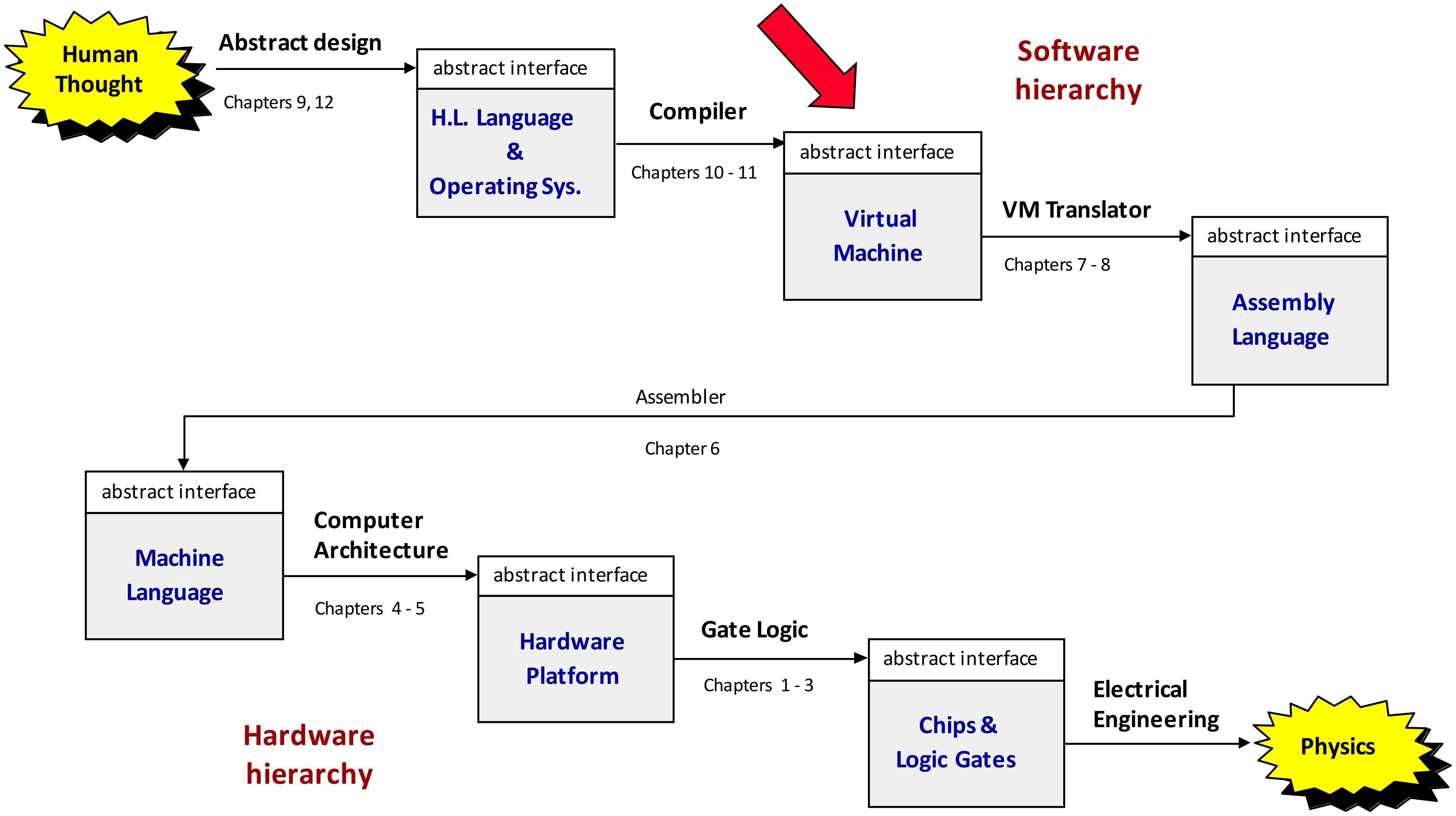
School of Computer Science

COMP SCI 2000 Computer Systems
Lecture 13

Review - where we are:



This Lecture

- VM memory model
- VM functions
- Basic VM translation to assembly code

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 high-level languages normally feature the following variable kinds:

<u>Class level:</u>

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

<u>Method level:</u>

- Local variables
- Argument variables

<u>When translated into the VM language,</u>

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 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

Notes:
(In all our code examples thus far, *memorySegment* was `static`)
The roles of the eight memory segments will become relevant when we talk about compiling
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

The example VM program includes four new VM commands:

- `function` *functionSymbol int* // function declaration
- `label` *labelSymbol* // label declaration
- `goto` *labelSymbol* // jump to execute the command after *labelSymbol*
- `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.
```

VM programming (example)

High-level code

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

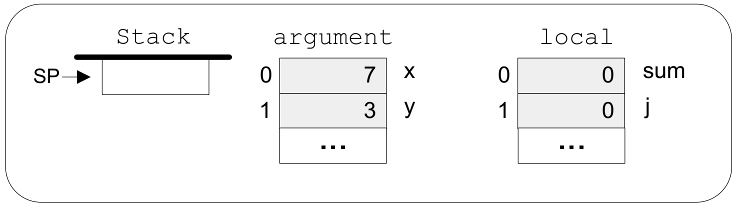
VM code (first approx.)

```
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
```

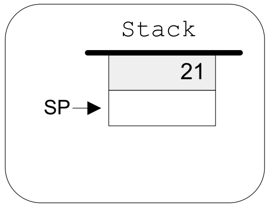
VM code

