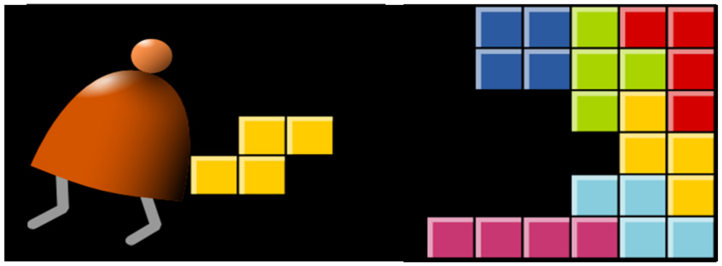
Virtual Machine

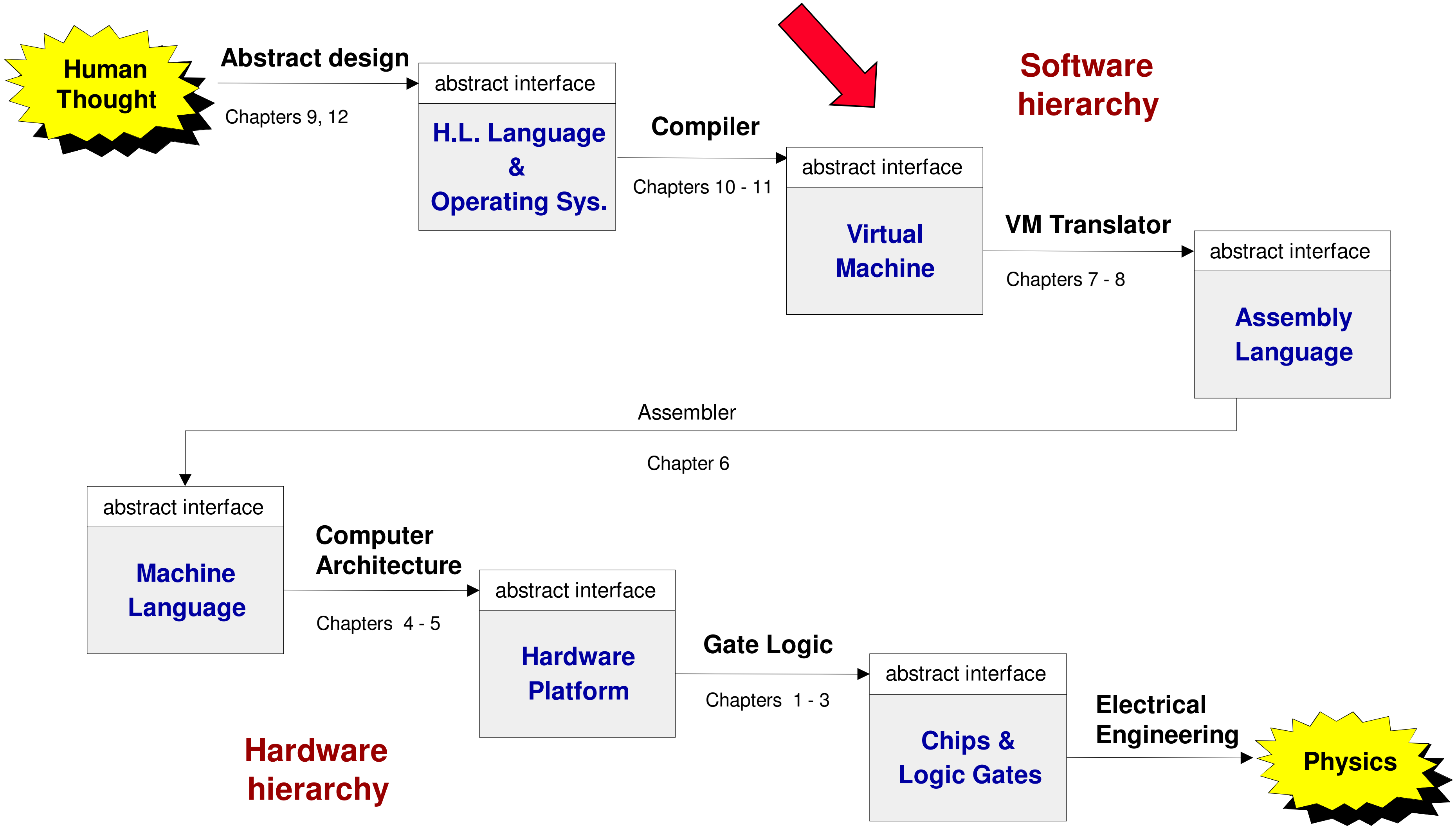
Part I: Stack Arithmetic



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



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)

