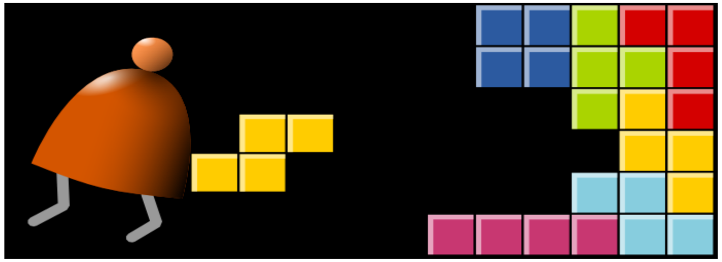
Virtual Machine

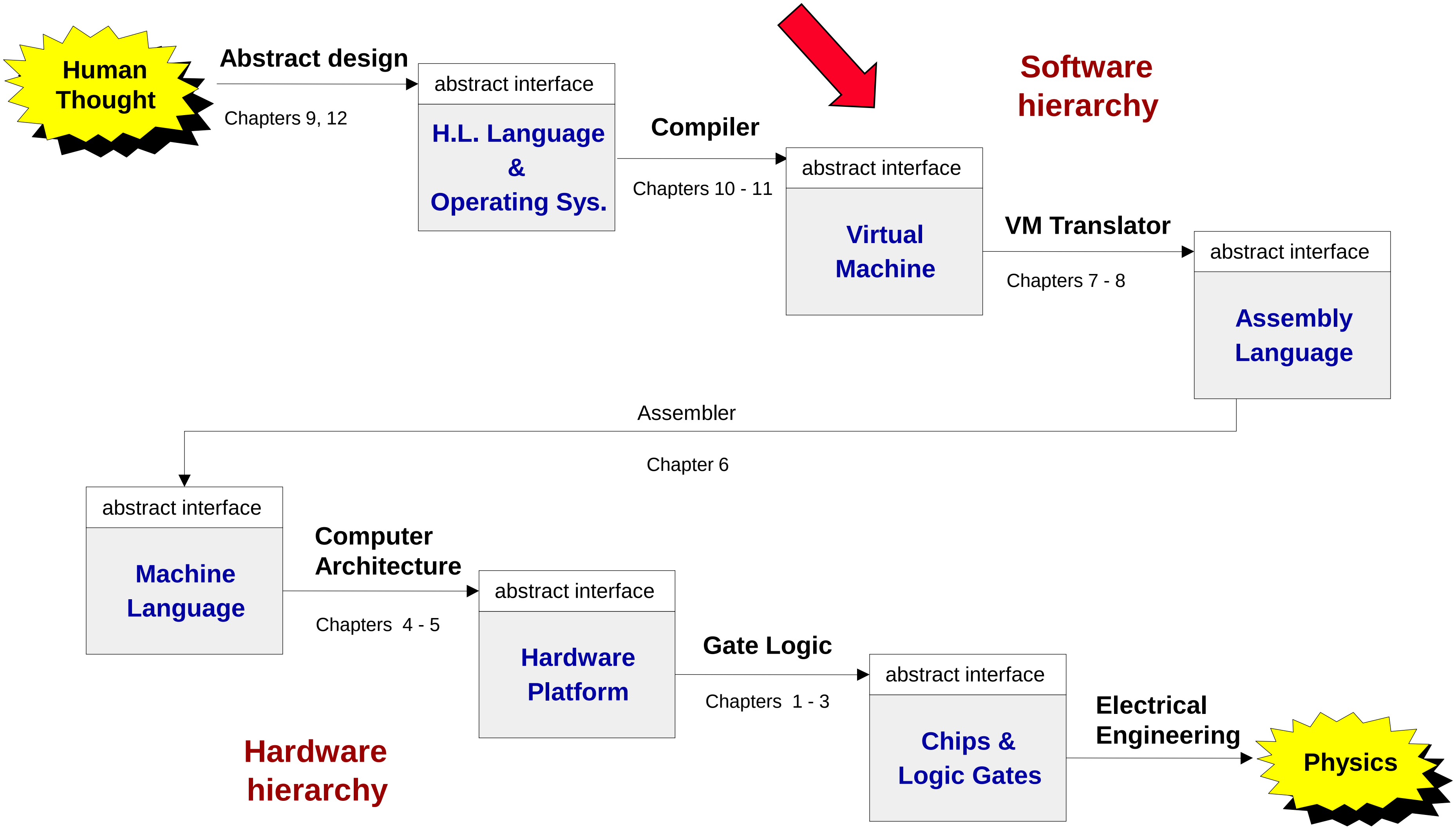
Part I: Stack Arithmetic



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



Human Thought

Abstract design

Chapters 9, 12

abstract interface

H.L. Language & Operating Sys.

Compiler

Chapters 10 - 11

abstract interface

Virtual Machine

Software hierarchy

VM Translator

Chapters 7 - 8

abstract interface

Assembly Language

Assembler

Chapter 6

abstract interface

Machine Language

Computer Architecture

Chapters 4 - 5

abstract interface

Hardware Platform

Gate Logic

Chapters 1 - 3

abstract interface

Chips & Logic Gates

Electrical Engineering

Physics

Hardware hierarchy

Motivation

Jack code (example)

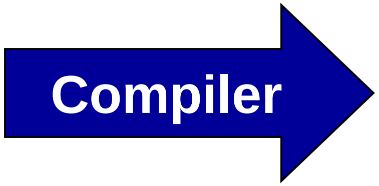
```
class Main {
  static int x;

  function void main() {
    // Inputs and multiplies two 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 ~(j = 0) {
      let result = result + x;
      let j = j – 1;
    }
    return result;
  }
}
```

Our ultimate goal:

Translate high-level programs into executable code.

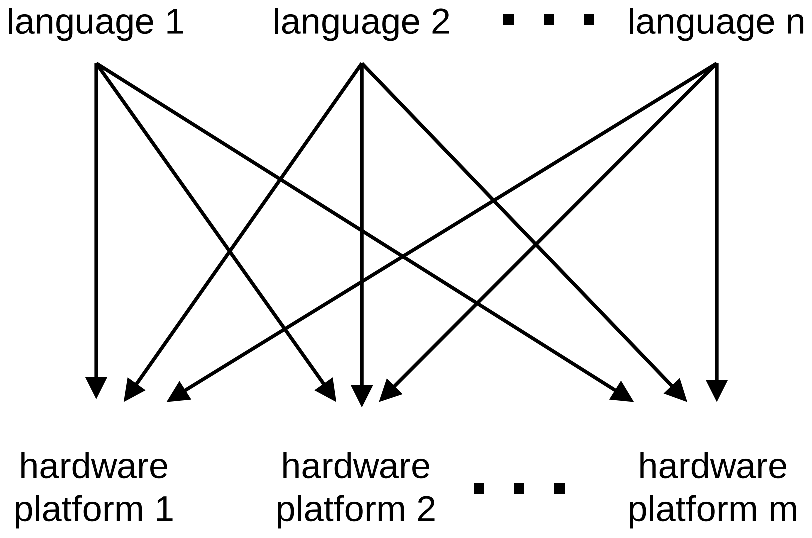


Hack code

```
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
...
```

Compilation models

direct compilation:

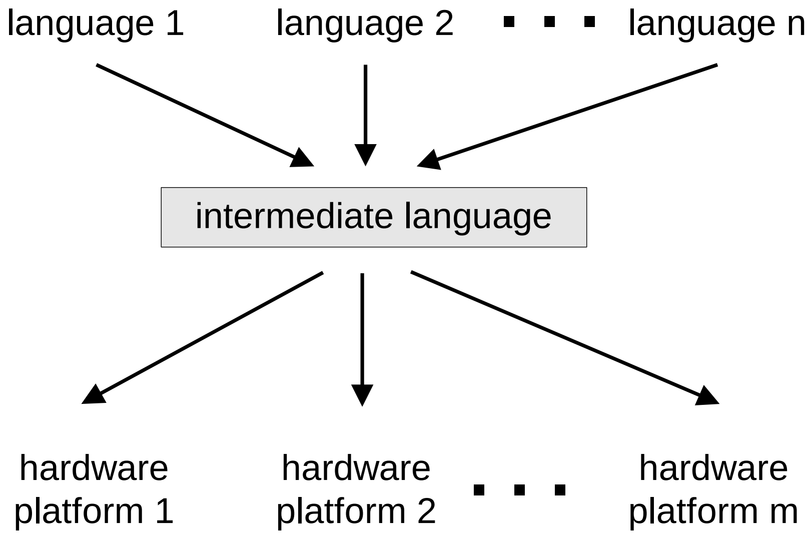


language 1 language 2 ▪ ▪ ▪ language n

hardware platform 1 hardware platform 2 ▪ ▪ ▪ hardware platform m

requires *n*▪ *m* translators

2-tier compilation:



language 1 language 2 ▪ ▪ ▪ language n

intermediate language

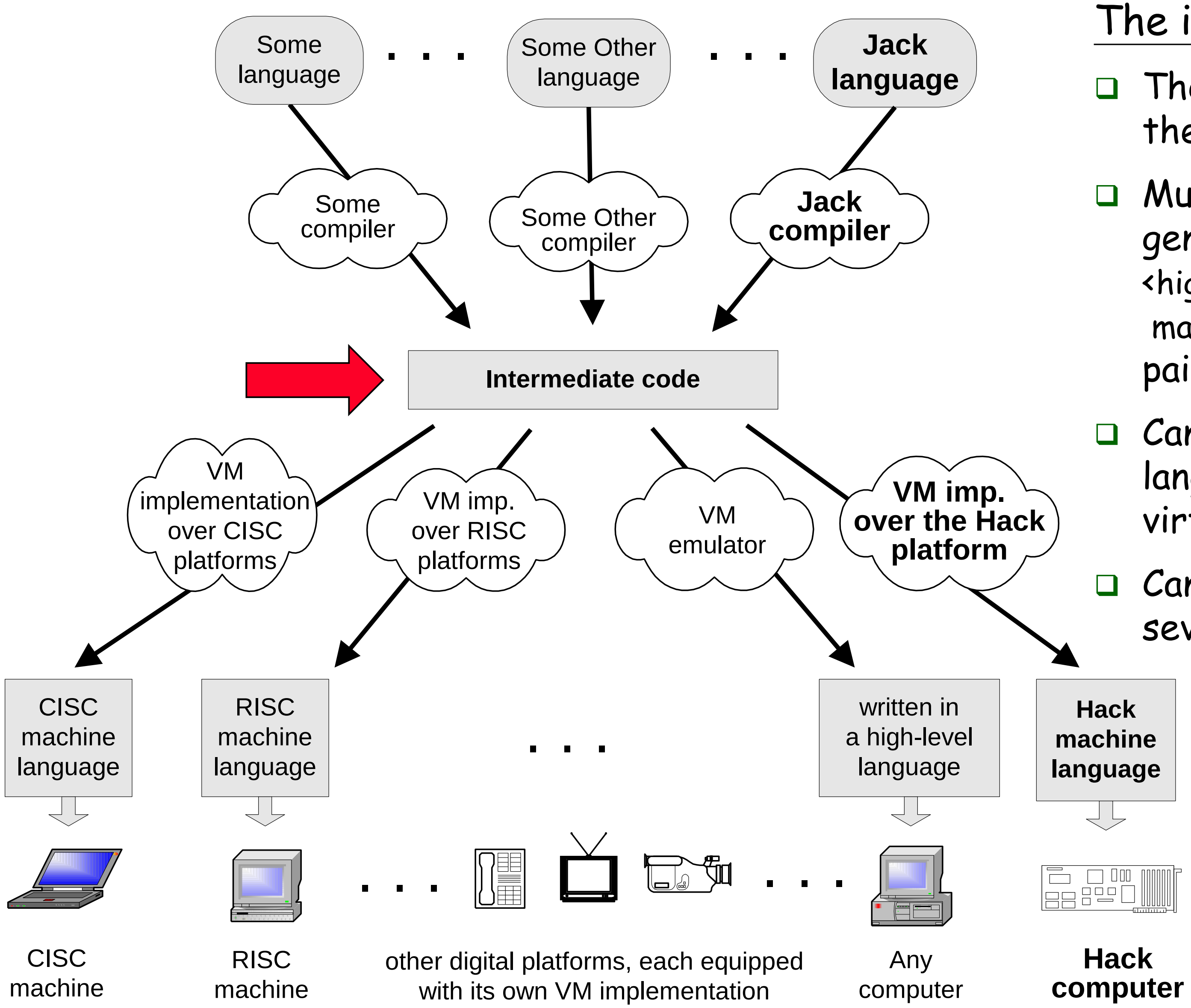
hardware platform 1 hardware platform 2 ▪ ▪ ▪ hardware platform m

requires *n* + *m* translators

Two-tier compilation:

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

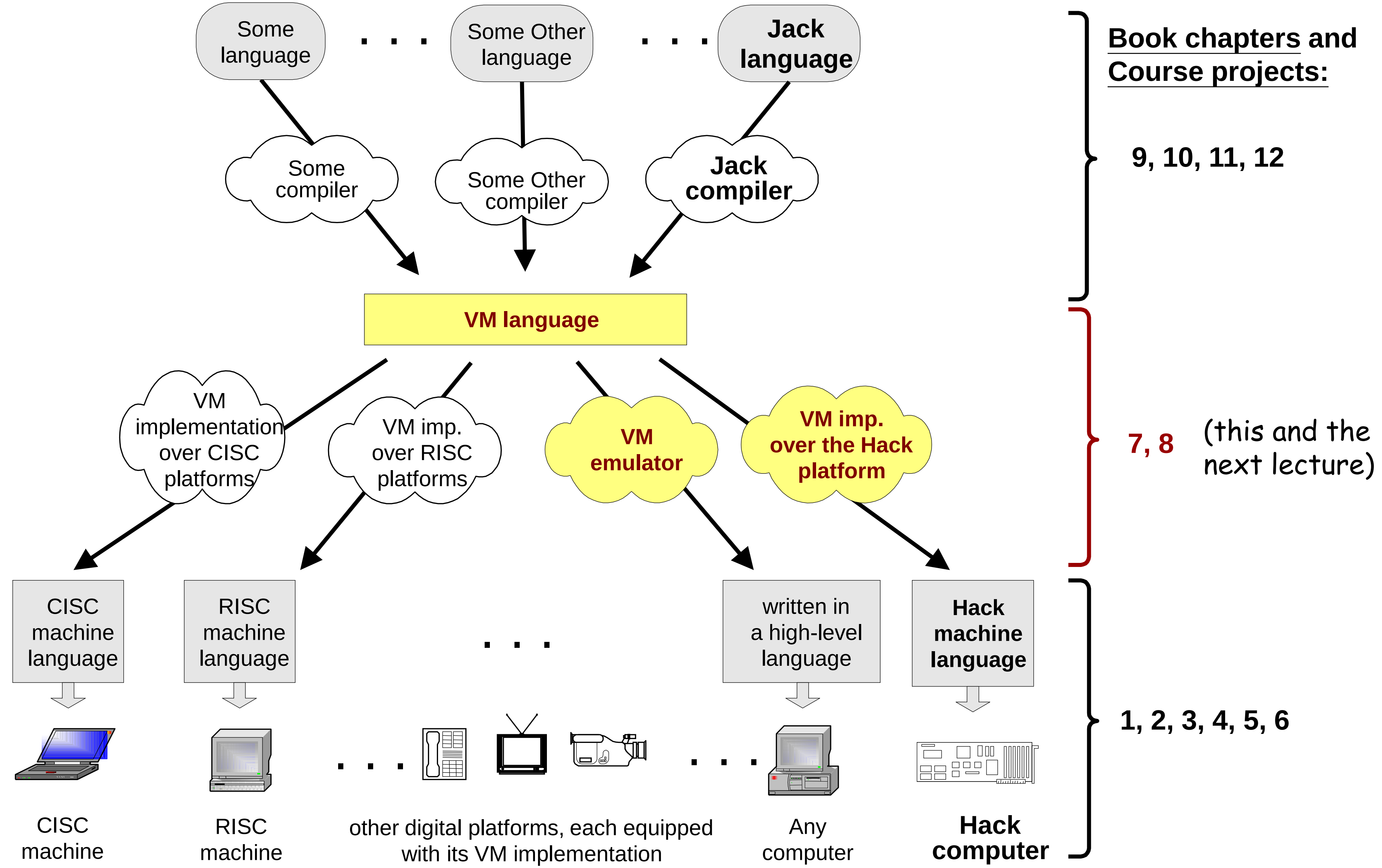
The big picture



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 several different ways.

Focus of this lecture (yellow):



Some language . . . Some Other language . . . Jack language

Some compiler

Some Other compiler

Jack compiler

VM language

VM implementation over CISC platforms

VM imp. over RISC platforms

VM emulator

VM imp. over the Hack platform

CISC machine language

RISC machine language

. . .

written in a high-level language

Hack machine language

CISC machine

RISC machine

. . . other digital platforms, each equipped with its VM implementation . . .

Any computer

Hack computer

Book chapters and Course projects:

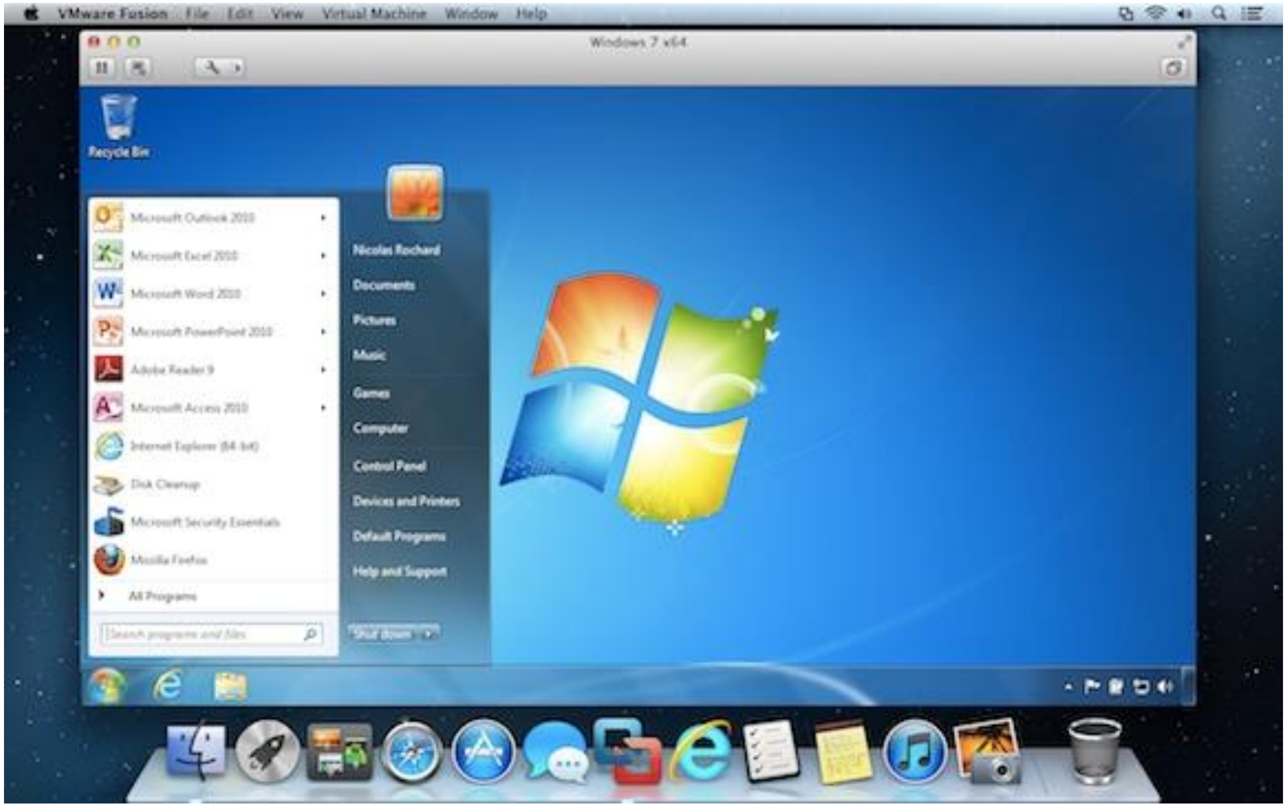
9, 10, 11, 12

7, 8 (this and the next lecture)

1, 2, 3, 4, 5, 6

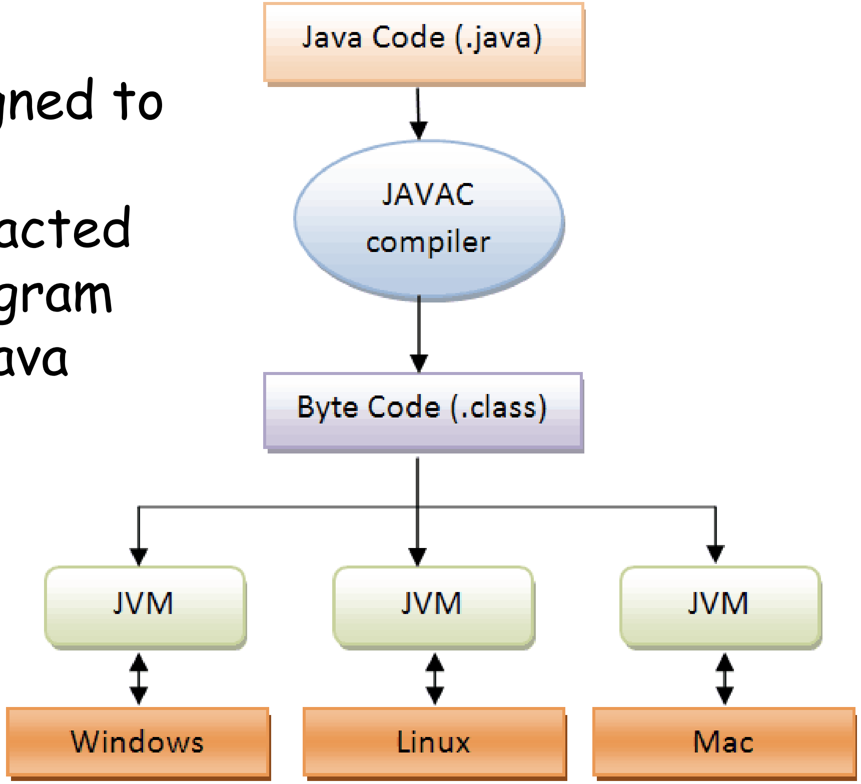
Virtual machines

- A **virtual machine (VM)** is an emulation of a particular (real or hypothetical) computer system.
  - System virtual machine (full virtualization VMs): a complete substitute for the targeted real machine and a level of functionality required for the execution of a complete operating system, e.g., VirtualBox.



Virtual machines

- A **virtual machine (VM)** is an emulation of a particular (real or hypothetical) computer system.
  - System virtual machine (full virtualization VMs): a complete substitute for the targeted real machine and a level of functionality required for the execution of a complete operating system, e.g., VirtualBox.
  - Process virtual machine: designed to execute a single computer program by providing an abstracted and platform-independent program execution environment, e.g., Java virtual machine (JVM).



The VM model and language

Perspective:

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

Other VM models (like Java's JVM/JRE and .NET's IL/CLR) are similar in spirit, but differ in scope and details.

Hack virtual machine

Goal: Specify and implement a VM model and language:

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)

Chapter 8

Function calling commands

`function` (declaration)

`call` (a function)

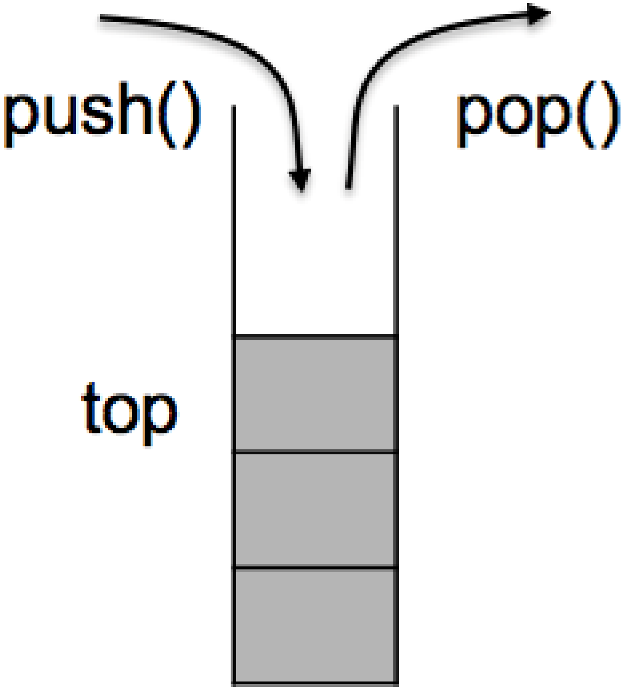
`return` (from a function)

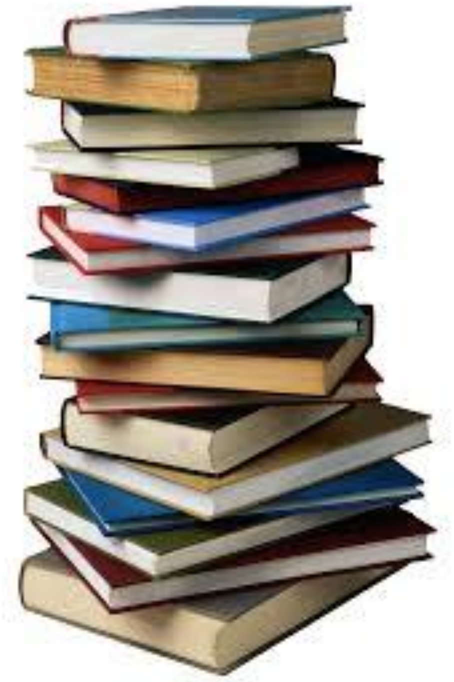
Our game plan: (a) describe the VM abstraction (3 types of instructions)
(b) propose how to implement it over the Hack platform.

The stack

The stack:

- A classical LIFO data structure
- Elegant and powerful
- Several hardware / software implementation options.

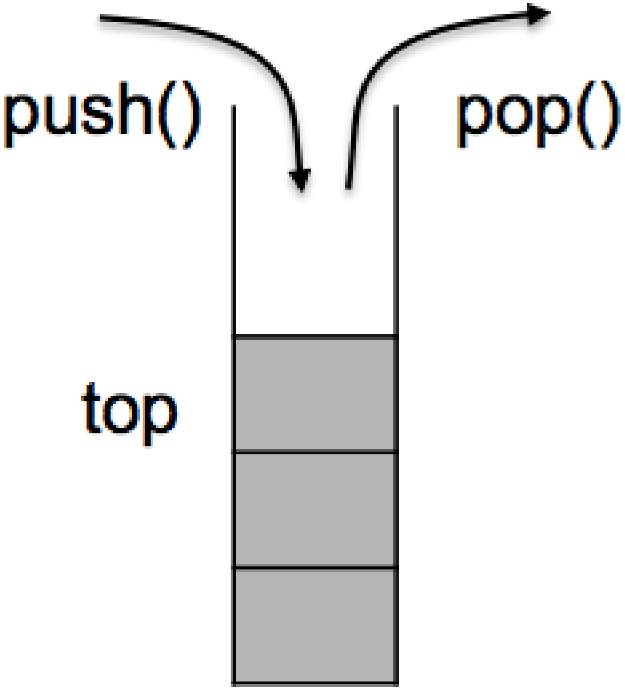




The stack

The stack:

- A classical LIFO data structure
- Elegant and powerful
- Several hardware / software implementation options.
- Several flavors: next empty/valid, increase/decrease



```
push(x)
    stack[top]=x;
    top++;

pop()
    top--;
    return stack[top];

peek(), empty()
```

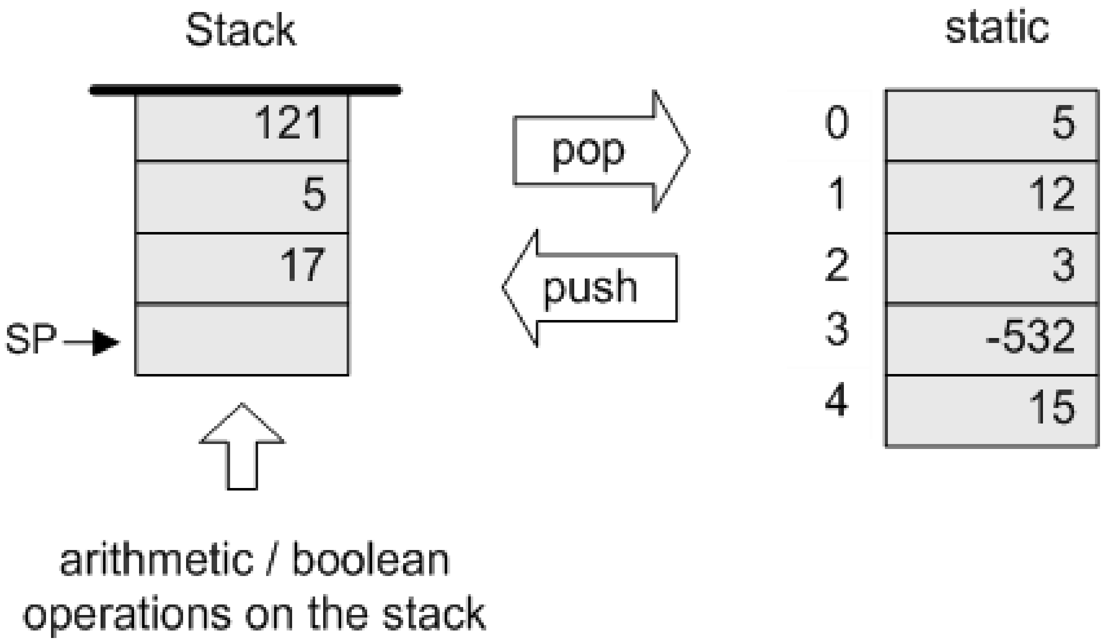
What is the stack good for?

- Stack can be used for evaluating arithmetic expressions
- Expression: 5 * (6+2) – 12/4
  - Infix
  - Prefix
  - Postfix

Stack is also good for implementing function call structures, such as subroutines, local variables and recursive calls. Will discuss it later.

Our VM model is *stack-oriented*

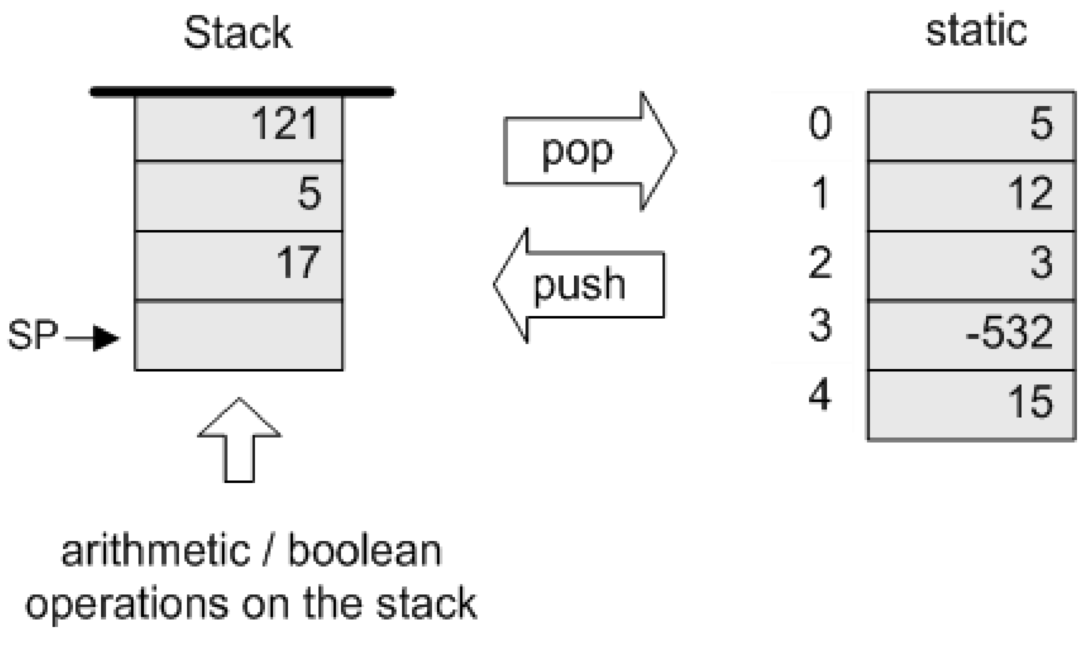
- All operations are done on a stack
- Data is saved in several separate *memory segments*
- All the memory segments behave the same
- One of the memory segments m is called `static`, and we will use it (as an arbitrary example) in the following examples:



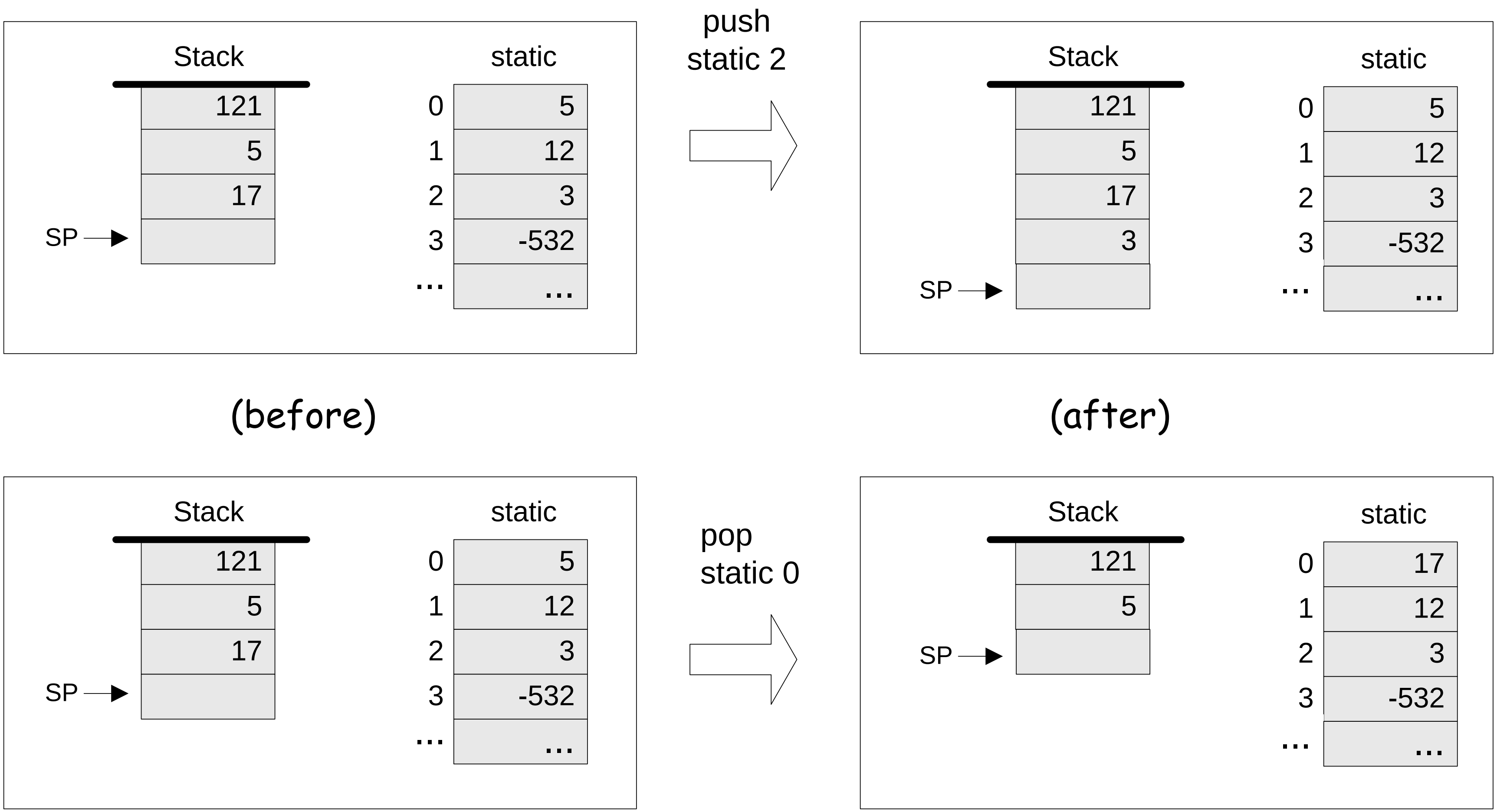
Data types

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

- an integer value (16-bit 2's complement: -32768, ... , 32767)
- a Boolean value (0 and -1, standing for true and false)
- a pointer (memory address)



Memory access operations



(before)

Stack

121
5
17
SP →

static

0	5
1	12
2	3
3	-532
...	...

push
static 2

(after)

Stack

121
5
17
3
SP →

static

0	5
1	12
2	3
3	-532
...	...

