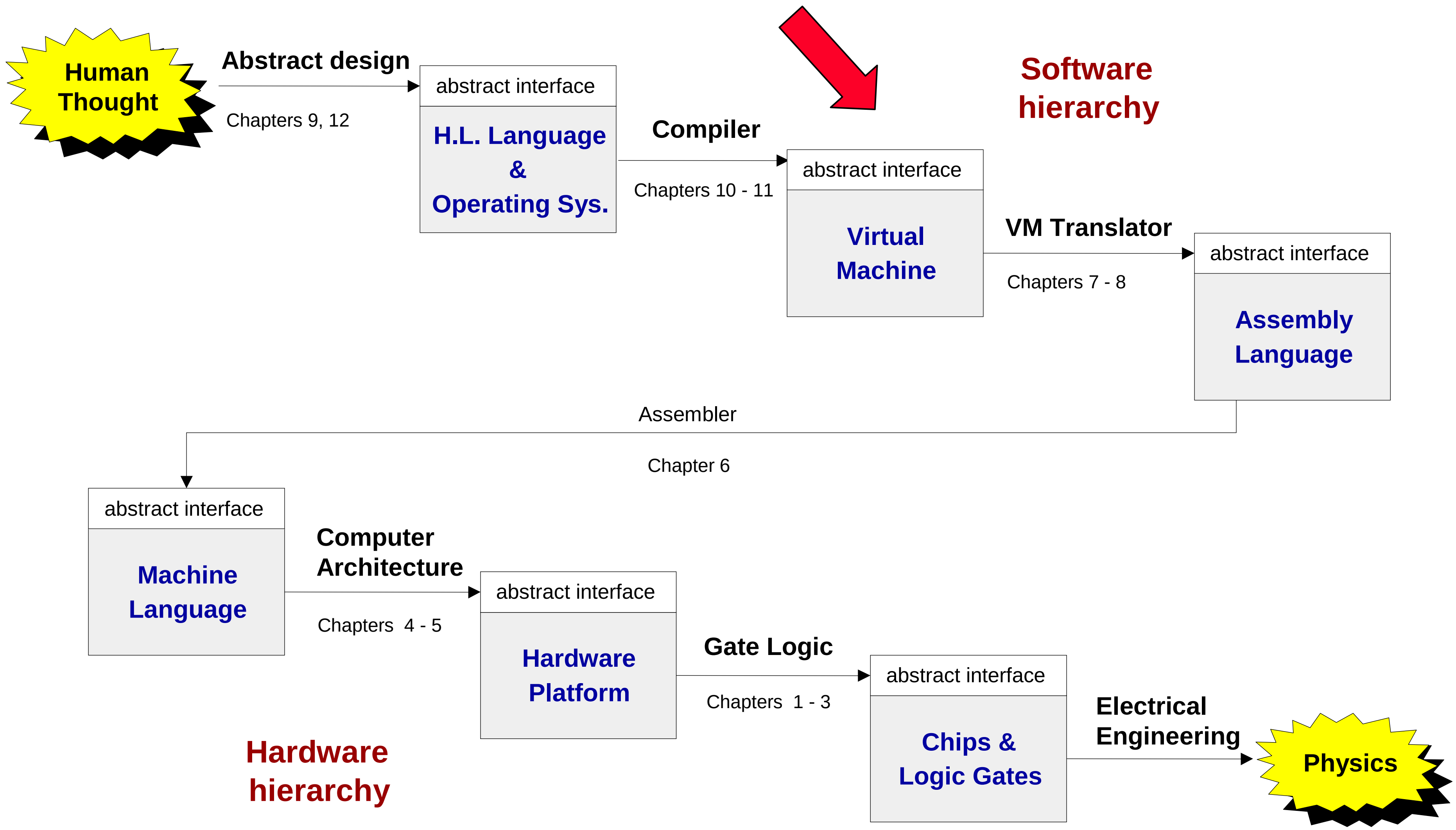
Virtual Machine I:
Stack Arithmetic



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



Motivation

```
class Main {
  static int x;

  function void main() {
    // Input and multiply 2 numbers
    var int a, b, x;
    let a = Keyboard.readInt("Enter a number");
    let b = Keyboard.readInt("Enter a number");
    let x = mult(a,b);
    return;
  }
}

  // Multiplies two numbers.
  function int mult(int x, int y) {
    var int result, j;
    let result = 0; let j = y;
    while not(j = 0) {
      let result = result + x;
      let j = j – 1;
    }
    return result;
  }
}
```

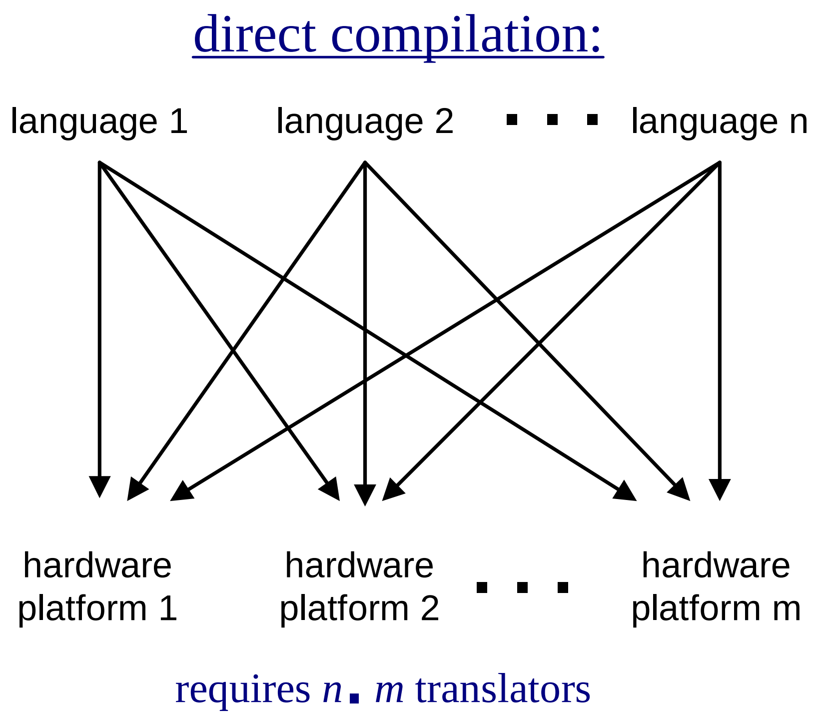
Ultimate goal:

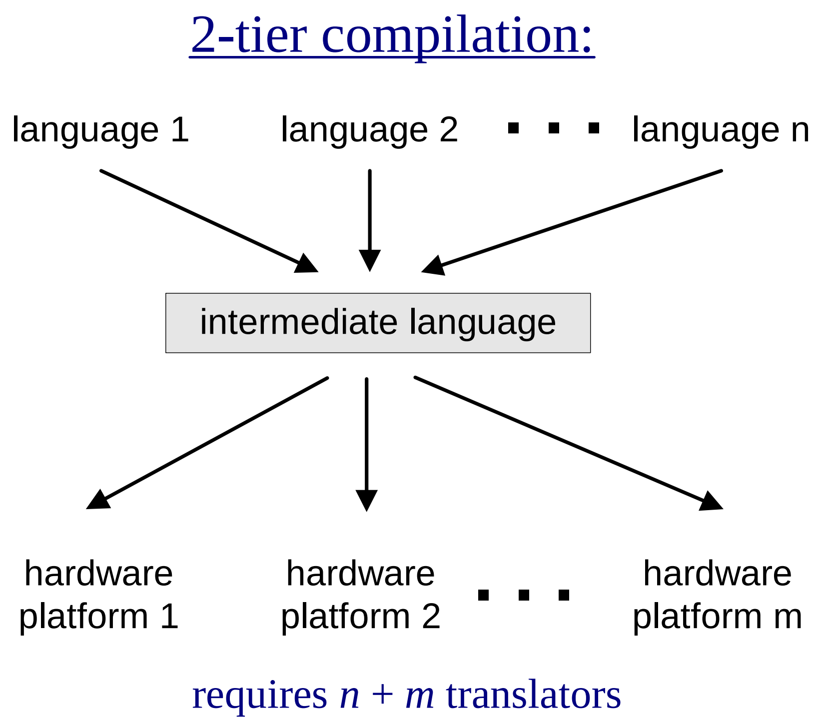
Translate high-level programs into executable code.

Compiler

```
   ...
   @a
   M=D
   @b
   M=0
(LOOP)
   @a
   D=M
   @b
   D=D-A
   @END
   D;JGT
   @j
   D=M
   @temp
   M=D+M
   @j
   M=M+1
   @LOOP
   0;JMP
(END)
   @END
   0;JMP
   ...
```

Compilation models

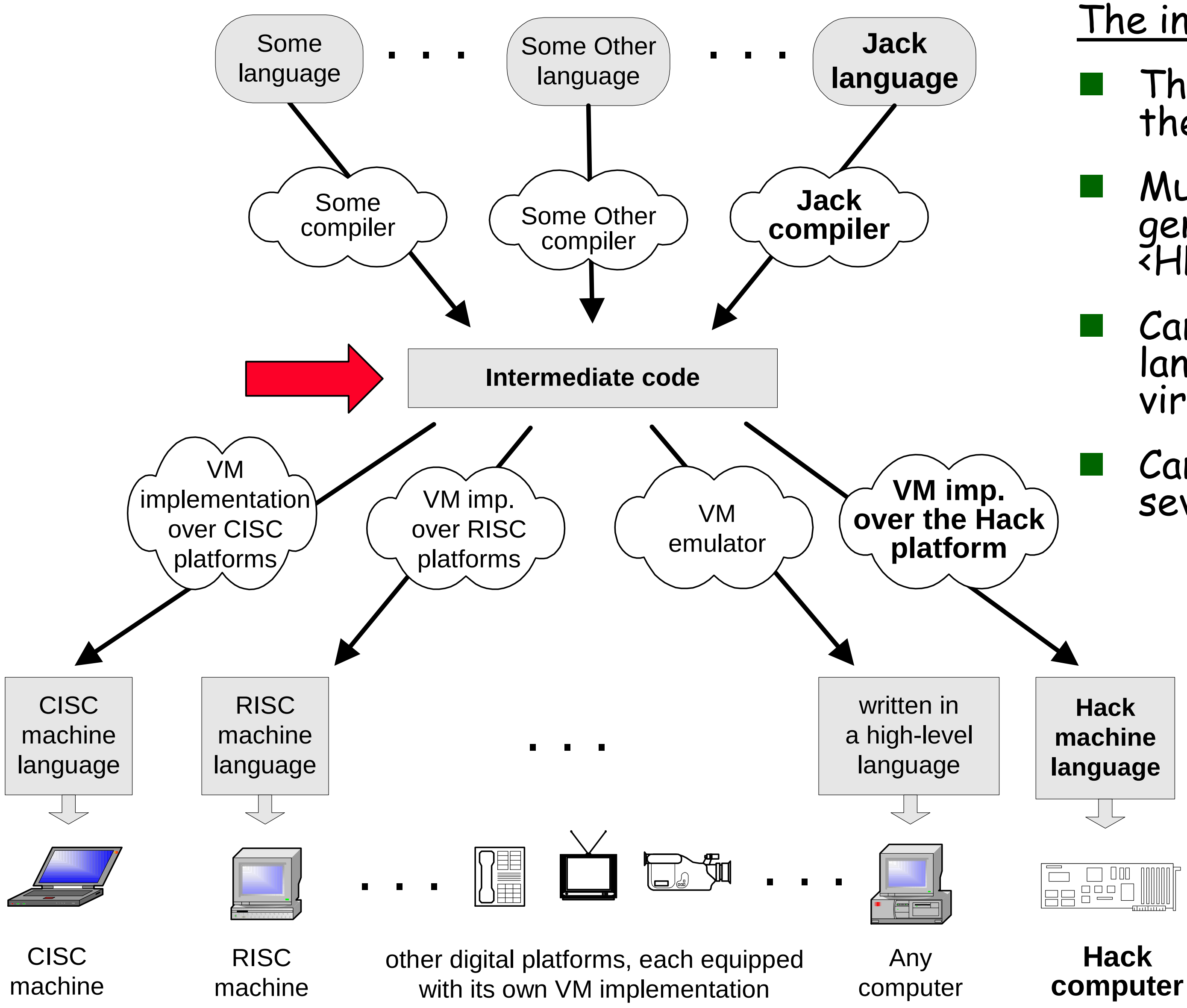




Two-tier compilation:

- 1st compilation stage depends only on the details of the source language
- 2nd compilation stage depends only on the details of the target platform.