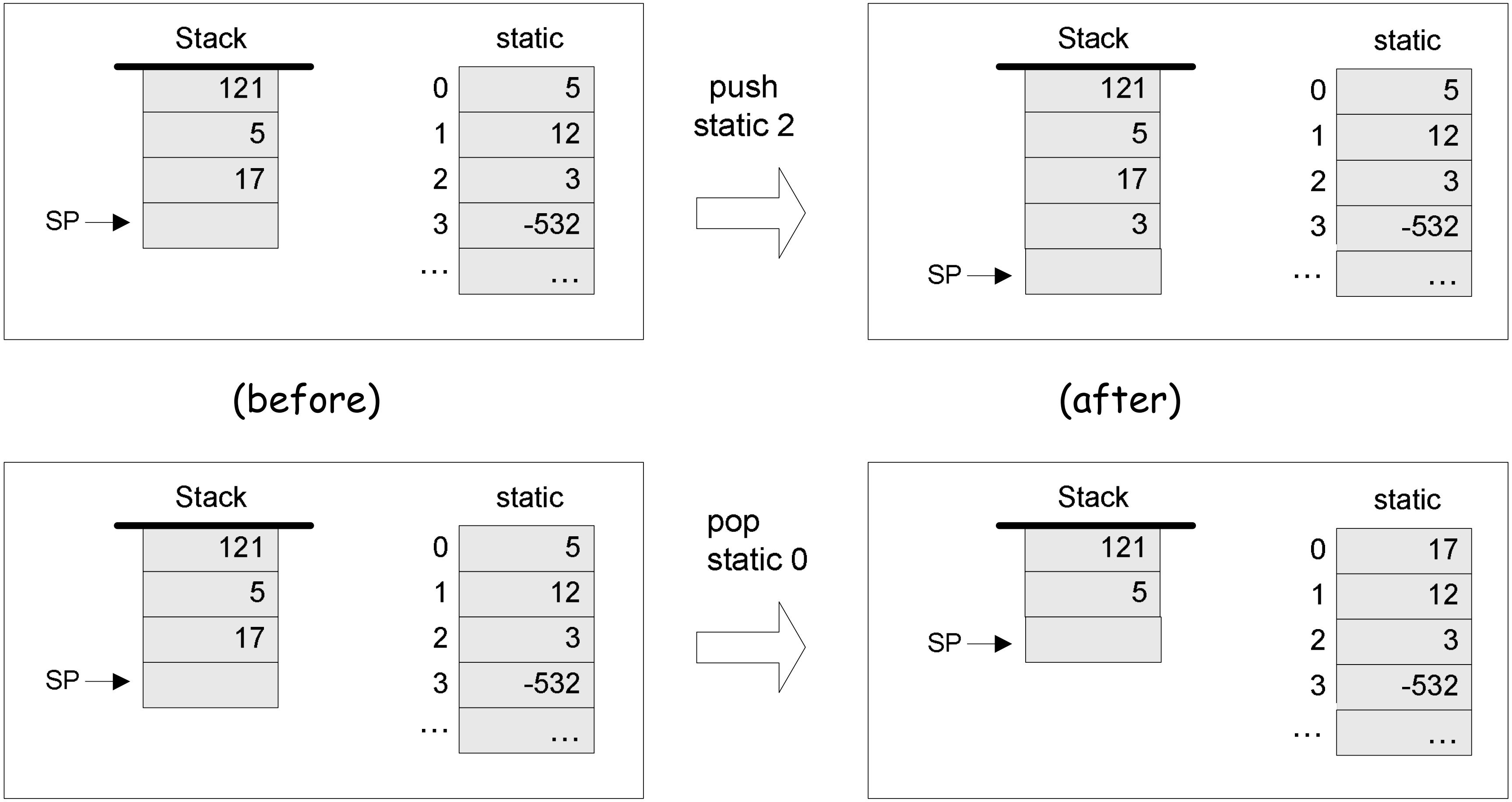
return (from a function)

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

Our VM model is *stack-oriented*

- The VM is an architecture in its own right.
- The VM architecture is an example of a CISC architecture.
- The Hack architecture is an example of a RISC architecture.
- Intel's x86 architecture is a CISC architecture. x86 instructions are *translated by the hardware* into RISC-like micro-ops. These micro-ops are the actual units of execution within an x86 processor. In a sense, an x86 processor's front-end performs a VM translation analogous to the translation you'll be working on.

