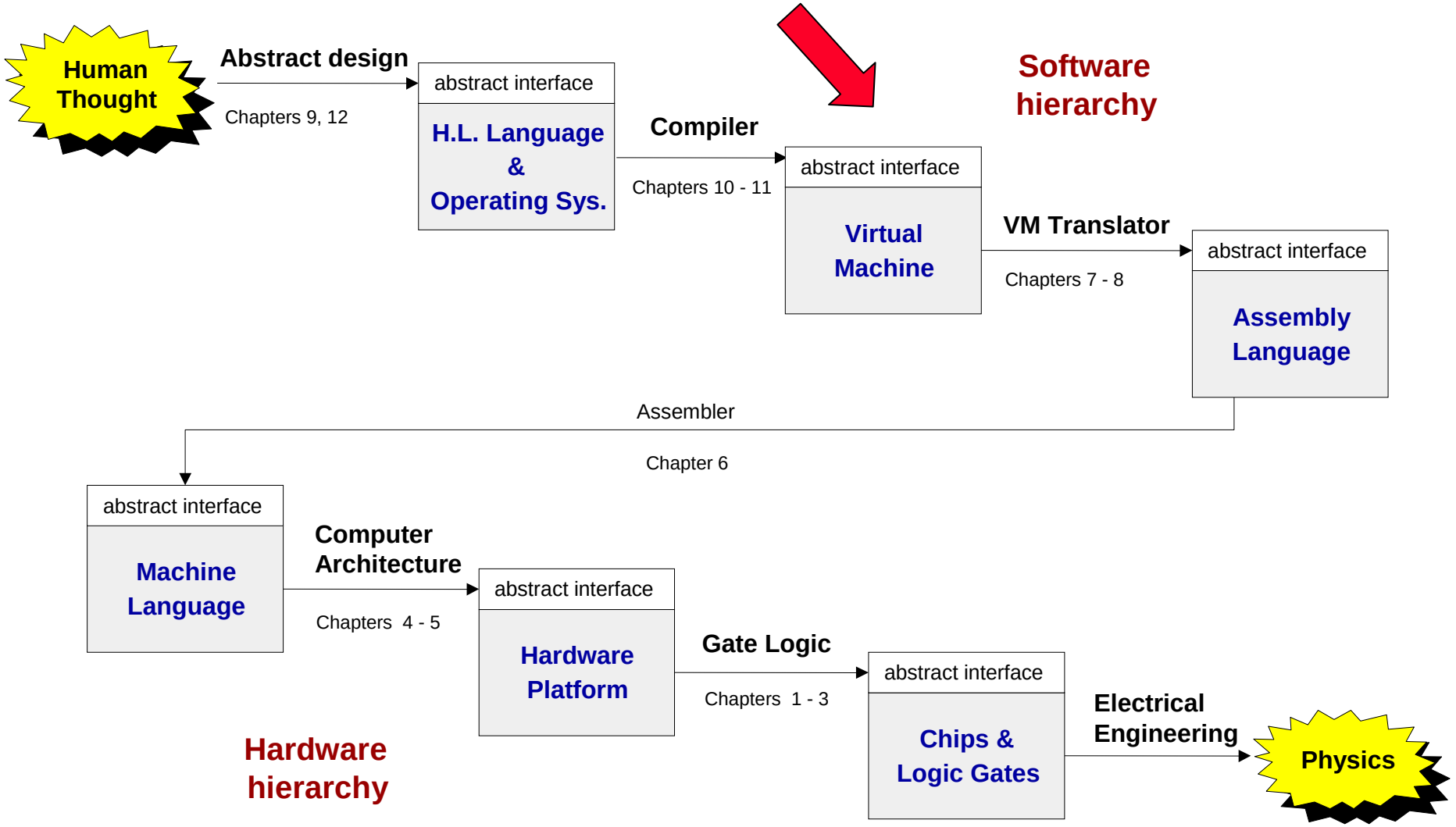
Virtual Machine
Part I: Stack Arithmetic

Where we are at:



Motivation

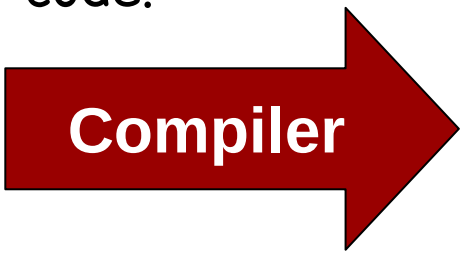
```
class Main {
  static int x;

  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:

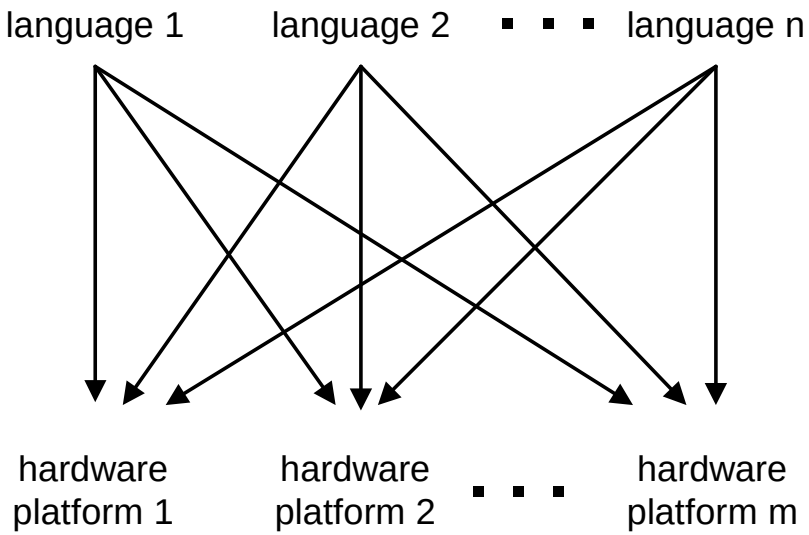
Translate high-level programs into executable code.