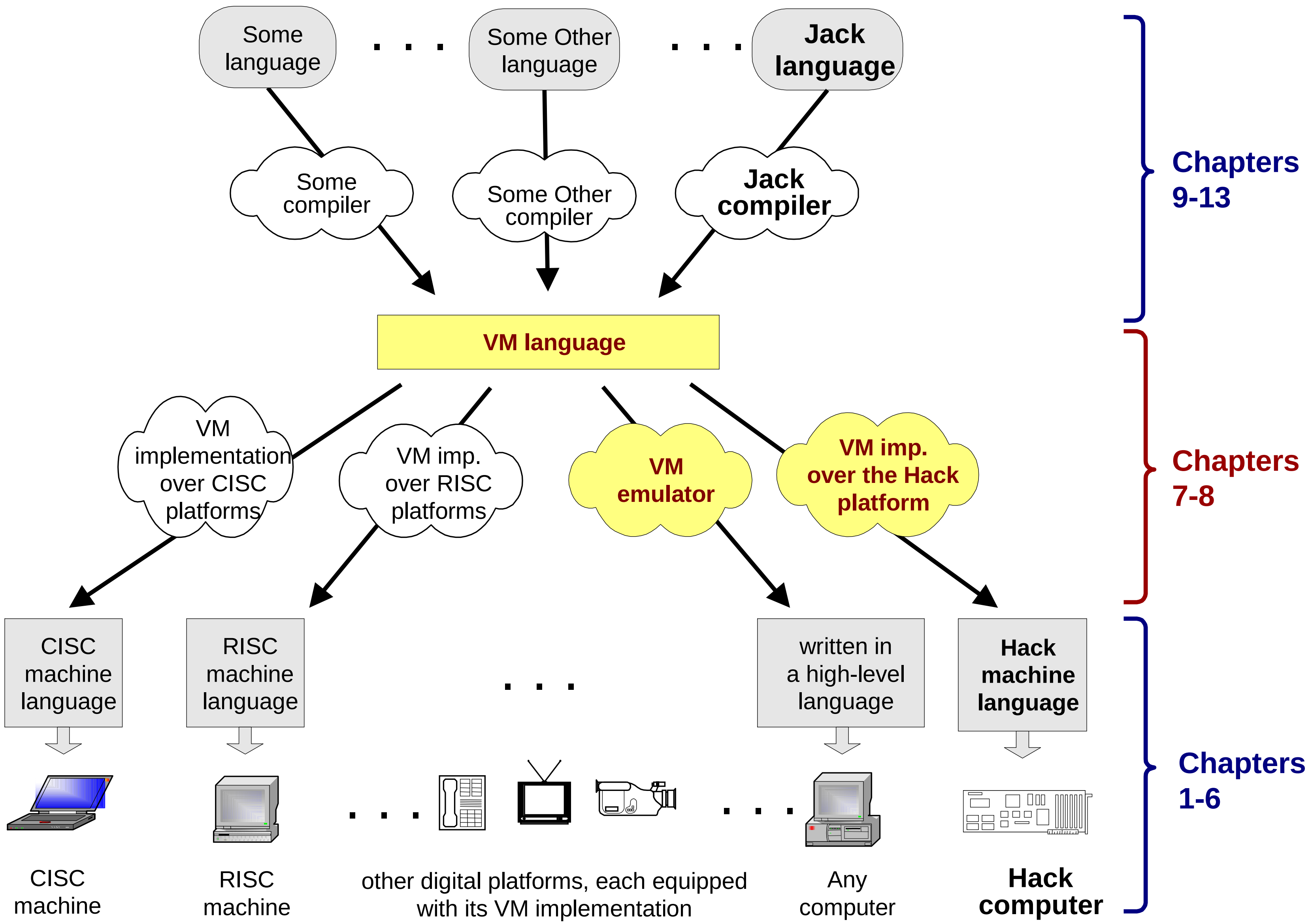
The big picture



The intermediate code:

- The interface between the 2 compilation stages
- Must be sufficiently general to support many <HLL , ML> pairs
- Can be modeled as the language of an abstract virtual machine (VM)
- Can be implemented in several different ways.

Focus of this lecture:



Lecture plan

Goal: Specify and implement a VM model and language:

Arithmetic / Boolean commands

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

This 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)

Next lecture

Function calling commands

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

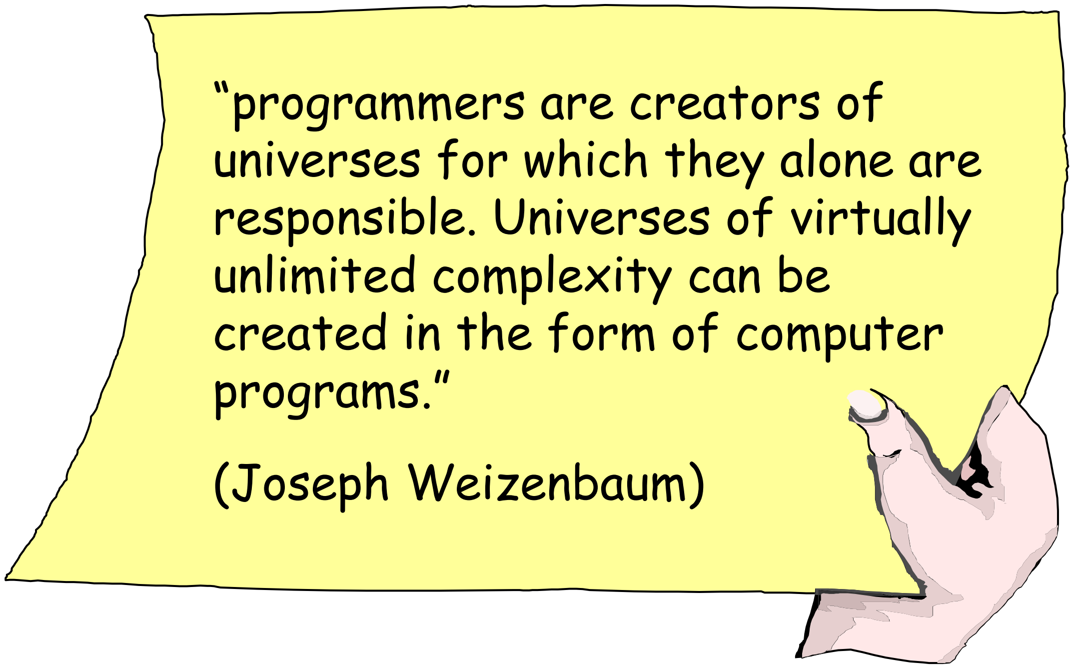
Method: (a) specify the abstraction (model's constructs and commands)
(b) propose how to implement it over the Hack platform.