Memory access operations



Stack
121
5
17
SP →

	static
0	5
1	12
2	3
3	-532
…	…

push
static 2

Stack
121
5
17
3
SP →

	static
0	5
1	12
2	3
3	-532
…	…

(before)

(after)

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
…	…

The stack:

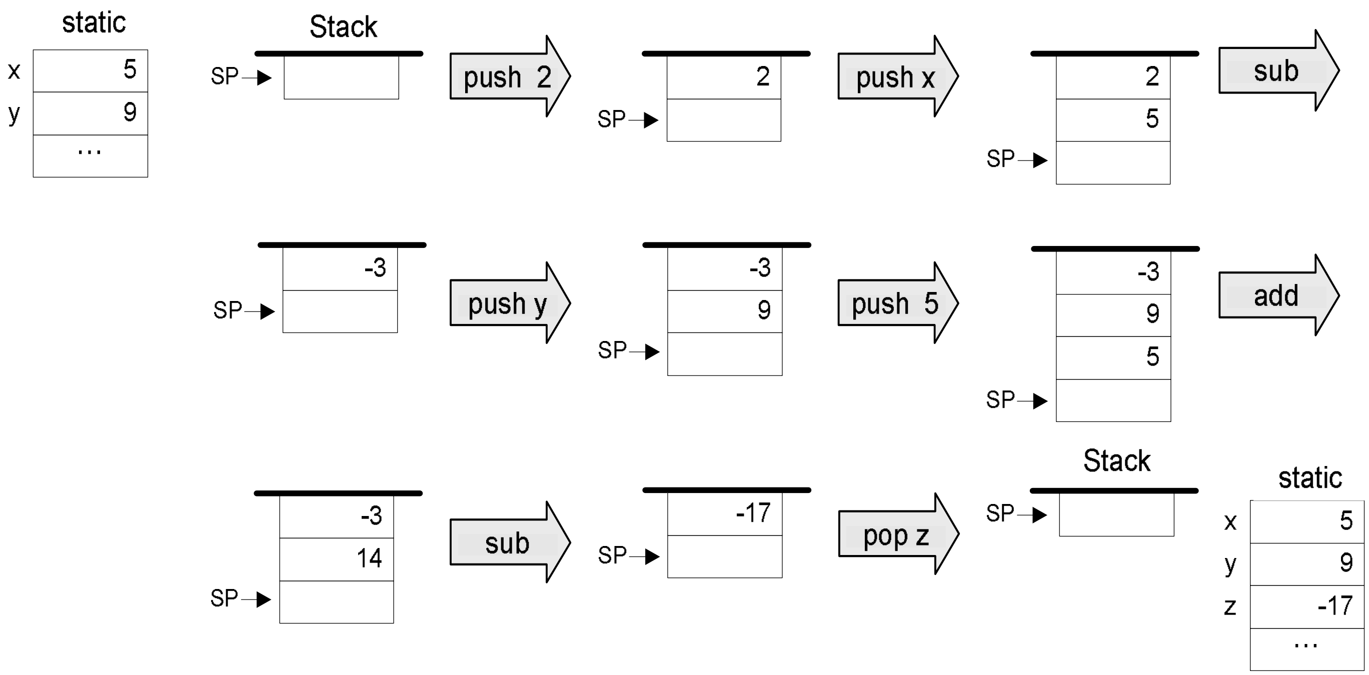
- A classical LIFO data structure
- Elegant and powerful
- Several hardware / software implementation options.