Stack

121
5
17
SP →

static

0	5
1	12
2	3
3	-532
...	...

pop
static 0

Stack

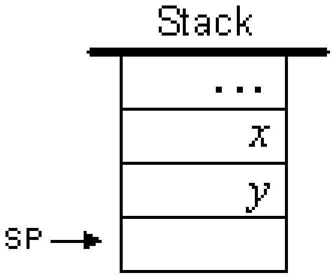
121
5
SP →

static

0	17
1	12
2	3
3	-532
...	...

Arithmetic and Boolean commands in the VM language (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

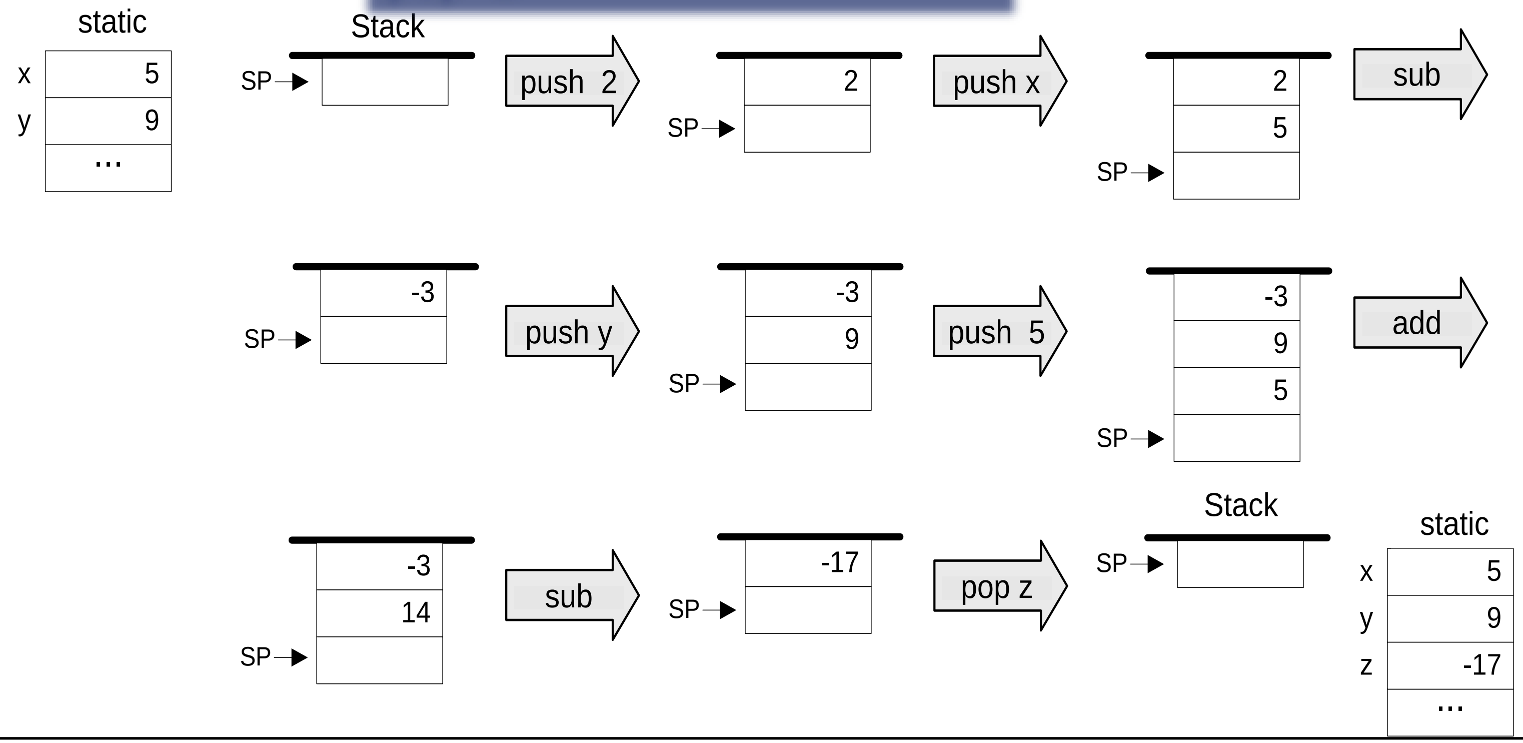


Evaluation of arithmetic expressions

VM code (example)

```
// z=(2-x)-(y+5)
push 2
push x
sub
push y
push 5
add
sub
pop z
```

(suppose that
x refers to static 0,
y refers to static 1,
z refers to static 2)

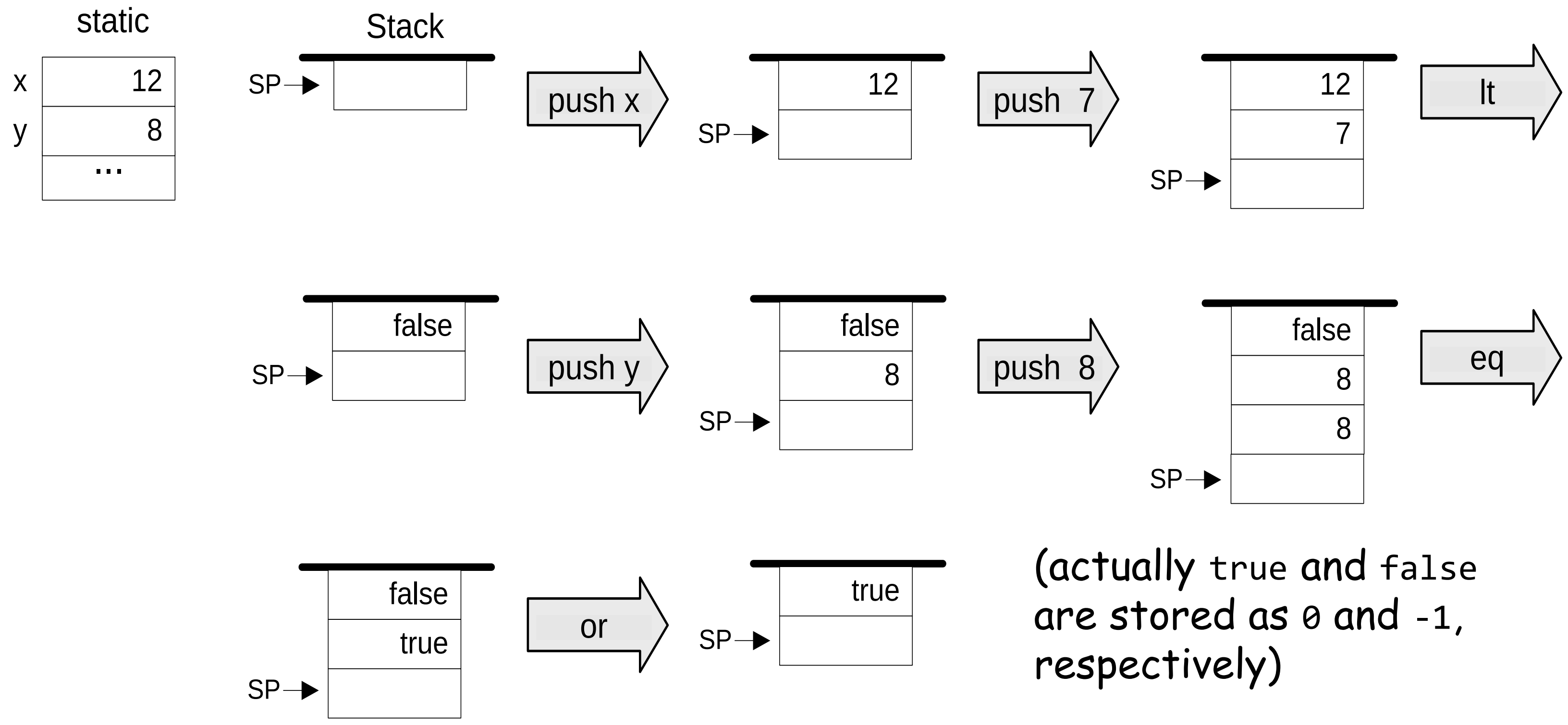


Evaluation of Boolean expressions

VM code (example)

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

(suppose that
x refers to static 0,
y refers to static 1)



static

x 12

y 8

...

Stack

SP → (empty)

push x

12

SP →

push 7

12

7

SP →

lt

false

SP →

push y

false

8

SP →

push 8

false

8

8

SP →

eq

false

true

SP →

or

true

SP →

(actually true and false are stored as 0 and -1, respectively)

The VM’s Memory segments

A VM program is designed to provide an interim abstraction of a program written in some high-level language.

Modern OO languages normally feature the following variable kinds:

Class level:

- Static variables (class-level variables)
- Private variables (aka “object variables” / “fields” / “properties”)

Method level:

- Local variables
- Argument variables

When translated into the VM language,

The static, private, local and argument variables are mapped by the compiler on the four memory segments `static`, `this`, `local`, `argument`

In addition, there are four additional memory segments, whose role will be presented later: `that`, `constant`, `pointer`, `temp`.