The VM model and language

Perspective:

From here till the end of this and the next lecture we describe the VM model used in the Hack-Jack platform

Other VM models (like JVM/JRE and IL/CLR) are similar in spirit but different in scope and details.

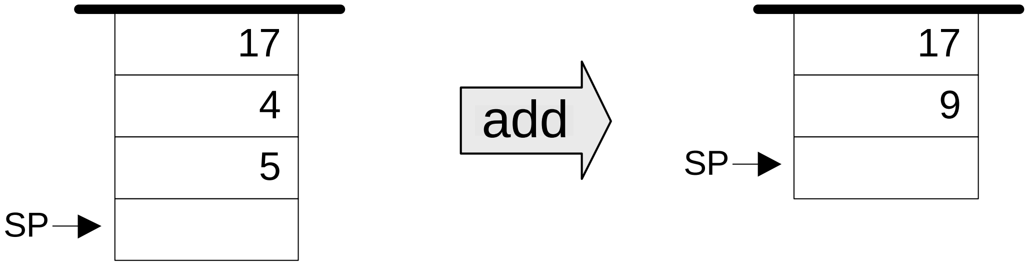


The VM model + language are an example of one such universe

View I: the VM is a machine that makes sense in its own right

View II: the VM is a convenient "in-between" representation of a computer program: an interim "station" between the high-level code and the machine level code.

Our VM is a stack-oriented machine



Our VM features a single 16-bit data type that can be used as:

- Integer
- Boolean
- Pointer

Typical operation:

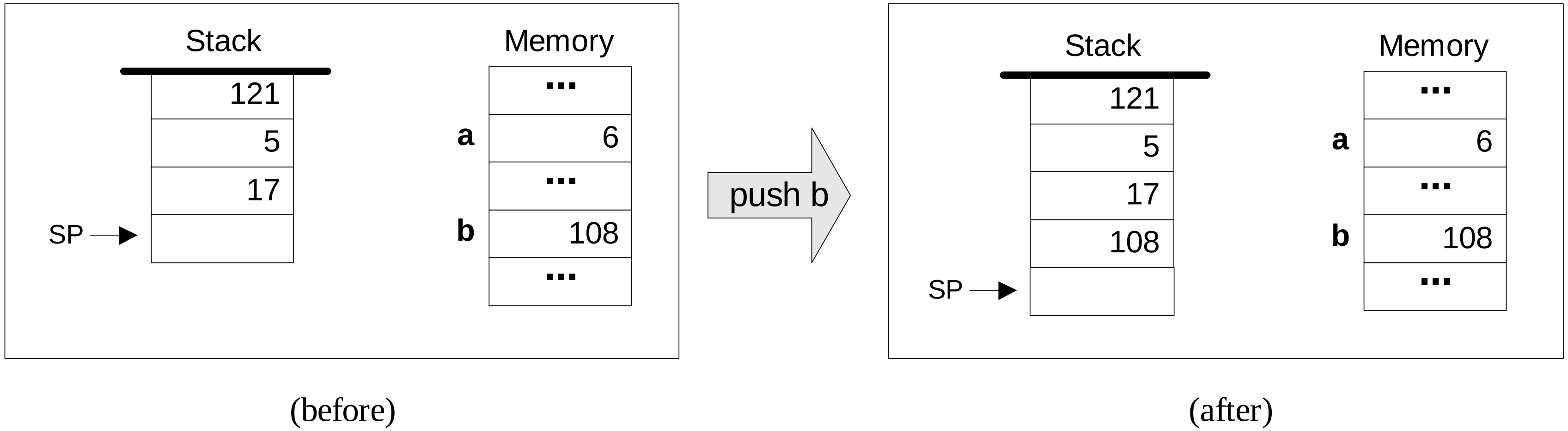
- Pops topmost values x,y from the stack
- Computes the value of some function f(x,y)
- Pushes the result onto the stack

(Unary operations are similar, using x and f(x) instead)