```
   ...
   @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
   ...
```

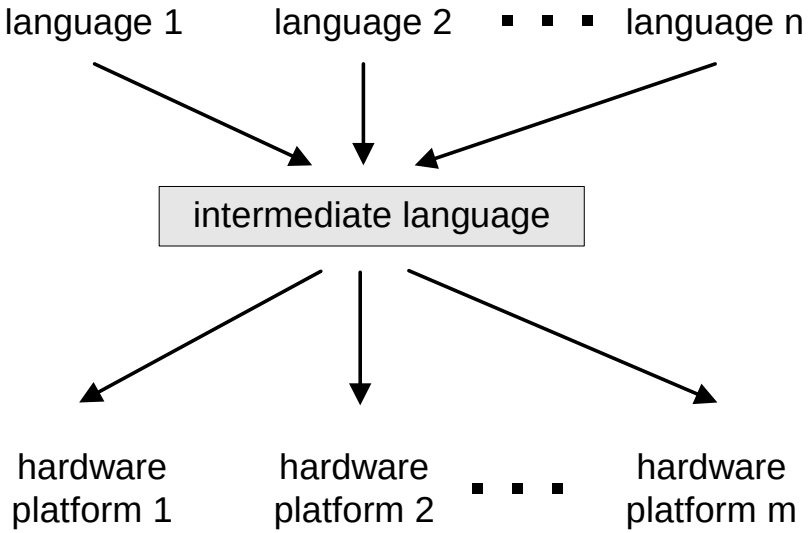
Compilation models

direct compilation:



requires $n \cdot m$ translators

2-tier compilation:

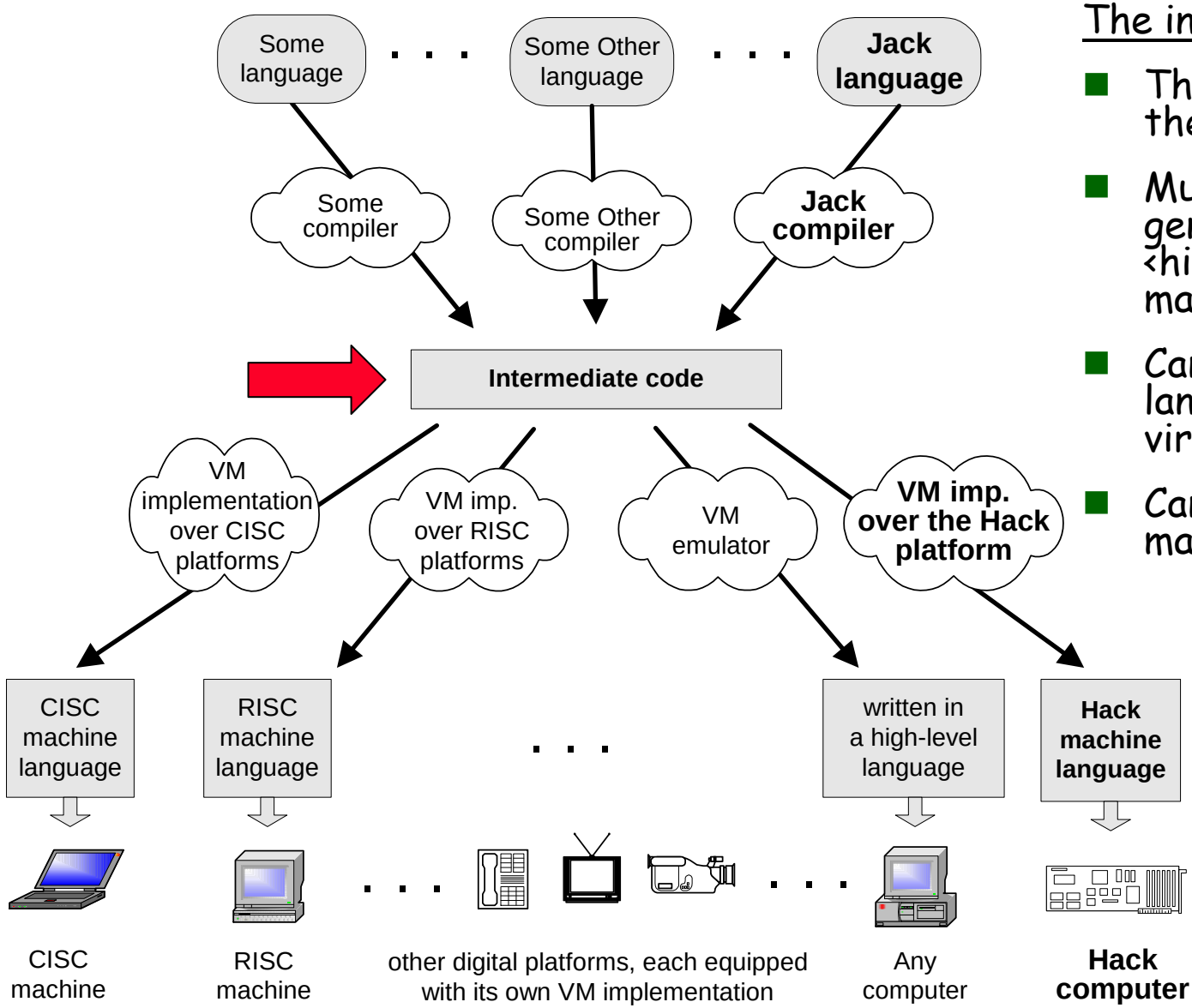


requires $n + m$ translators

Two-tier compilation:

- First compilation stage depends only on the details of the source language
- Second compilation stage depends only on the details of the target platform.

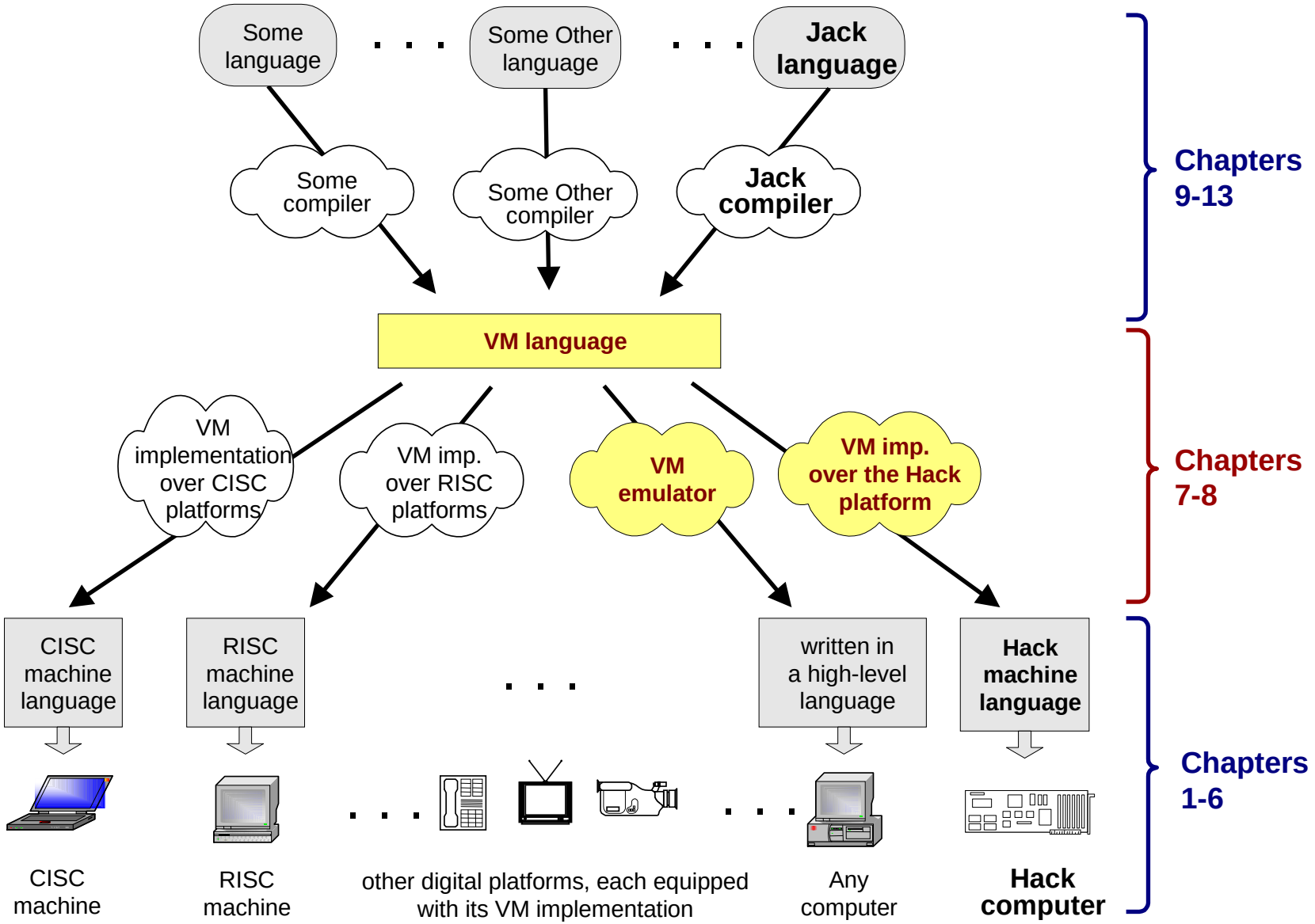