Memory segments and memory access commands

The VM abstraction includes 8 separate memory segments named:
`static`, `this`, `local`, `argument`, `that`, `constant`, `pointer`, `temp`

As far as VM programming commands go, all memory segments look and behave the same

To access a particular segment entry, use the following generic syntax:

Memory access VM commands:

- pop *memorySegment index*
- push *memorySegment index*

Where *memorySegment* is `static`, `this`, `local`, `argument`, `that`, `constant`, `pointer`, or `temp`

And *index* is a non-negative integer

(In all our code examples thus far, *memorySegment* was `static`)

The different roles of the eight memory segments will become relevant when we'll talk about the compiler

At the VM abstraction level, all memory segments are treated the same way.

VM programming

VM programs are normally written by *compilers*, not by humans

However, compilers are written by humans ...

In order to write or optimize a compiler, it helps to first understand the spirit of the compiler's target language – the VM language

So, we'll now see an example of a VM program

VM programming

The example includes three new VM commands:

- `function` *functionSymbol* // function declaration
- `label` *labelSymbol* // label declaration
- `if-goto` *labelSymbol* // pop x
  // if x=true, jump to execute the
  // command after *labelSymbol*
  // else proceed to execute the next
  // command in the program

For example, to effect `if (x > n) goto loop`, we can use the following VM commands:

```
push x
push n
gt
if-goto loop      // Note that x, n, and the truth value
                  // were removed from the stack.
```

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;
}
```

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;
}
```

Pseudo code

```
...
loop:
   if (j=0) goto end
   result=result+x
   j=j-1
   goto loop
end:
...
```

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;
}
```

Pseudo code

```
...
loop:
   if (j=0) goto end
   result=result+x
   j=j-1
   goto loop
end:
...
```

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
```

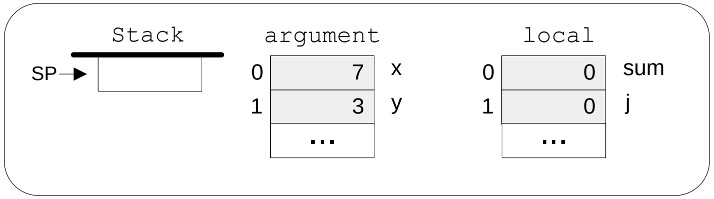
VM code

```
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
```

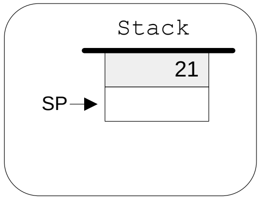
High-level code

```
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

Summary: Hack VM has the following instructions and eight memory segments.

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)

Chapter 8

Function calling commands

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

Method: (a) specify the abstraction (stack, memory segments, commands)

(b) how to implement the abstraction over the Hack platform.

Implementation

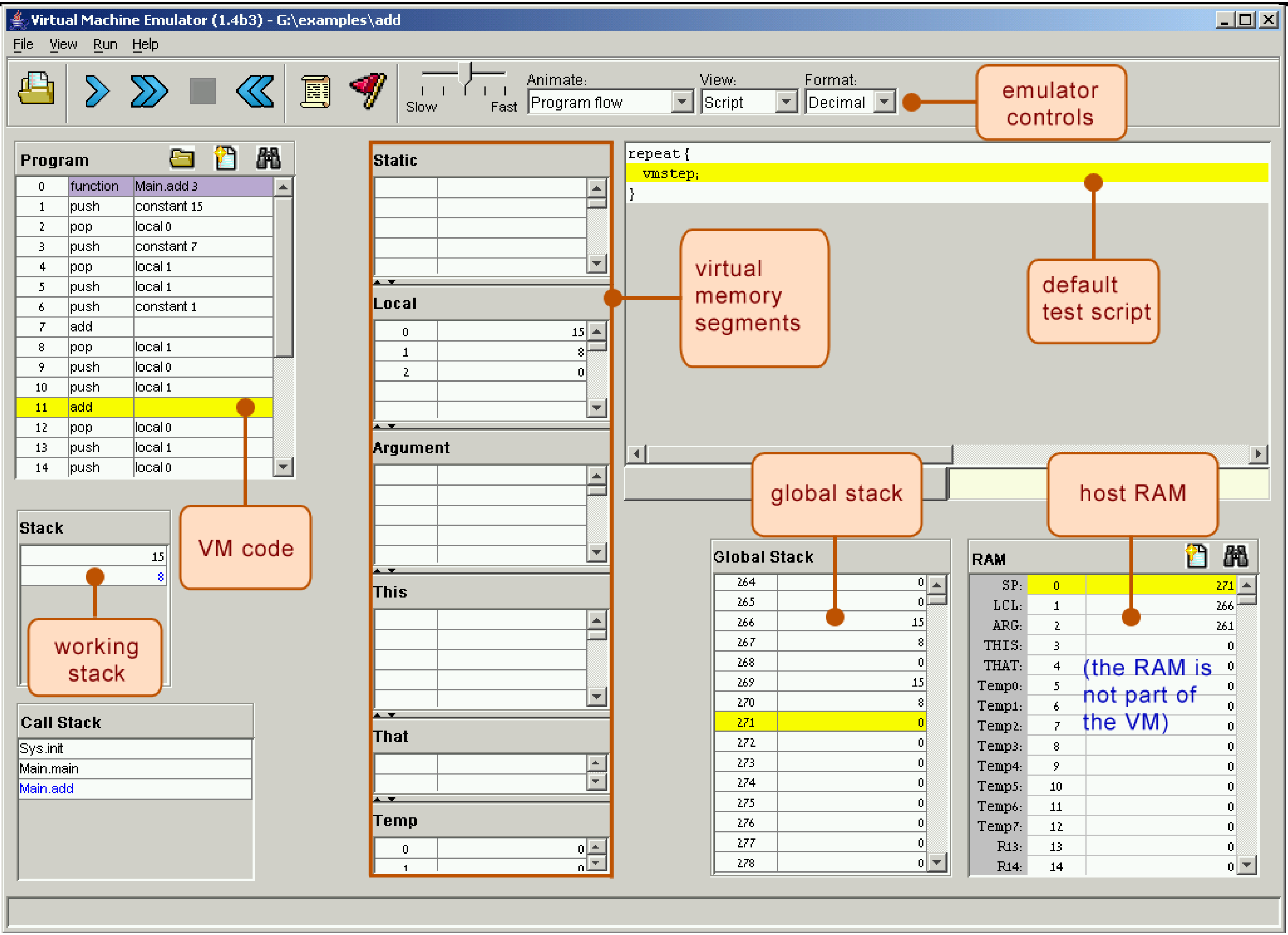
VM implementation options:

- Emulator-based (e.g. emulate the VM model using Java)
- Translator-based (e. g. translate VM programs into the Hack machine language)
- Hardware-based (realize the VM model using dedicated memory and registers)

Implementation of VM on Hack

- Each VM instruction must be translated into a set of Hack assembly code
- VM segments need to be realized on the host memory

Software implementation: VM emulator (part of the course software suite)



Animate: Program flow

View: Script

Format: Decimal

emulator controls

Program

0	function	Main.add 3
1	push	constant 15
2	pop	local 0
3	push	constant 7
4	pop	local 1
5	push	local 1
6	push	constant 1
7	add	
8	pop	local 1
9	push	local 0
10	push	local 1
11	add	
12	pop	local 0
13	push	local 1
14	push	local 0

VM code

Stack

15
8

working stack

Call Stack

Sys.init
Main.main
Main.add

Static

Local

0	15
1	8
2	0

Argument

This

That

Temp

0	0
1	0

virtual memory segments