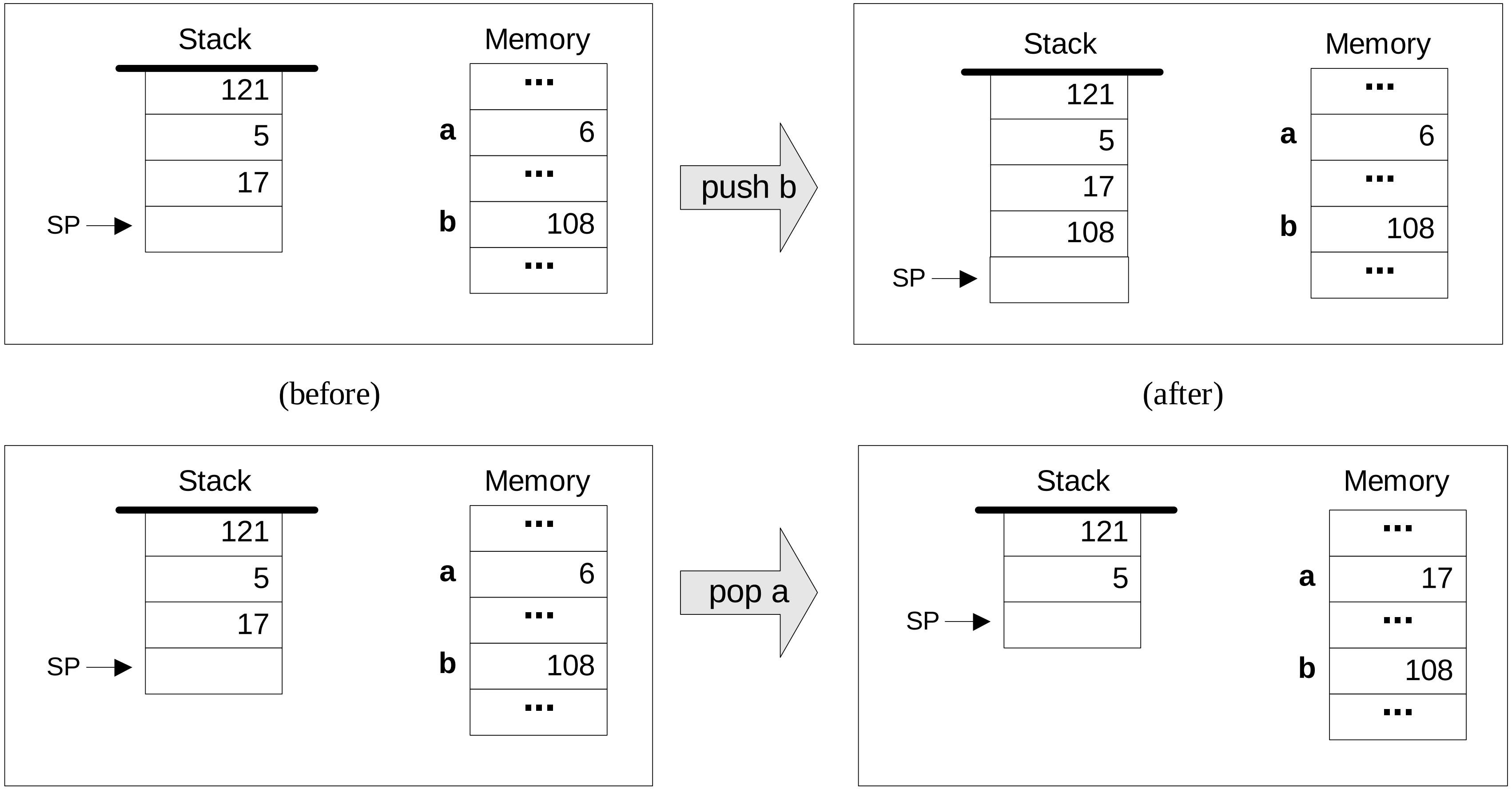
Impact: the operands are replaced with the operation's result

In general: all arithmetic and Boolean operations are implemented similarly.

Memory access (first approximation)



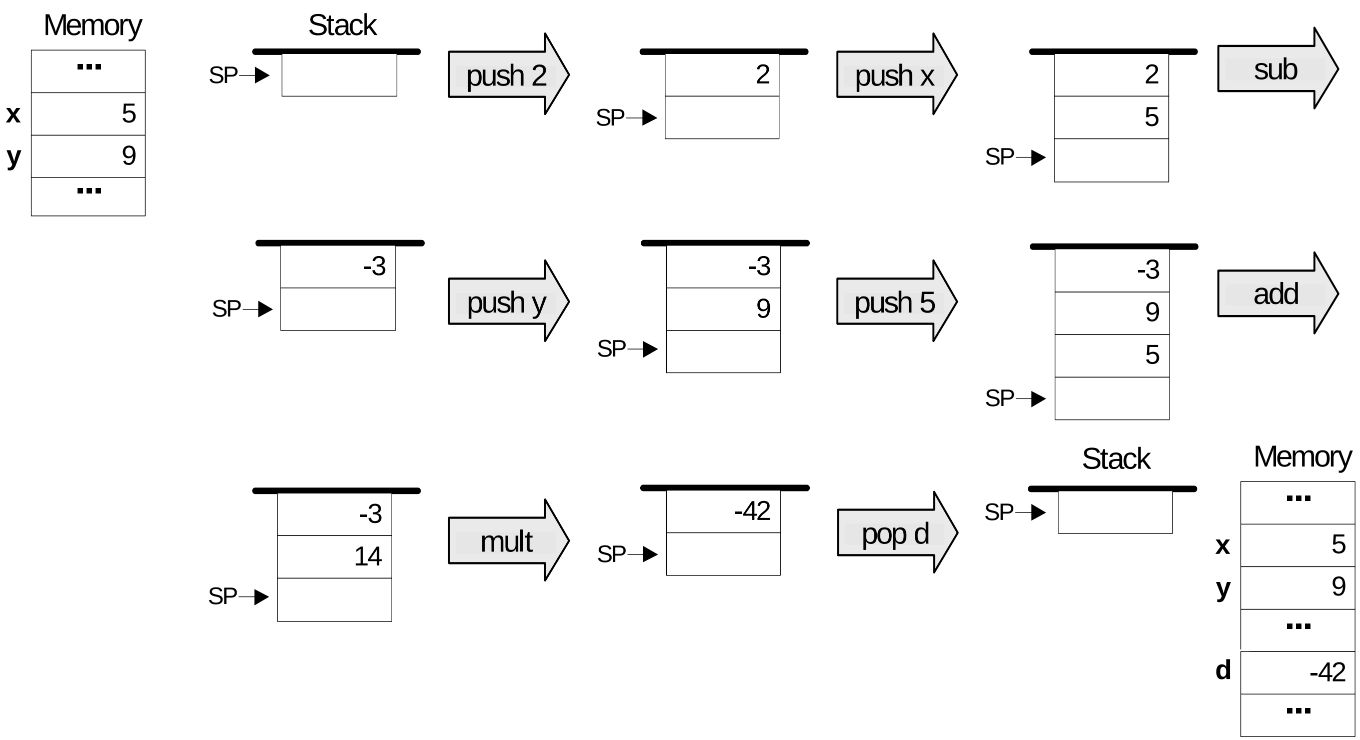
Memory access (first approximation)



- Classical data structure
- Elegant and powerful
- Several hw/sw implementation options.