The big picture



The intermediate code:

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

The big picture



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` (variable)
- `push y` (variable or 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 language

Important:

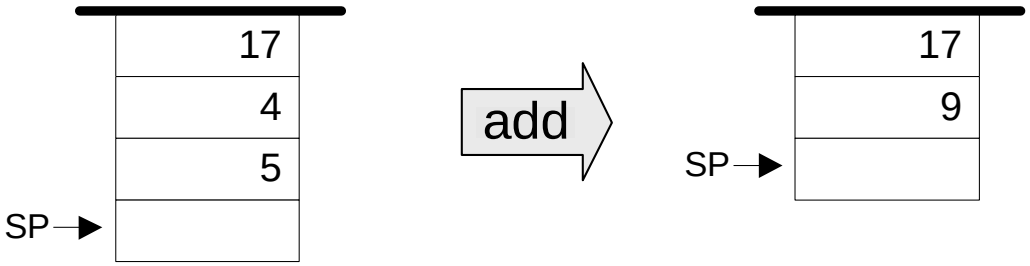
From here till the end of this and the next lecture we describe the VM model used in the Hack-Jack platform

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

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

- Integer
- Boolean
- Pointer.

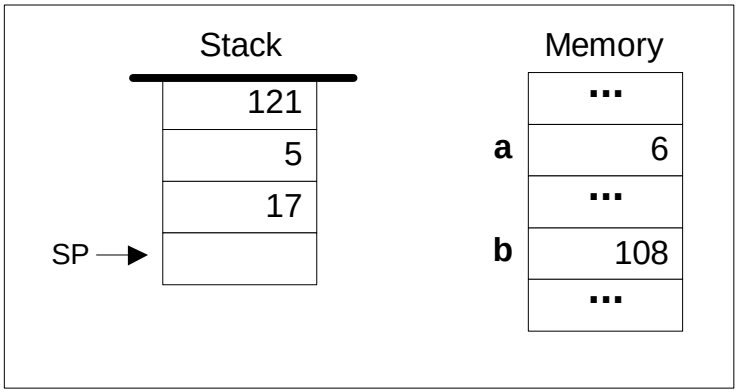
Stack arithmetic



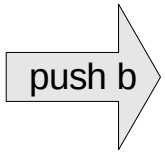
- Typical operation:
  - Pops topmost values x,y from the stack
  - Computes the value of some function f(x,y)
  - Pushes the result onto the stack

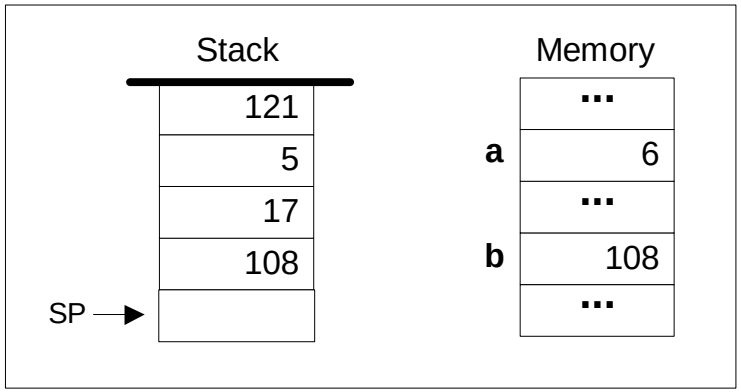
  (Unary operations are similar, using x and f(x) instead)
- Impact: the operands are replaced with the operation's result
- In general: all arithmetic and Boolean operations are implemented similarly.