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, and
z refers to static 2)

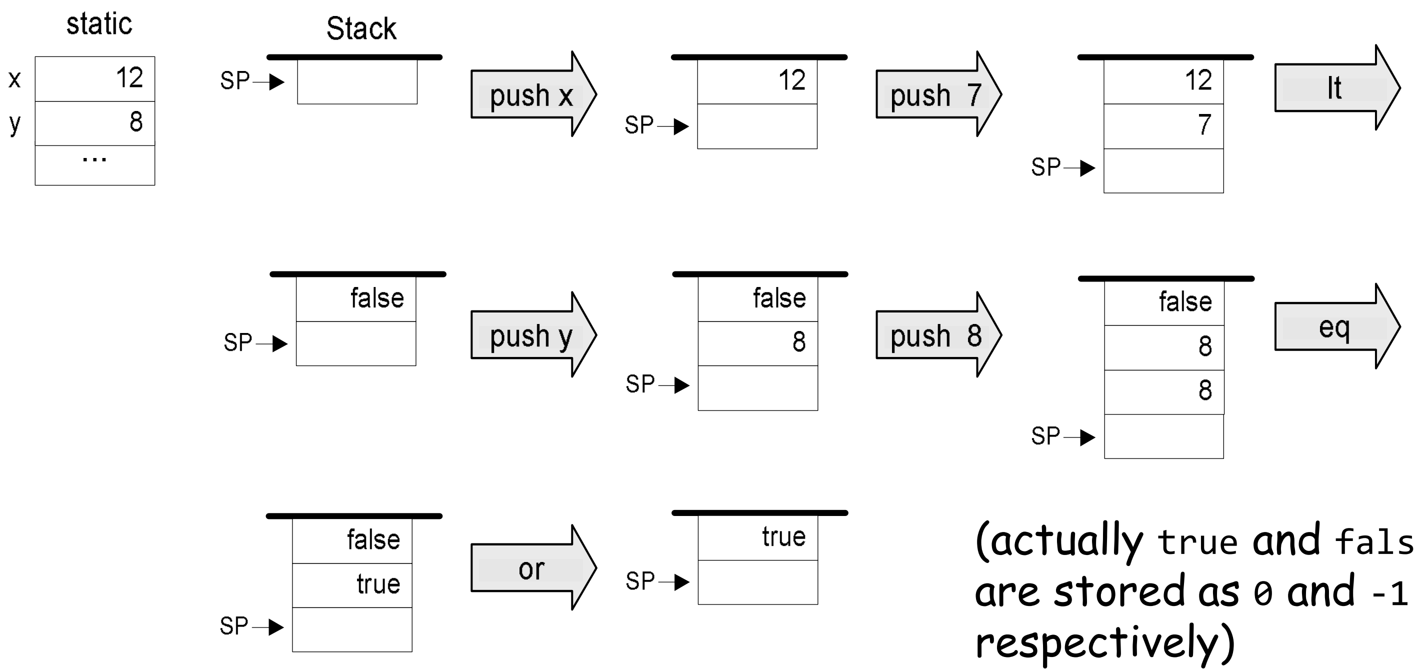


Evaluation of Boolean expressions

VM code (example)