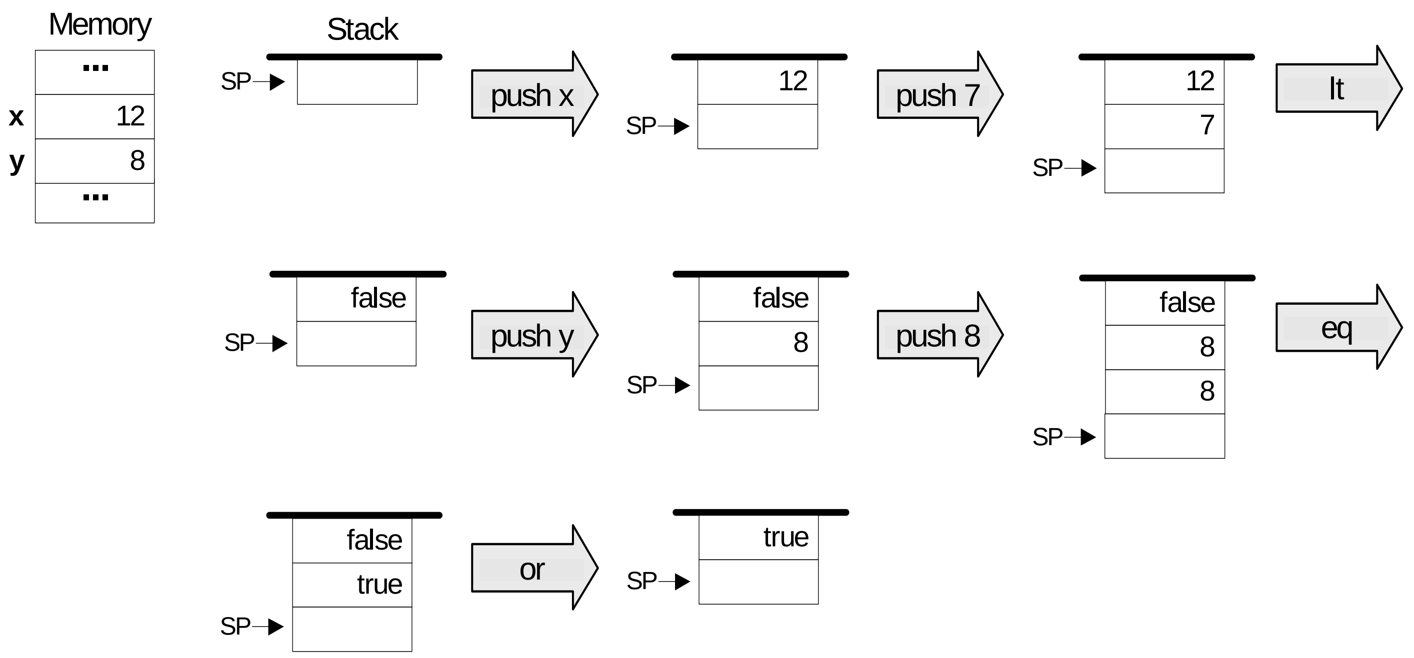
Evaluation of arithmetic expressions

```
// d=(2-x)*(y+5)
push 2
push x
sub
push y
push 5
add
mult
pop d
```



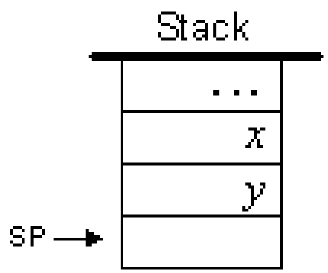
Evaluation of Boolean expressions

```
// if (x<7) or (y=8)
push x
push 7
lt
push y
push 8
eq
or
```



Arithmetic and Boolean commands (wrap-up)

Command	**Return value** (after popping the operand/s)	**Comment**	
add	$x + y$	Integer addition	(2's complement)
sub	$x - y$	Integer subtraction	(2's complement)
neg	$-y$	Arithmetic negation	(2's complement)
eq	true if $x = y$ and false otherwise	Equality	
gt	true if $x > y$ and false otherwise	Greater than	
lt	true if $x < y$ and false otherwise	Less than	
and	x And y	Bit-wise	
or	x Or y	Bit-wise	
not	Not y	Bit-wise	



Memory segments

Modern programming languages normally feature the following variable kinds:

Class level

- Static variables
- Private variables (AKA "object variables" / "fields" / "properties")

Method level:

- Local variables
- Argument variables

The VM abstraction must support (at least) these variable kinds.
In our VM model, these variables are stored in *virtual memory segments*

Specifically, our VM model consists of 8 memory segments:
`static`, **`this`**, **`local`**, **`argument`**
As well as:
`that`, **`constant`**, **`pointer`**, and **`temp`**.

Memory segments and memory access commands

So we have 8 virtual memory segments:

`static`, **`this`**, **`local`**, **`argument`**, **`that`**, **`constant`**, **`pointer`**, **`temp`**

But, at the VM level, there is no need to differentiate among the different roles of these segments

As far as VM programming commands go, a segment is a segment is a segment;
to access a particular segment entry, use the syntax:

<u>Memory access command format:</u>

pop *segment i*

push *segment i*

- These commands are used instead of **pop** *x* and **push** *y*, as shown in previous slides, which was a conceptual simplification
- The meaning of the different eight segments will become important when we'll talk about the compiler.

VM programming

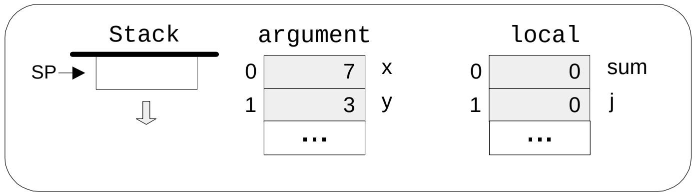
- VM programs are normally written by *compilers*, not by humans
- But, for a human to write or optimize a compiler, the human must first understand the spirit of VM programming
- So, here is an example.