Memory access (first approximation)



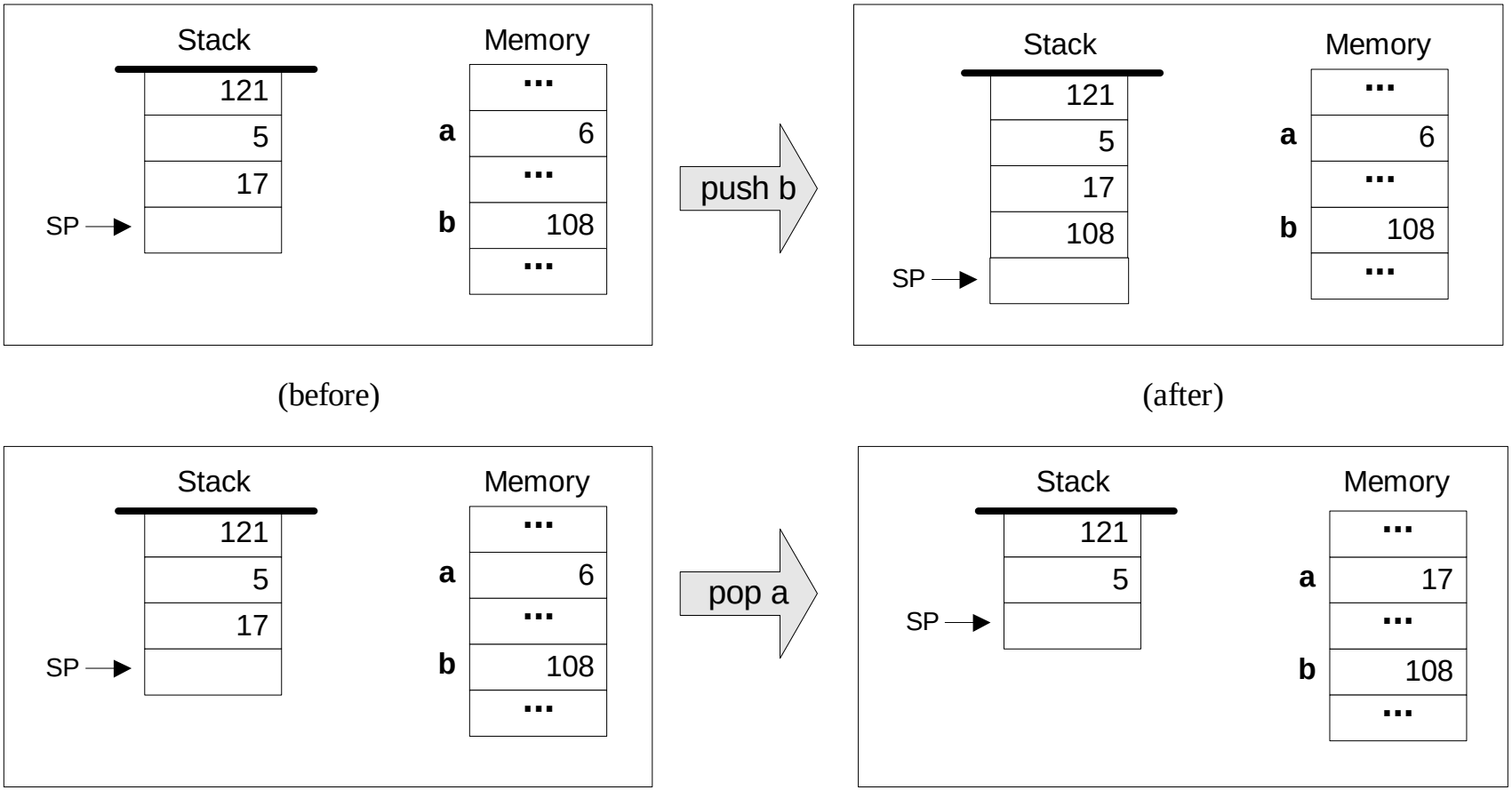
(before)





(after)

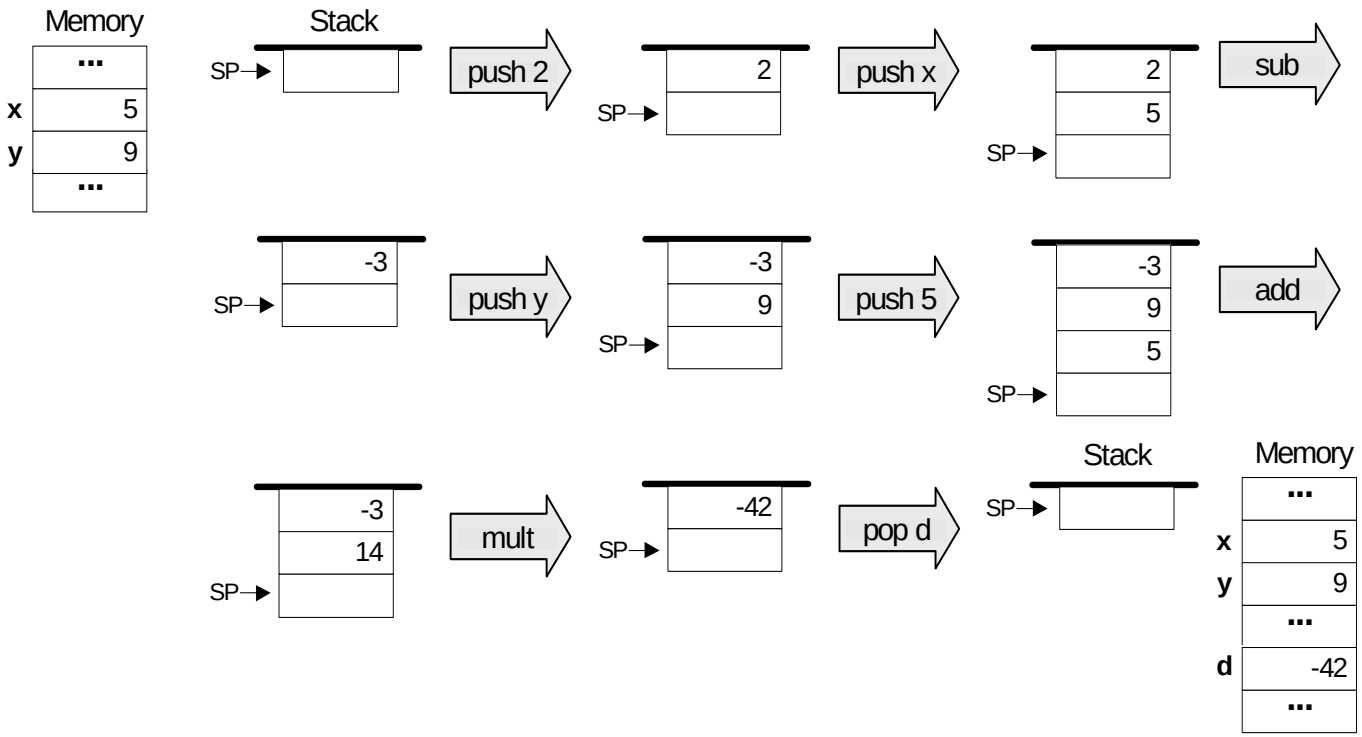
Memory access (first approximation)



- Classical data structure
- Elegant and powerful
- Many implementation options.

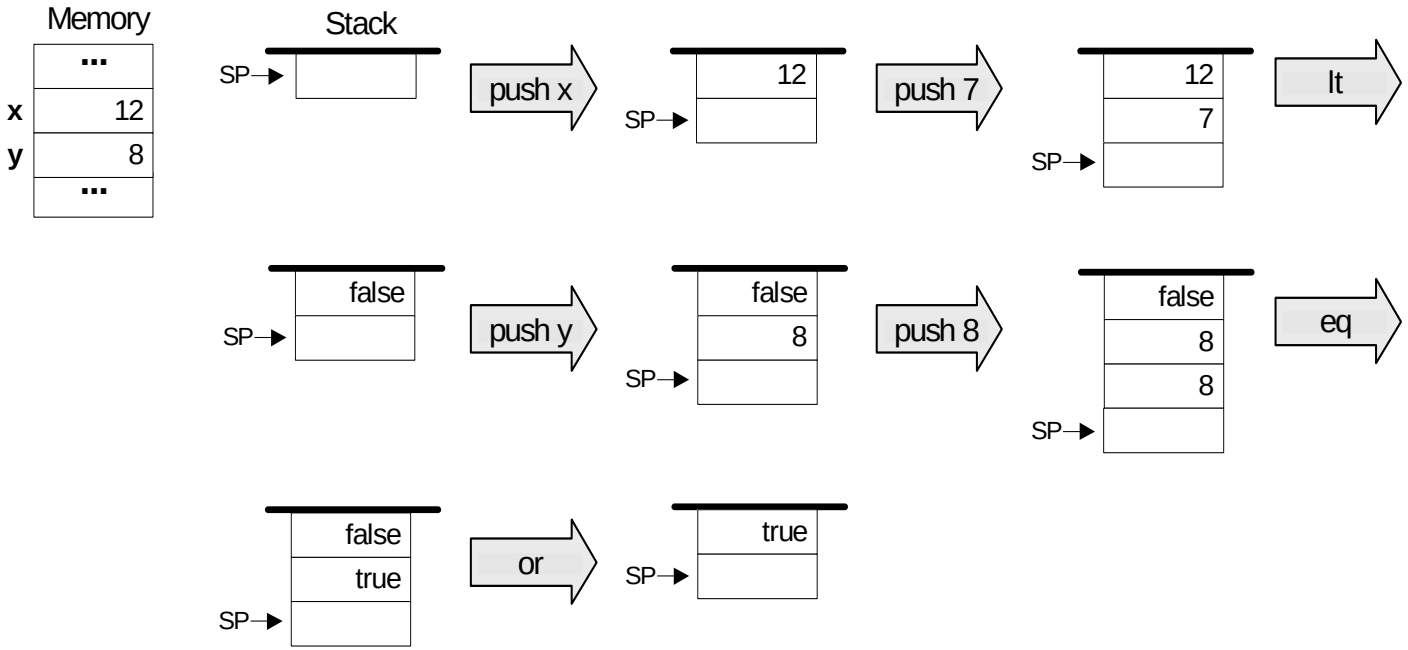
Evaluation of arithmetic expressions

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



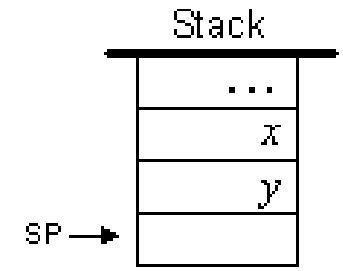
Evaluation of Boolean expressions