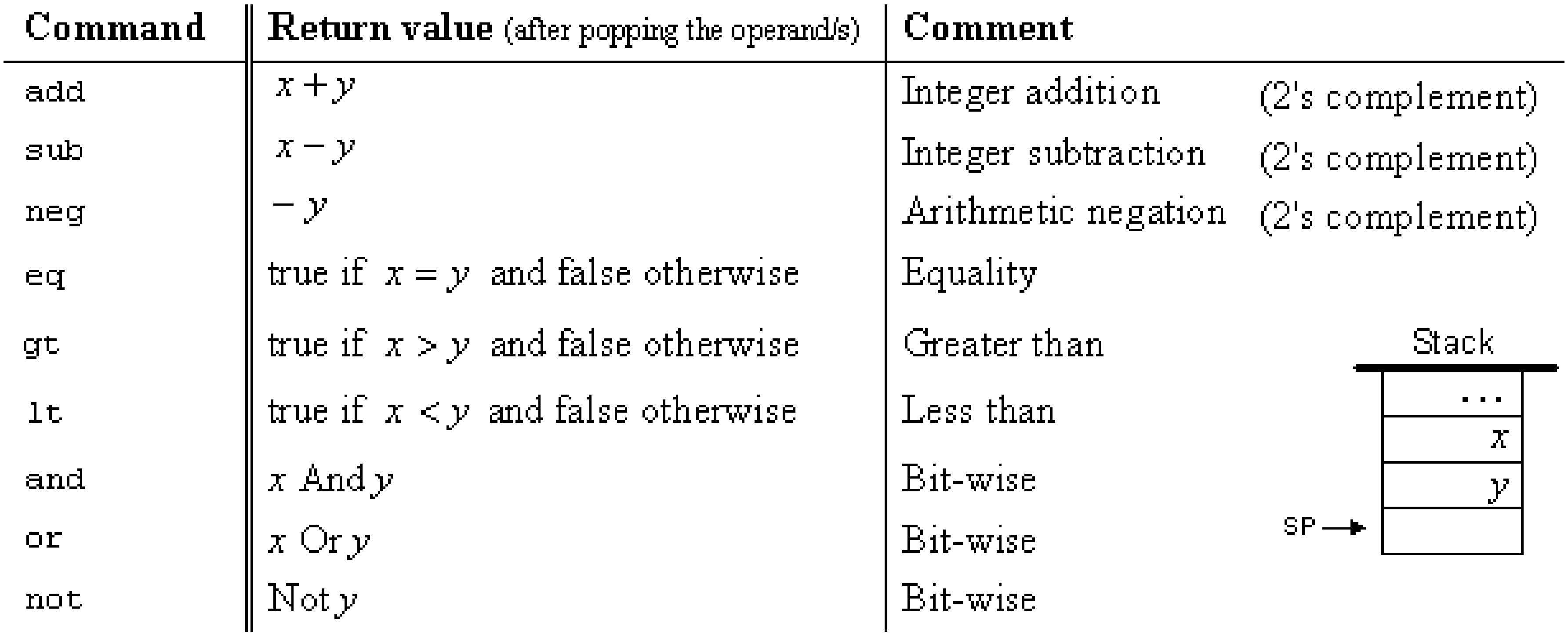
```
// (x<7) or (y=8)
push x
push 7
lt
push y
push 8
eq
or
```

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



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

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	

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:

<u>Memory access VM commands:</u>

- 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

<u>Notes:</u>

(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.

Implementation

VM implementation options:

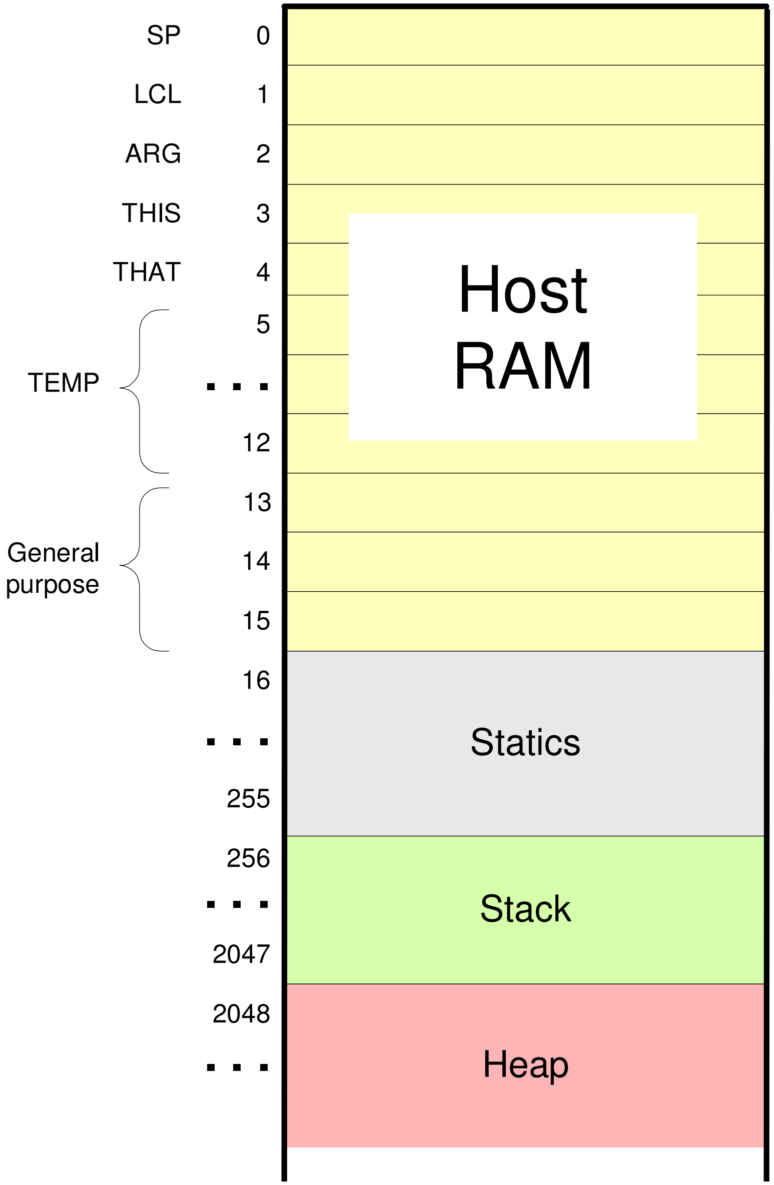
- Software-based (e.g. emulate the VM model using Java)
- Translator-based (e. g. translate VM programs into the Hack machine language)
- Hardware-based (realize the VM model using dedicated memory and registers)