```
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
```

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



VM Programming Example

- Worksheet Lecture 13, Question 1
  - Translating a function into Hack Virtual Machine code

VM programming: multiple functions

Compilation:

- A Jack application is a set of 1 or more class files (just like `.java` files).
- When we apply the Jack compiler to these files, the compiler creates a set of 1 or more `.vm` files (just like `.class` files). Each method in the Jack app is translated into a VM function written in the VM language
- Thus, a VM file consists of one or more VM functions.

Execution:

- At any given point of time, only one VM function is executing (the "current function"), while 0 or more functions are waiting for it to terminate (the functions up the "calling hierarchy")
- For example, a `main` function starts running; at some point we may reach the command `call factorial`, at which point the `factorial` function starts running; then we may reach the command `call mult`, at which point the `mult` function starts running, while both `main` and `factorial` are waiting for it to terminate

`The stack:` a global data structure, used to save and restore the resources (memory segments) of all the VM functions up the calling hierarchy (e.g. `main` and `factorial`). The tip of this stack if the working stack of the current function (e.g. `mult`).

Lecture plan

Goal: Specify and implement a VM model and language:

Arithmetic / Boolean commands	Program flow commands
add	label (declaration)
sub	goto (label)
neg	if-goto (label)
eq	Next lecture
gt	**Function calling commands**
lt	function (declaration)
and	call (a function)
or	return (from a function)
not	
This lecture	
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 (stack, memory segments, commands)

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

Implementation

VM implementation options:

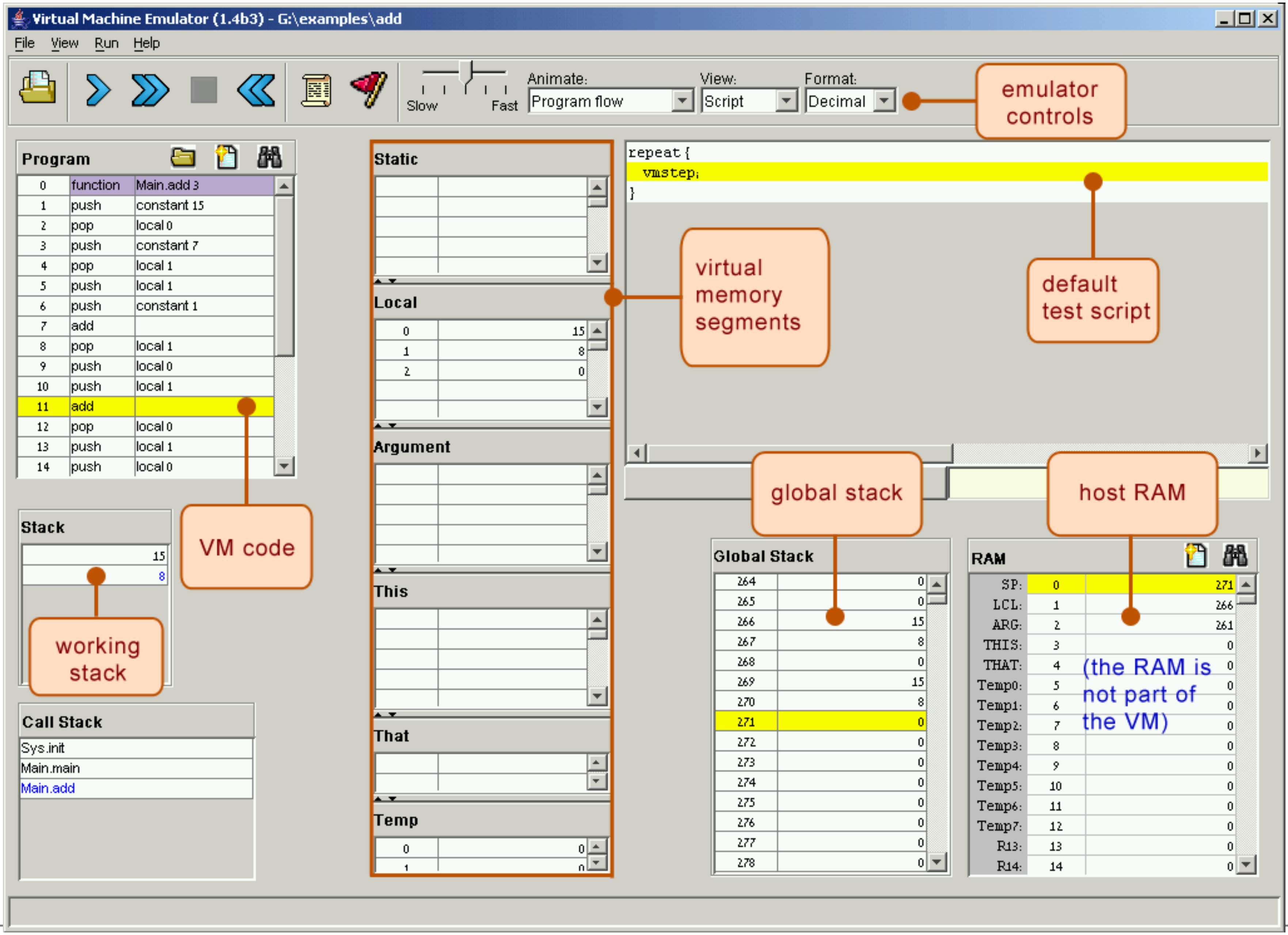
- Software-based (e.g. emulate the VM model using Java)
- Translator-based (e. g. translate VM programs into the Hack machine language)
- Hardware-based (realize the VM model using dedicated memory and registers)

Two well-known translator-based implementations:

JVM: Javac translates Java programs into bytecode;
The JVM translates the bytecode into
the machine language of the host computer

CLR: C# compiler translates C# programs into IL code;
The CLR translated the IL code into
the machine language of the host computer.

Software implementation: Our VM emulator



Virtual Machine Emulator (1.4b3) - G:\examples\add

File View Run Help

Animate: Program flow — View: Script — Format: Decimal

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

Stack

```
15
8
```

Call Stack

```
Sys.init
Main.main
Main.add
```

Static

Local

0	15
1	8
2	0

Argument

This

That

Temp

0	0
1	0