```
// 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 access (motivation)

Modern programming languages normally feature the following variable kinds:

- Class level
  - Static variables
  - Private variables (AKA "object variabls" / "fields" / "properties")
- Method level:
  - Local variables
  - Argument variables

A VM abstraction must support (at least) all these variable kinds.

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

Memory access commands

Command format:

pop *segment i*

push *segment i*

(Rather than **pop** *x* and **push** *y*,
as was shown in previous slides,
which was a conceptual simplification)

Where *i* is a non-negative integer and *segment* is one of the following:

- **static**: holds values of global variables, shared by all functions in the same class
- **argument**: holds values of the argument variables of the current function
- **local**: holds values of the local variables of the current function
- **this**: holds values of the private ("object") variables of the current object
- **that**: holds array values
- **constant**: holds all the constants in the range 0…32767 (pseudo memory segment)
- **pointer**: used to align **this** and **that** with different areas in the heap
- **temp**: fixed 8-entry segment that holds temporary variables for general use; Shared by all VM functions in the program.

VM programming

- VM programs are normally written by *compilers*, not by humans
- In order to write compilers, it helps to understand the spirit of VM programming. So we will now see how some common programming tasks can be implemented in the VM abstraction:
  - Arithmetic task
  - Object handling task
  - Array handling task

Disclaimer:

These programming examples don't belong here; They belong to the compiler chapter, since expressing programming tasks in the VM language is the business of the compiler (e.g., translating Java programs to Bytecode programs)

We discuss them here to give some flavor of programming at the VM level.

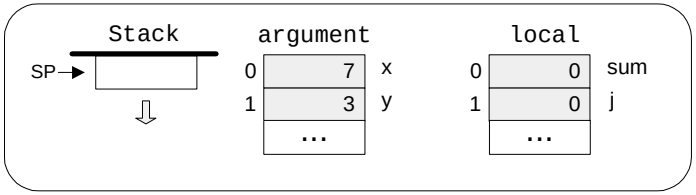
(One can safely skip from here to slide 21)

Arithmetic example

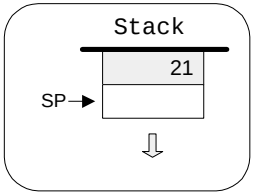
High-level code

```
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:



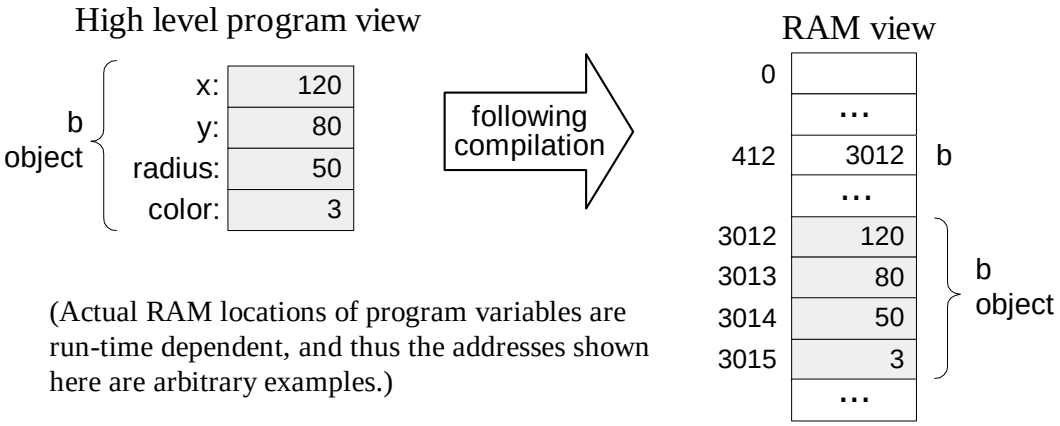
VM code (first approx.)

```
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
```

Object handling example

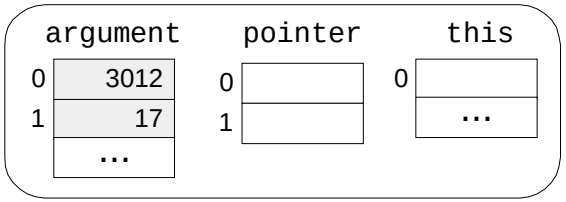


(Actual RAM locations of program variables are run-time dependent, and thus the addresses shown here are arbitrary examples.)

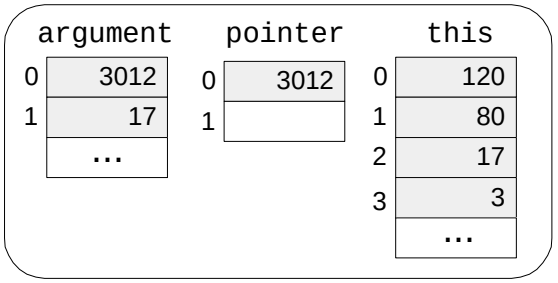
```
/* Assume that b and r were
   passed to the function as
   its first two arguments.
   The following code
   implements the operation
   b.radius=r. */