```
repeat{
  vmstep;
}
```

default test script

global stack

Global Stack

264	0
265	0
266	15
267	8
268	0
269	15
270	8
271	0
272	0
273	0
274	0
275	0
276	0
277	0
278	0

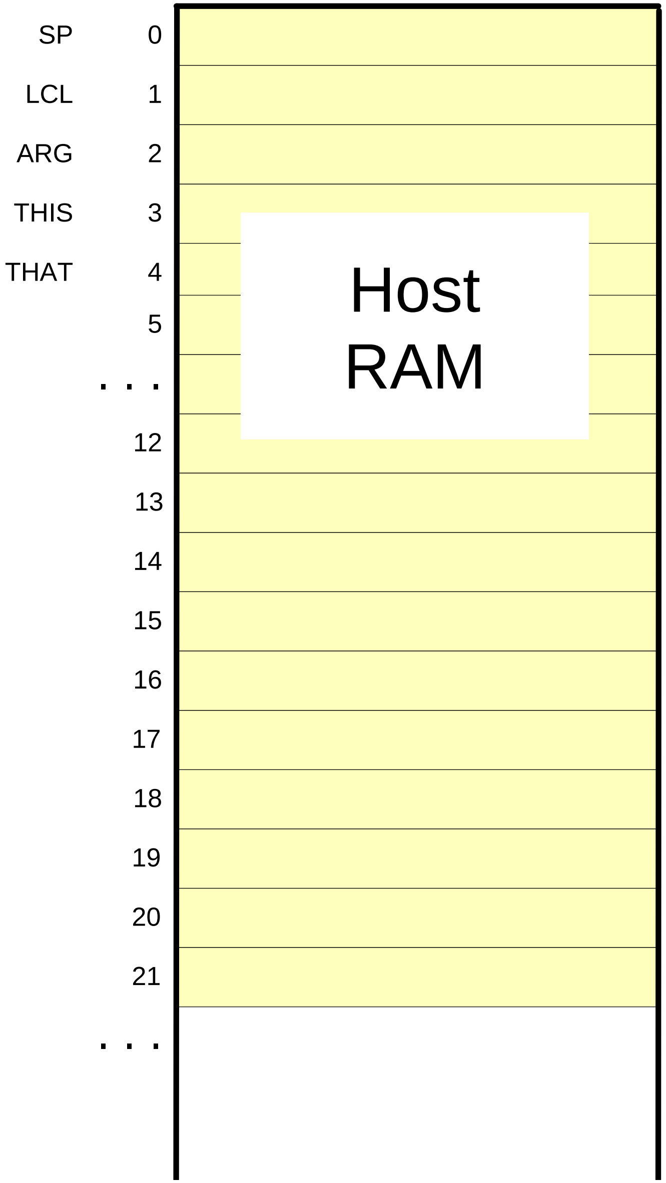
host RAM

RAM

SP:	0	271
LCL:	1	266
ARG:	2	261
THIS:	3	0
THAT:	4	0
Temp0:	5	0
Temp1:	6	0
Temp2:	7	0
Temp3:	8	0
Temp4:	9	0
Temp5:	10	0
Temp6:	11	0
Temp7:	12	0
R13:	13	0
R14:	14	0

(the RAM is not part of the VM)

VM implementation on the Hack platform (memory)



The stack: a global data structure, used to save and restore the resources of all the VM functions up the calling hierarchy.

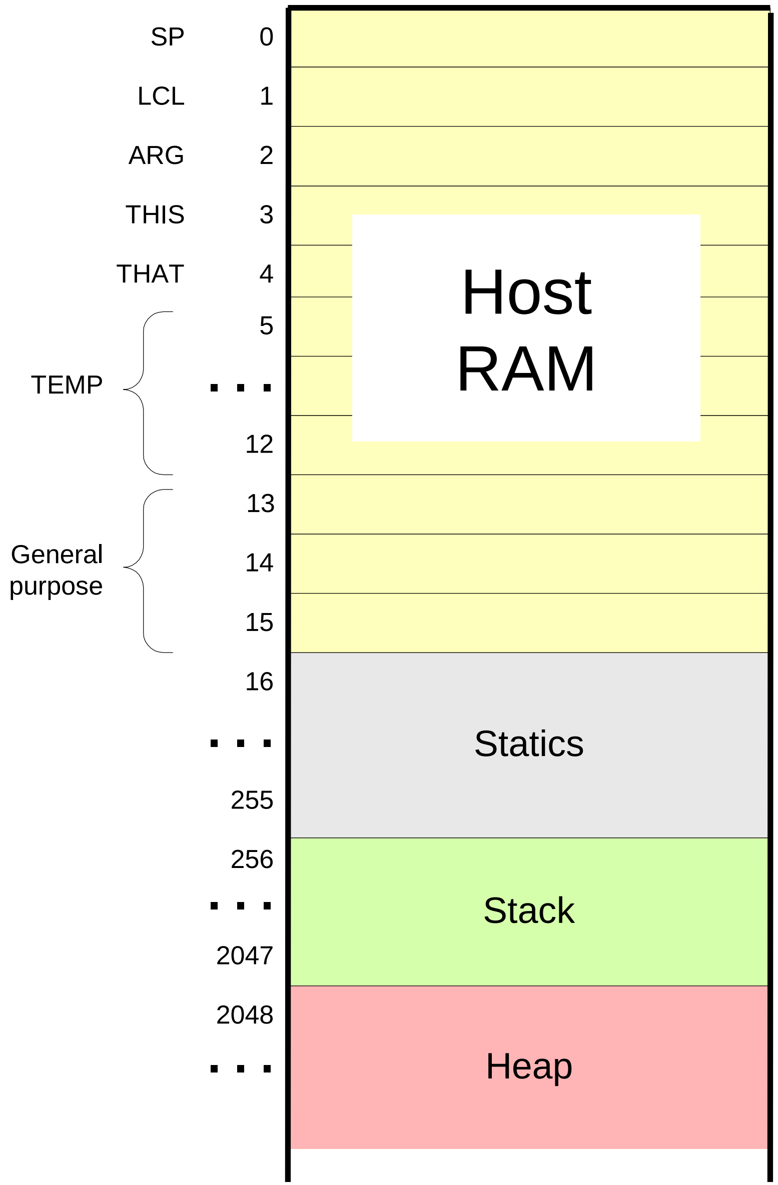
The top of this stack is the working stack of the current function

static, constant, temp: Global memory segments, all functions see the same four segments

local,argument,this,that, pointer: these segments are local at the function level; each function sees its own, private copy of each one of these four segments

The challenge: represent all these logical constructs on the same single physical address space -- the host RAM.

VM implementation on the Hack platform (memory)

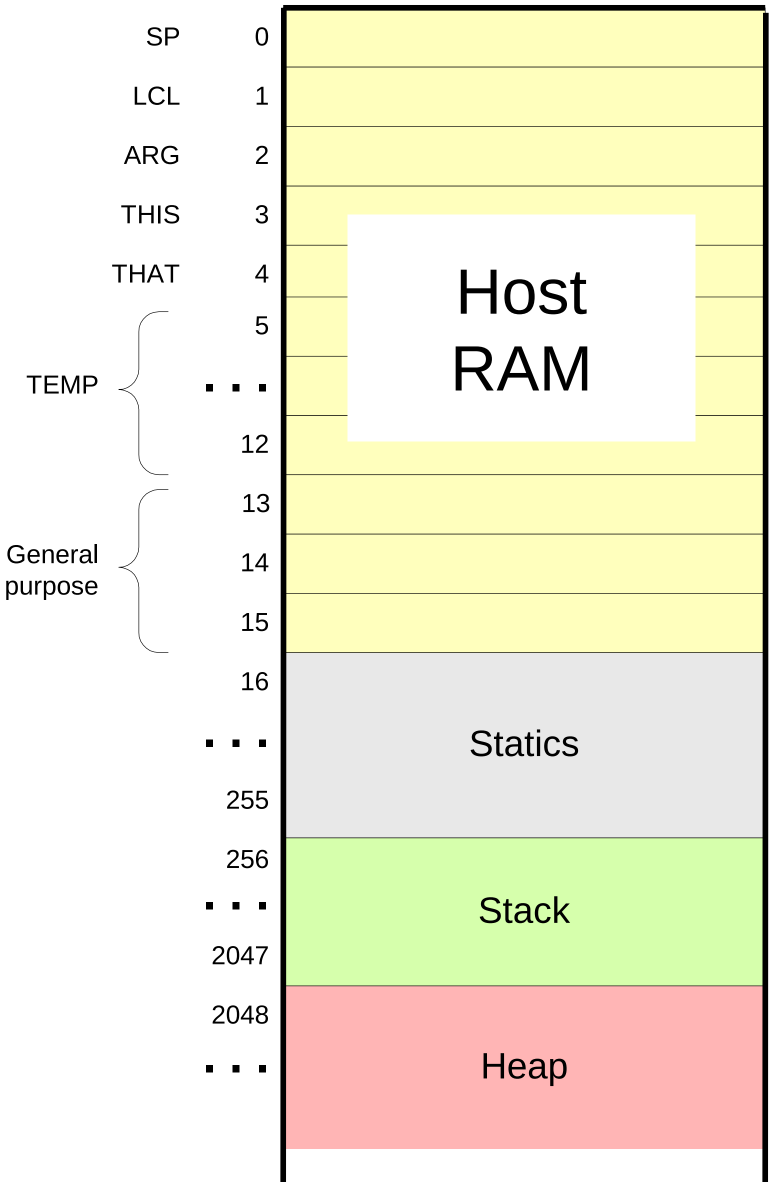


Basic idea: the mapping of the stack and the global segments on the RAM is easy (fixed); the mapping of the function-level segments is dynamic, using pointers

The stack: mapped on RAM[256 .. 2047]; The stack pointer is kept in RAM address SP

static: mapped on RAM[16 ... 255]; each segment reference static *i* appearing in a VM file named f is compiled to the assembly language symbol f.i (recall that the assembler further maps such symbols to the RAM, from address 16 onward)

VM implementation on the Hack platform (memory)



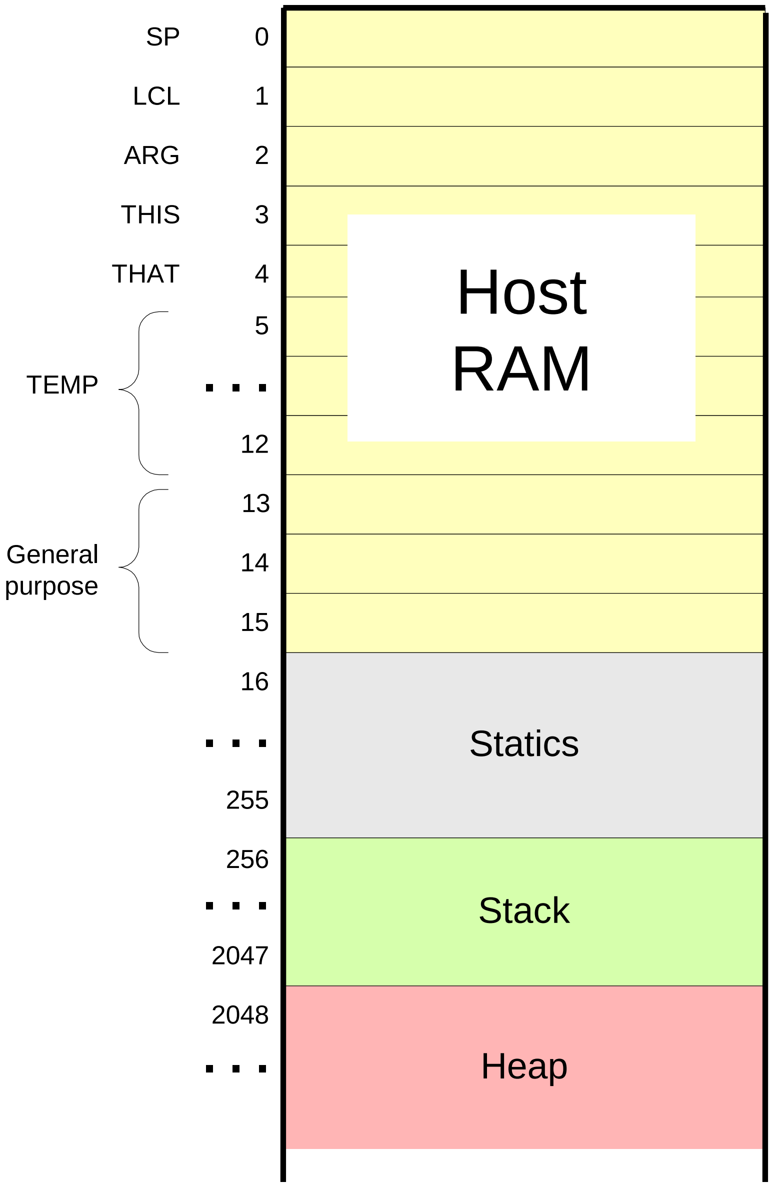
`local,argument`: these method-level segments are stored in the stack, The base addresses of these segments are kept in RAM addresses LCL and ARG. Access to the *i*-th entry of any of these segments is implemented by accessing RAM[segmentBase + *i*]

`this,that`: these dynamically allocated segments are mapped somewhere from address 2048 onward, in an area called "heap". The base addresses of these segments are kept in RAM addresses THIS, and THAT.

`constant`: a truly virtual segment: access to constant *i* is implemented by supplying the constant *i*.

`pointer`: discussed later.

VM implementation on the Hack platform (memory)



Practice exercises

Now that we know how the memory segments are mapped on the host `RAM`, we can write Hack commands that realize the various VM commands. for example, let us write the Hack code that implements the following VM commands:

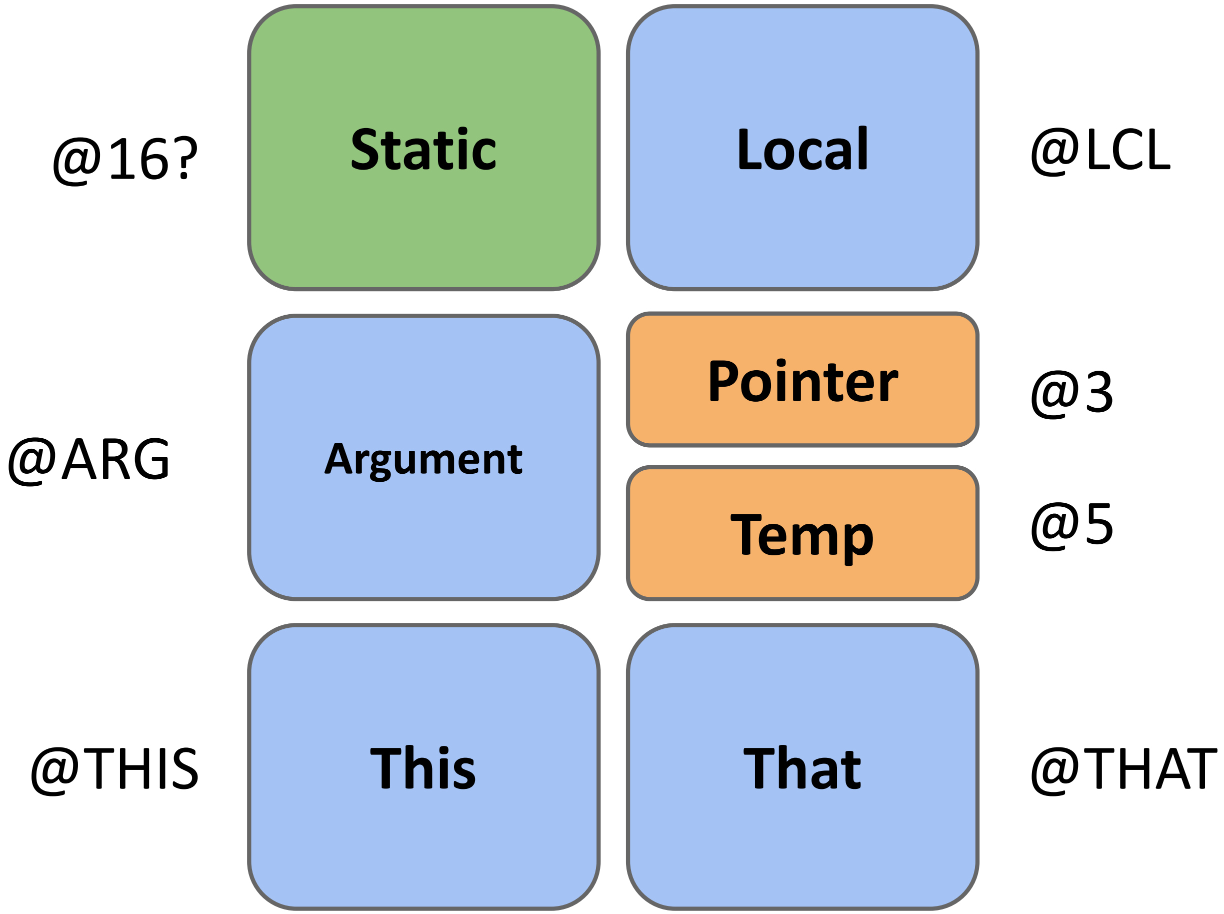
- ❑ `push constant 1`
- ❑ `pop static 7` (suppose it appears in a VM file named `f`)
- ❑ `push constant 5`
- ❑ `add`
- ❑ `pop local 2`
- ❑ `eq`

Tips:

1. The implementation of any one of these VM commands requires several Hack assembly commands involving pointer arithmetic (using commands like `A=M`)

2. If you run out of registers (you have only two ...), you may use `R13`, `R14`, and `R15`.

Memory Segments



@16? Static

Local @LCL

@ARG Argument

Pointer @3

Temp @5

@THIS This

That @THAT

VM implementation on the Hack platform (translation)

- `push constant 1`

- `add`

- `pop local 2`

VM implementation on the Hack platform (translator)

- **push constant 1**