```
repeat {
  vmstep;
}
```

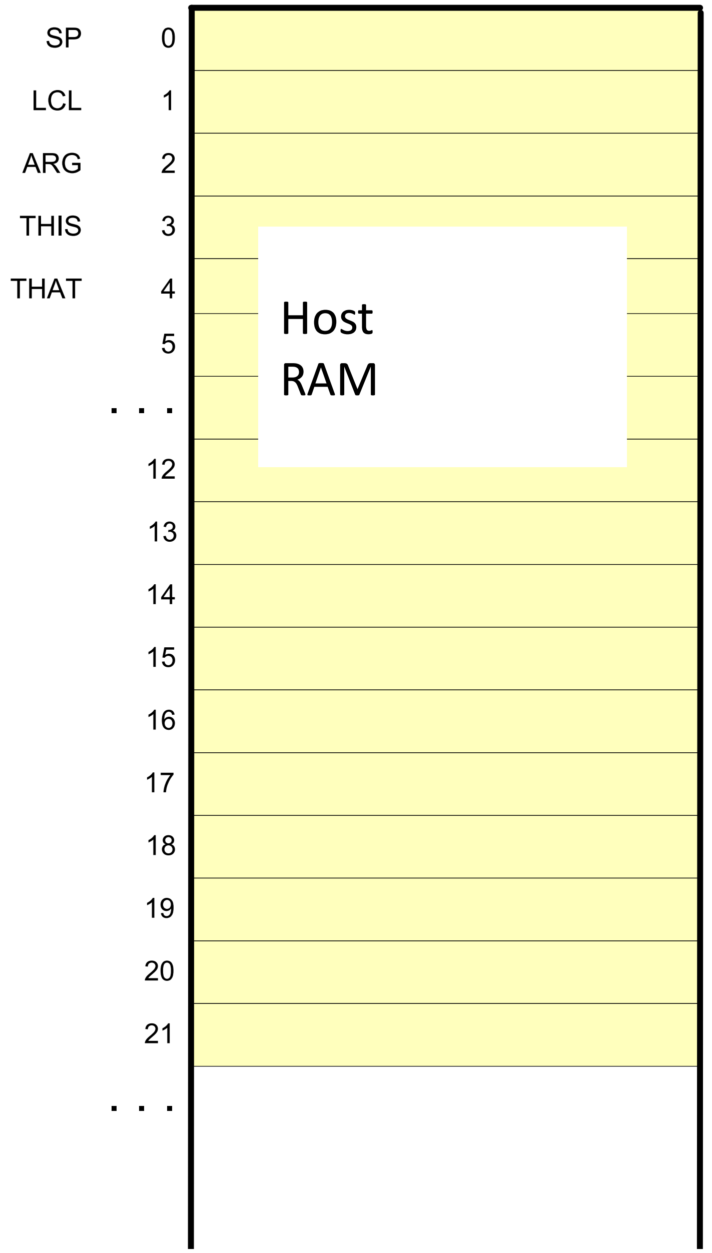
Global Stack

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

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

VM implementation on the Hack platform



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

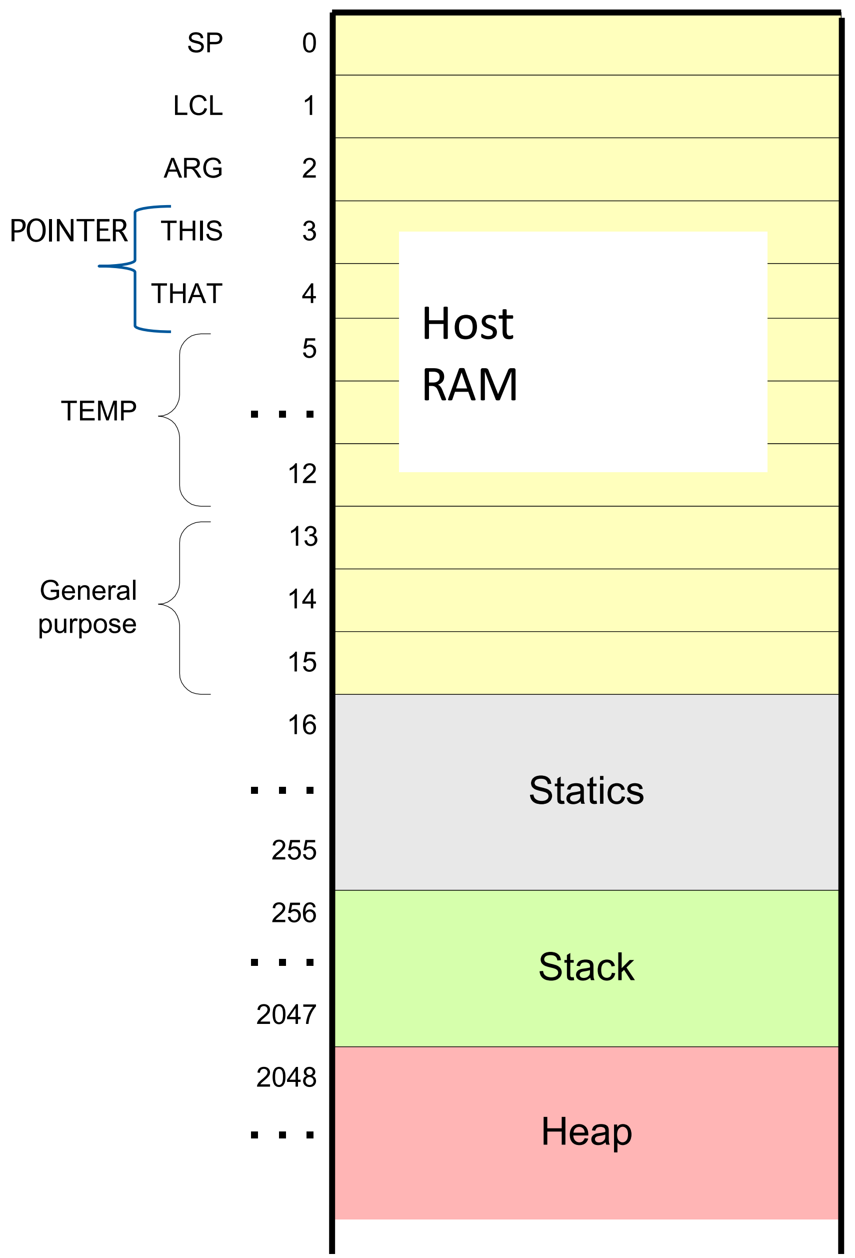
The tip of this stack is the working stack of the current function

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

`local,argument,this,that`:
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



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

<u>The stack</u>: mapped on `RAM[256 ... 2047]`;
The stack pointer is kept in RAM address `SP`

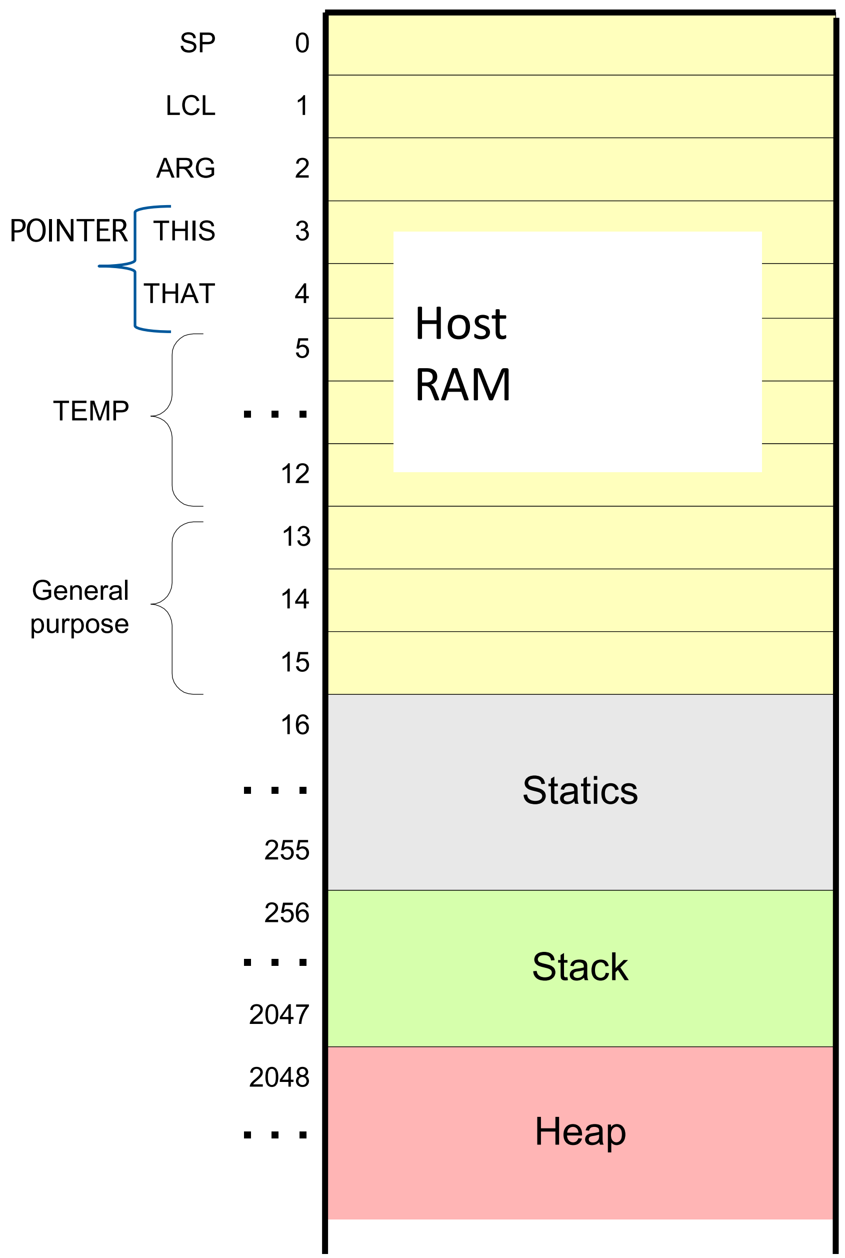
<u>`static`</u>: mapped on `RAM[16 ... 255]`;