// Get b's base address:
push argument 0
// Point the this seg. to b:
pop pointer 0
// Get r's value
push argument 1
// Set b's third field to r:
pop this 2
```

Virtual memory segments just before the operation `b.radius=17`:

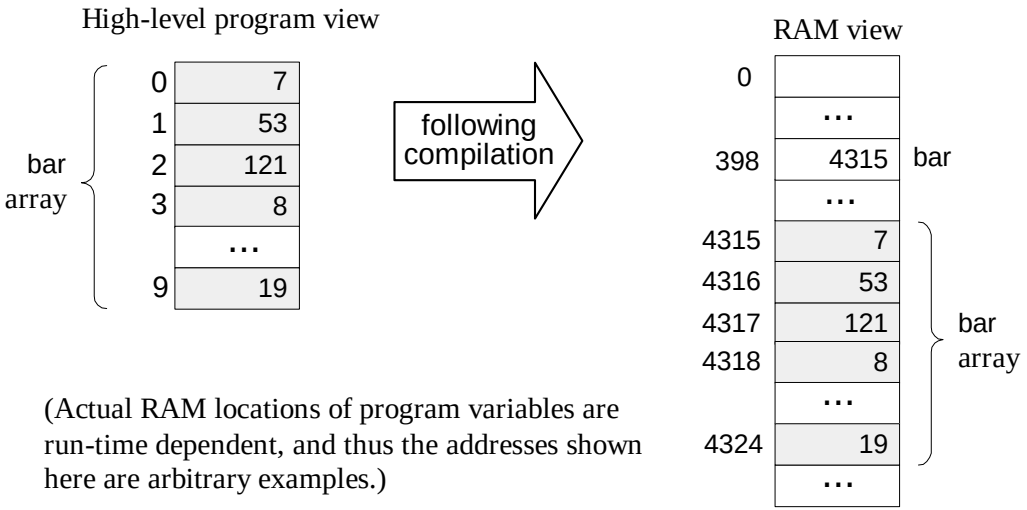


Virtual memory segments just after the operation `b.radius=17`:



(`this 0` is now alligned with `RAM[3012]`)

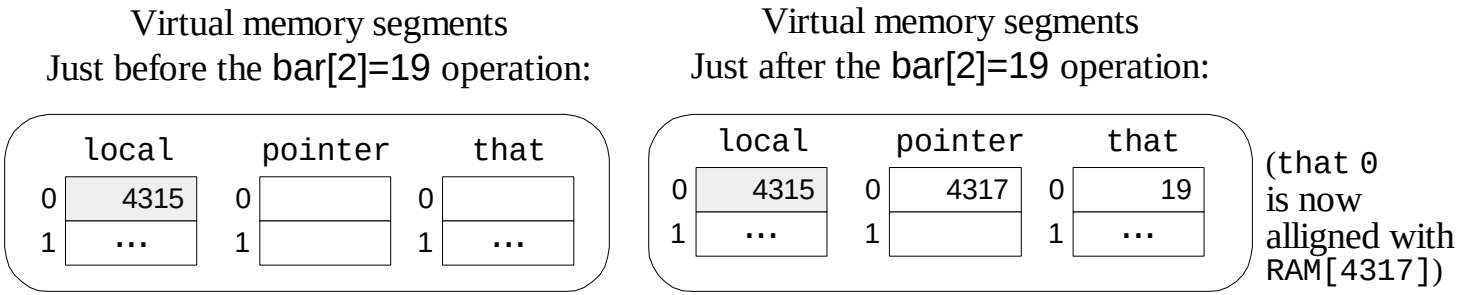
Array handling example



(Actual RAM locations of program variables are run-time dependent, and thus the addresses shown here are arbitrary examples.)

```
/* Assume that bar is the
first local variable declared
in the high-level program. The
code below implements
bar[2]=19, or *(bar+2)=19. */