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.

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 method-level segments is dynamic, using pointers

The stack: mapped on RAM[256 ... 2047]; The stack pointer is kept in RAM address SP. Local and argument segments mapped in here.

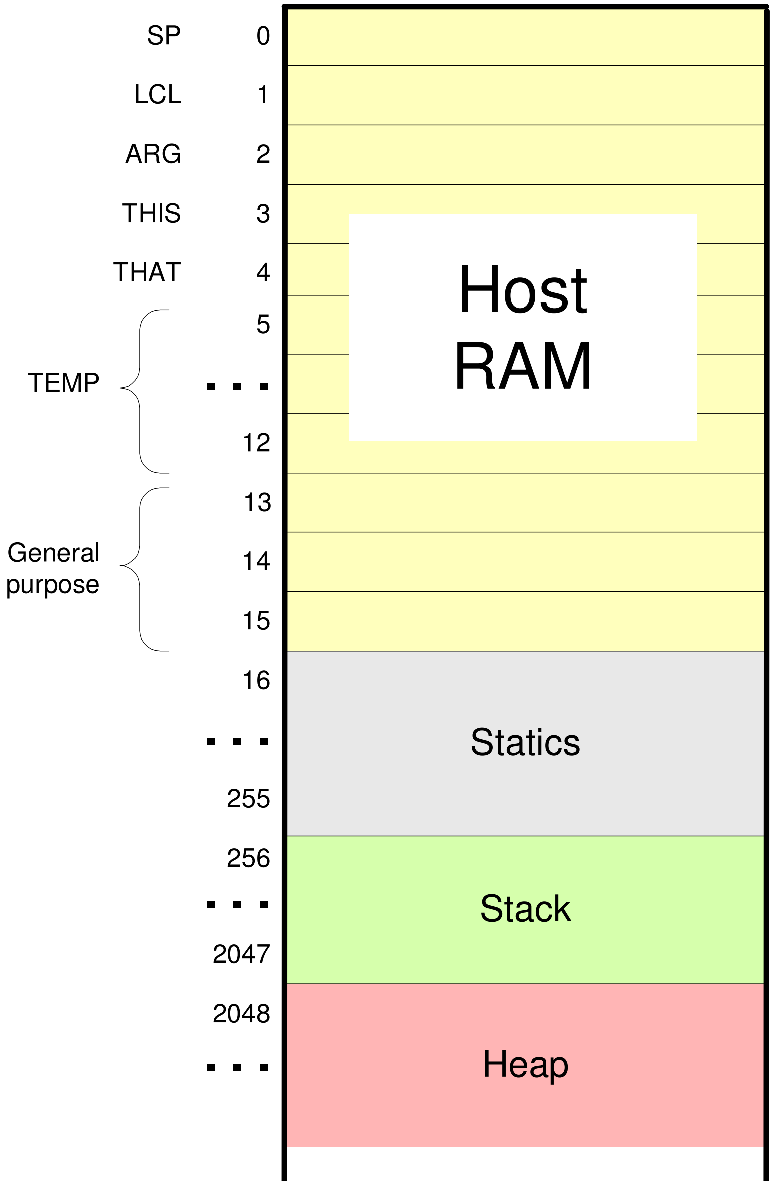
static: mapped on RAM[16 ... 255]; each segment reference static *i* appearing in a VM file named f is compiled to the assembly language symbol f.i

local,argument,this,that: these method-level segments are mapped somewhere from address 2048 onward, in an area called "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*]

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

pointer: discussed later.

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`.

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 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 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 the next lecture / chapter 8.