each segment reference `static` *i* appearing in a VM file named `f` is compiled to the assembly language symbol `f.i` (recall that the assembler further maps such symbols to the RAM, from address 16 onward)

<u>`local,argument,this,that`</u>: these method-level segments are mapped somewhere from address 256 onward, on the "stack" or the "heap". The base addresses of these segments are kept in RAM addresses `LCL`, `ARG`, `THIS`, and `THAT`. Access to the *i*-th entry of any of these segments is implemented by accessing `RAM[segmentBase + ` *i*`]`

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

<u>`pointer`</u>: RAM[3..4] to change `THIS` and `THAT`.

VM implementation on the Hack platform



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`.

VM Translator Parsing

- Worksheet Lecture 13, Question 2
  - Translating Hack Virtual Machine code into Hack Assembly language.

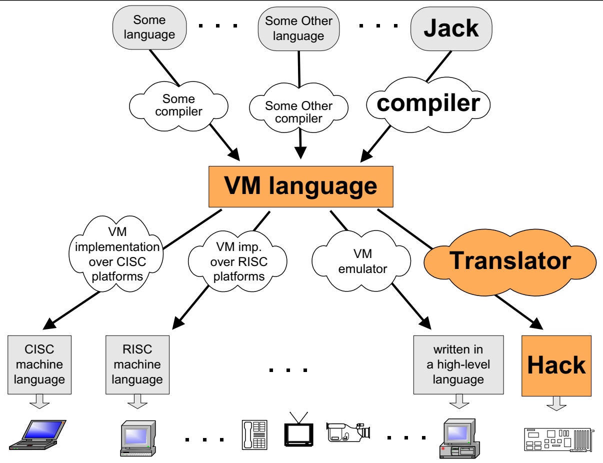
Proposed VM translator: Parser module

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 arg. 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: CodeWriter module

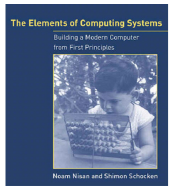
CodeWriter: Translates VM commands into Hack assembly code.			
Routine	**Arguments**	**Returns**	**Function**
Constructor	Output file / stream	--	Opens the output file/stream and gets ready to write into it.
`setFileName`	fileName (string)	--	Informs the code writer that the translation of a new VM file 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 the next lecture / chapter 8.			



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)

The Elements of Computing Systems
Building a Modern Computer from First Principles
Noam Nisan and Shimon Schocken

Next lecture

- Flow control implementation.
- Function implementation.
- Questions?