// Get bar's base address:
push local 0
push constant 2
add
// Set that's base to (bar+2):
pop  pointer 1
push constant 19
// *(bar+2)=19:
pop  that 0
```



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

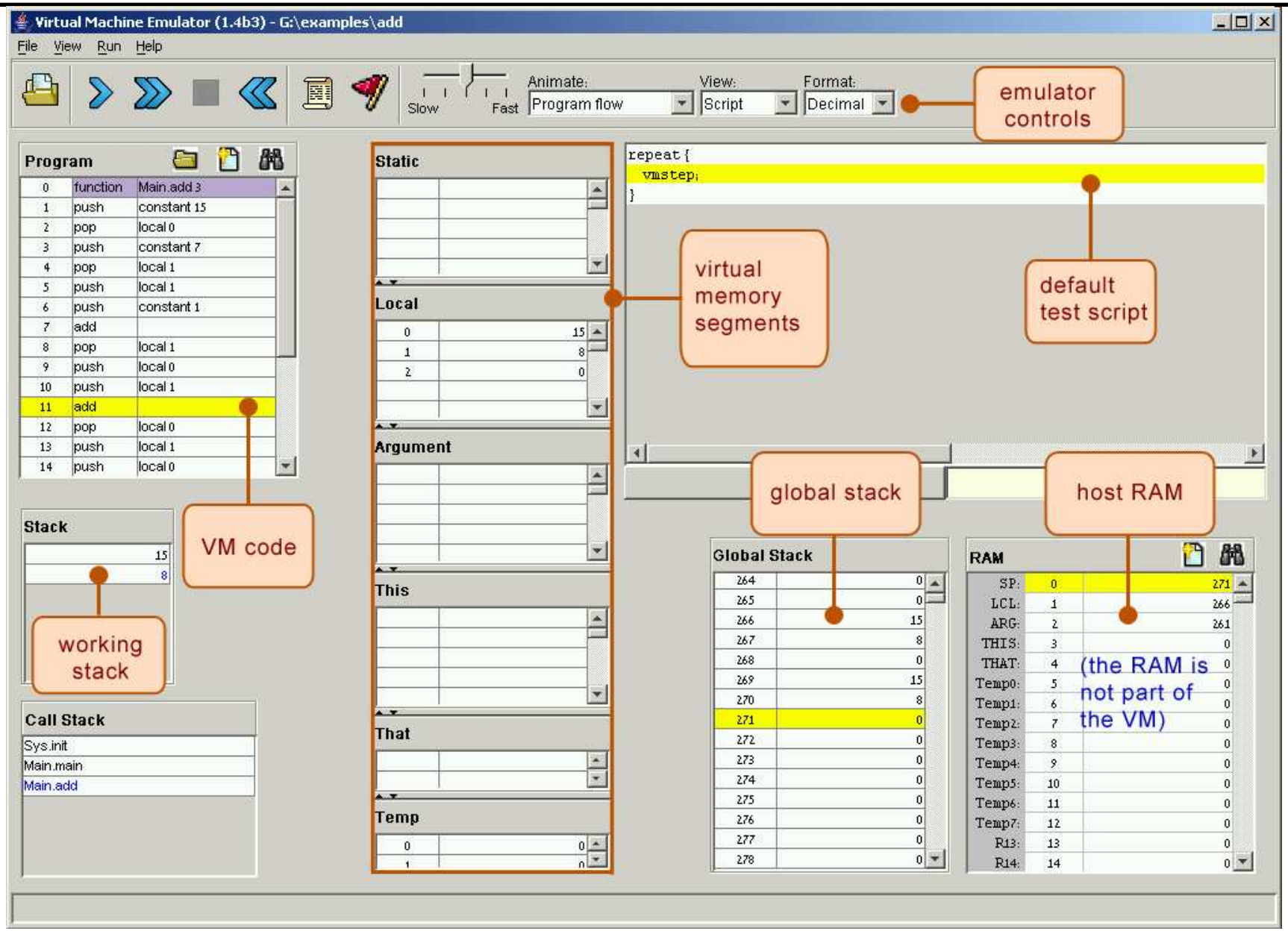
VM implementation options:

- Software-based (emulation)
- Translator-based (e.g., to the Hack language)
- Hardware-based (CPU-level)

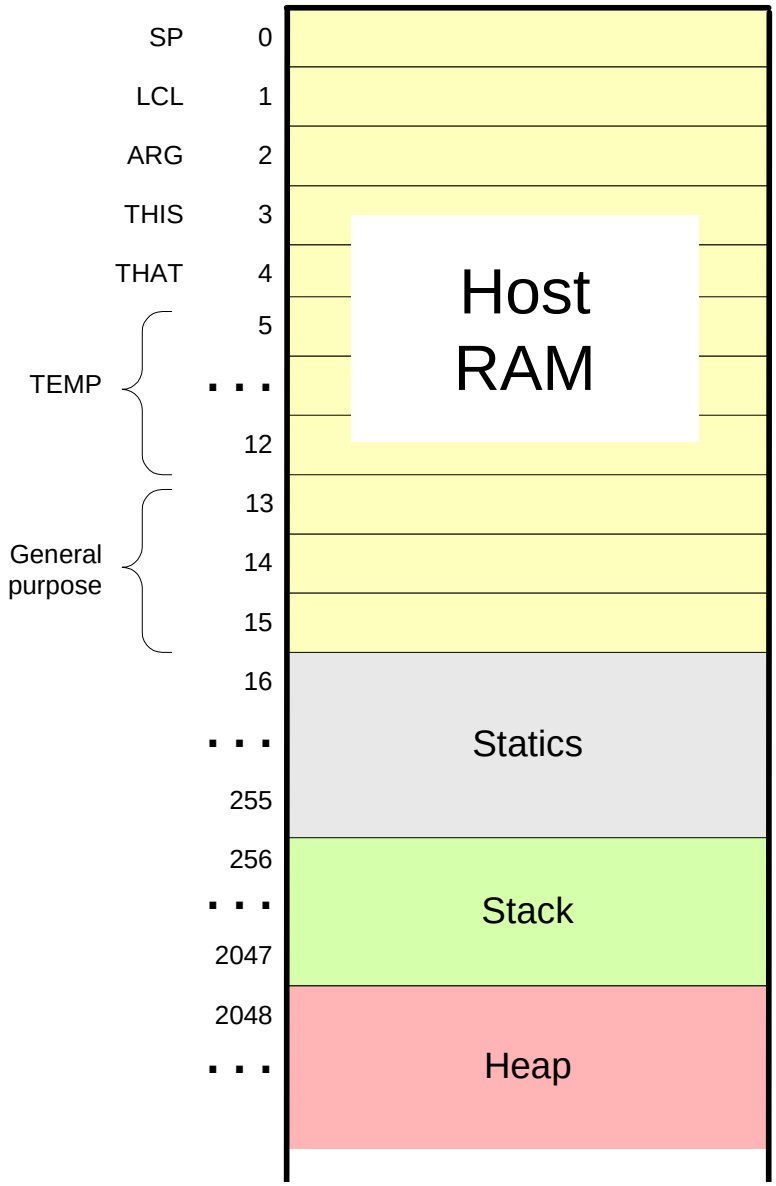
Well-known translator-based implementations:

- 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)



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, temp:`** see the book

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

Parser module (proposed design)

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`.

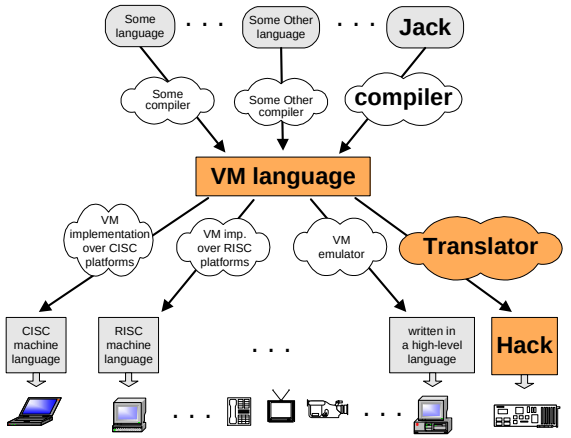
CodeWriter module (proposed design)

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 (implements the VM language on a target 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:

- Complete the VM specification and implementation (chapters 7,8)
- Introduce Jack, a high-level programming language (chapter 9)
- Build a compiler for it (chapters 10,11)
- Finally, build a mini-OS, i.e. a run-time library (chapter 12).

Conceptually similar to:

- JVM
- Java
- Java compiler
- JRE

And to:

- CLR
- C#
- C# compiler
- .NET base class library