Arithmetic example

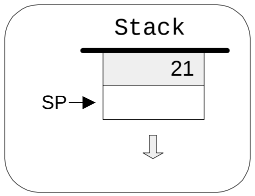
High-level code

```
function mult (x,y) {
  int result, j;
  result = 0;
  j = y;
  while ~(j == 0) {
    result = result + x;
    j = j - 1;
  }
  return result;
}
```

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



VM code (first approx.)

```
function mult(x,y)
   push 0
   pop result
   push y
   pop j
label loop
   push j
   push 0
   eq
   if-goto end
   push result
   push x
   add
   pop result
   push j
   push 1
   sub
   pop j
   goto loop
label end
   push result
   return
```

VM code

```
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
```

Lecture plan

Goal: Specify and implement a VM model and language

Arithmetic / Boolean commands

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

This lecture

Memory access commands

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

Program flow commands

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

Next lecture

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.

Implementation

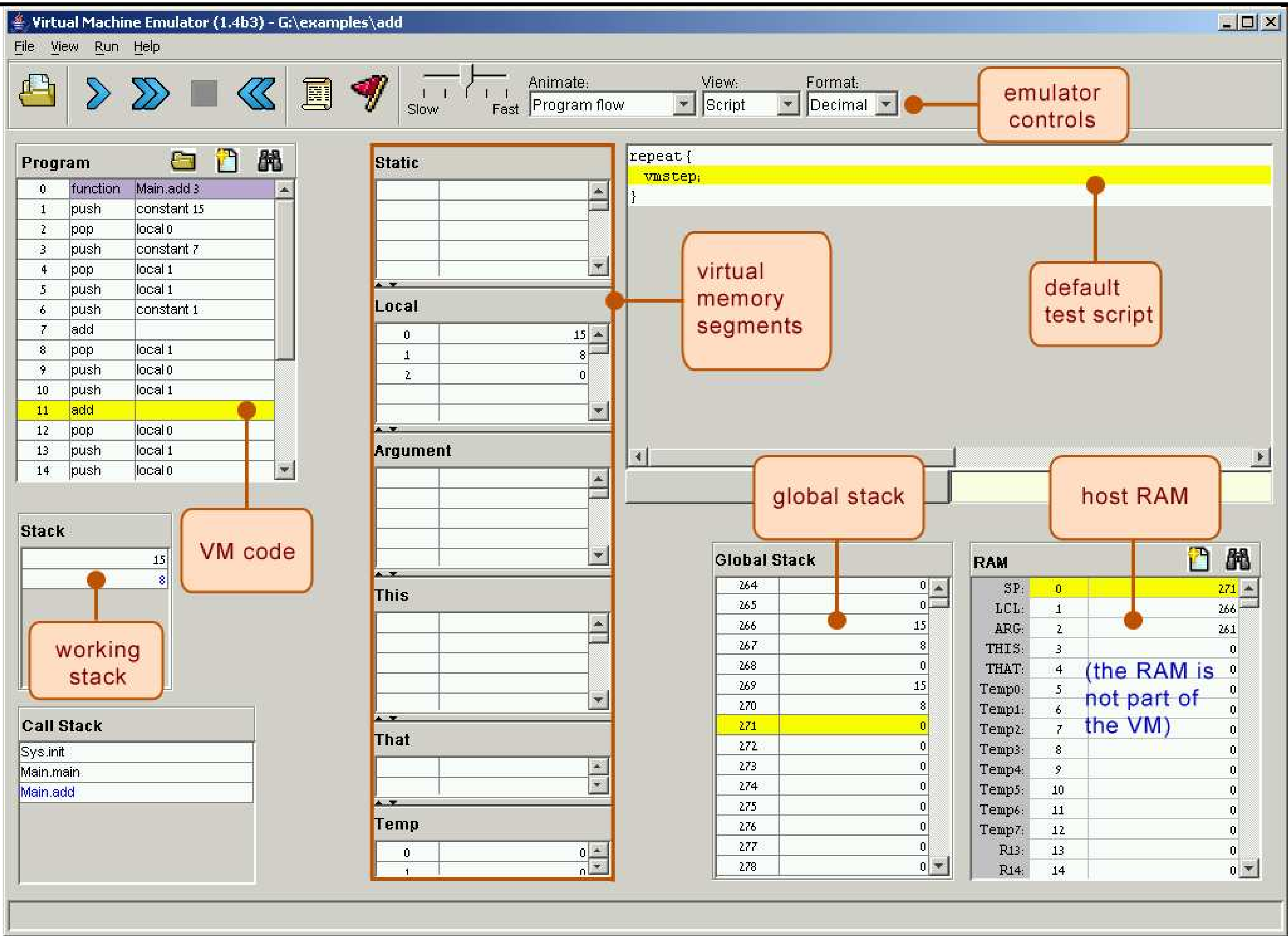
VM implementation options:

- Software-based (emulate the VM using, say, Java)
- Translator-based (translate VM programs into, say, the Hack language)
- Hardware-based (realize the VM using dedicated memory and registers)

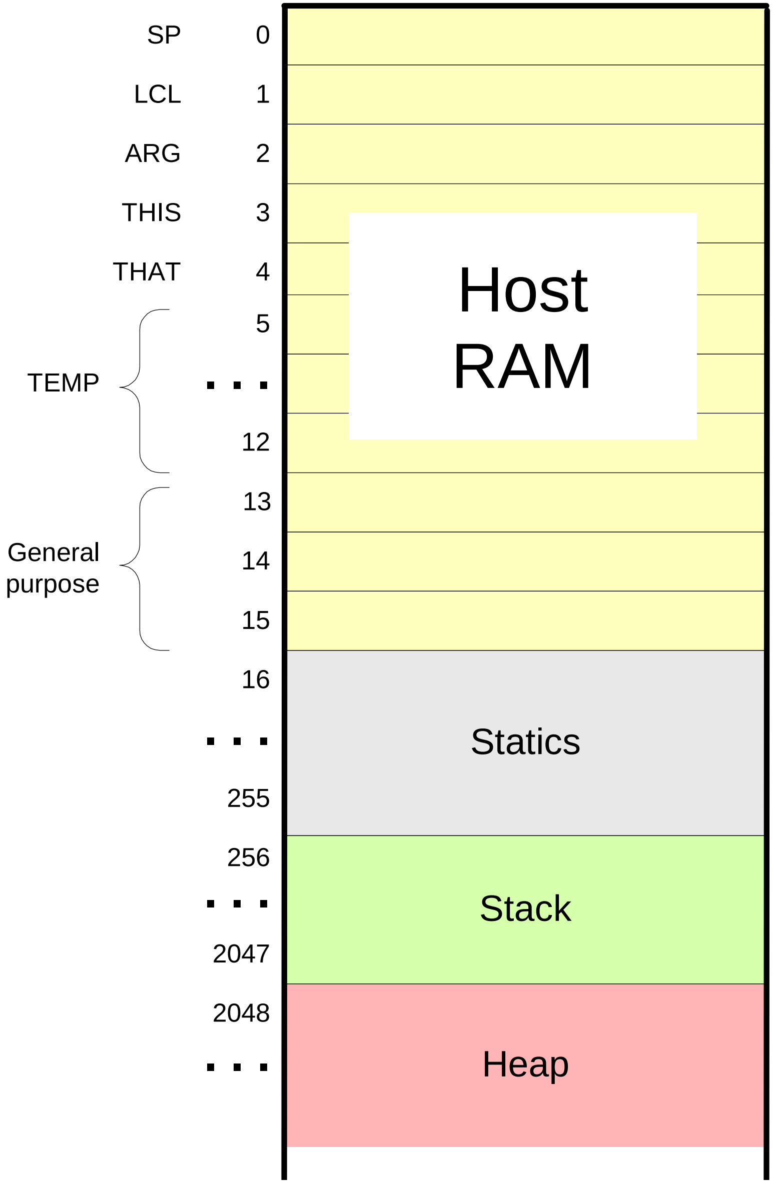
Well-known translator-based implementations:

- JVM (runs bytecode programs in the Java platform)
- CLR (runs IL programs in the .NET platform).

Our VM emulator (part of the course software suite)



Standard VM implementation on the Hack platform



The challenge: (i) map the VM constructs on the host RAM, and (ii) given this mapping, figure out how to implement each VM command using assembly commands that operate on the RAM

- `local,argument,this,that`: mapped on the heap. The base addresses of these segments are kept in `LCL,ARG,THIS,THAT`. Access to the *i*-th entry of a segment is implemented by accessing the segment's (base + *i*) word in the RAM
- `static`: static variable number *j* in a VM file *f* is implemented by the assembly language symbol `f.j` (and recall that the assembler maps such symbols to the RAM starting from address 16)
- `constant`: truly a virtual segment: Access to `constant i` is implemented by supplying the constant *i*
- `pointer`: used to align `this` and `that` with memory blocks on the heap

Exercise: given the above game rules, write the Hack commands that implement, say, `push constant 5` and `pop local 2`.

Proposed VM translator implementation: Parser module

Parser: Handles the parsing of a single `.vm` file, and encapsulates access to the input code. It reads VM commands, parses them, and provides convenient access to their components. In addition, it removes all white space and comments.

Routine	**Arguments**	**Returns**	**Function**
Constructor	Input file / stream	--	Opens the input file/stream and gets ready to parse it.
`hasMoreCommands`	--	boolean	Are there more commands in the input?
`advance`	--	--	Reads the next command from the input and makes it the current command. Should be called only if `hasMoreCommands()` is true. Initially there is no current command.
`commandType`	--	`C_ARITHMETIC, C_PUSH, C_POP, C_LABEL, C_GOTO, C_IF, C_FUNCTION, C_RETURN, C_CALL`	Returns the type of the current VM command. `C_ARITHMETIC` is returned for all the arithmetic commands.
`arg1`	--	string	Returns the first argument of the current command. In the case of `C_ARITHMETIC`, the command itself (`add`, `sub`, etc.) is returned. Should not be called if the current command is `C_RETURN`.
`arg2`	--	int	Returns the second argument of the current command. Should be called only if the current command is `C_PUSH, C_POP, C_FUNCTION, or C_CALL.`

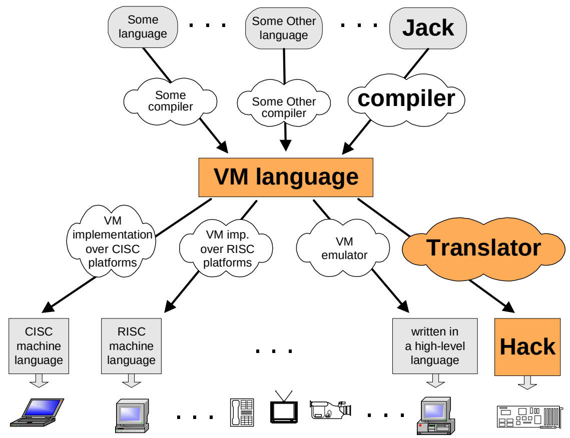
Proposed VM translator implementation: CodeWriter module

CodeWriter: Translates VM commands into Hack assembly code.			
Routine	**Arguments**	**Returns**	**Function**
Constructor	Output file / stream	--	Opens the output file/stream and gets ready to write into it.
`setFileName`	`fileName (string)`	--	Informs the code writer that the translation of a new VM file is started.
`writeArithmetic`	`command (string)`	--	Writes the assembly code that is the translation of the given arithmetic command.

`WritePushPop`	`Command (C_PUSH or C_POP),`
	`segment (string),`
	`index (int)`
		--	Writes the assembly code that is the translation of the given `command`, where `command` is either `C_PUSH or C_POP.`

`Close`	--	--	Closes the output file.
Comment: More routines will be added to this module in chapter 8.			

Perspective



- In this lecture we began the process of building a compiler
- Modern compiler architecture:
  - Front-end (translates from high level language to a VM language)
  - Back-end (translates from the VM language to the machine language of some target hardware platform)
- Brief history of virtual machines:
  - 1970's: p-Code
  - 1990's: Java's JVM
  - 2000's: Microsoft .NET
- A full blown VM implementation typically includes a common software library (can be viewed as a mini, portable OS).
- We will build such a mini OS later in the course.

The road ahead

Tasks:	Conceptually similar to:	And to:
Complete the VM specification and implementation (chapters 7,8)	JVM	CLR
Introduce Jack, a high-level programming language (chapter 9)	Java	C#
Build a compiler for it (chapters 10,11)	Java compiler	C# compiler
Finally, build a mini-OS, i.e. a run-time library (chapter 12).	JRE	.NET base class library