```
@1
D=A
@SP
A=M
M=D
@SP
M=M+1
```

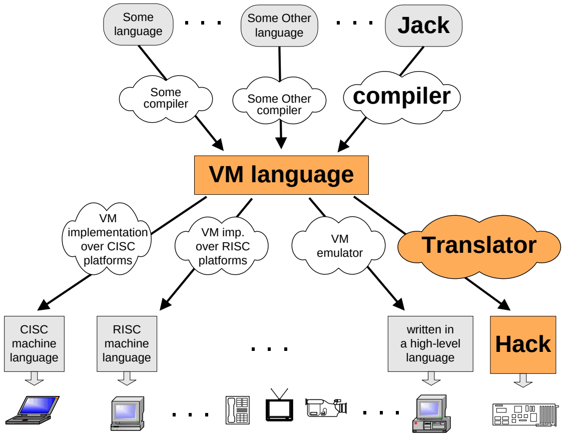
- **add**

```
@SP
AM=M-1
D=M
A=A-1
M=M+D
```

- **pop local 2**

```
@LCL
D=M
@2
D=D+A
@R15
M=D
@SP
AM=M-1
D=M
@R15
A=M
M=D
```

Perspective



- In this lecture we began the process of building a compiler

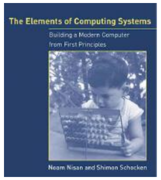
- Modern compiler architecture:
  - Front-end (translates from a high-level language to a VM language)
  - Back-end (translates from the VM language to the machine language of some target hardware platform)

- Brief history of virtual machines:
  - 1970's: p-Code
  - 1990's: Java's JVM
  - 2000's: Microsoft .NET

- A full blown VM implementation typically also 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 big picture

Java	Microsoft .net		
❑ JVM	❑ CLR	❑ VM	❑ 7, 8
❑ Java	❑ C#	❑ Jack	❑ 9
❑ Java compiler	❑ C# compiler	❑ Jack compiler	❑ 10, 11
❑ JRE	❑ .NET base class library	❑ Mini OS	❑ 12

(Book chapters and Course projects)