Agenda

What is a Virtual Machine?

The VM Language (Part 1)

- Stack arithmetic and logic
  - add, sub, neg, eq, gt, lt, and, or, not
- Virtual memory segments
  - push/pop commands

With slides from nand2tetris.org

Virtual machine

A **virtual machine** abstracts the hardware of a system using software

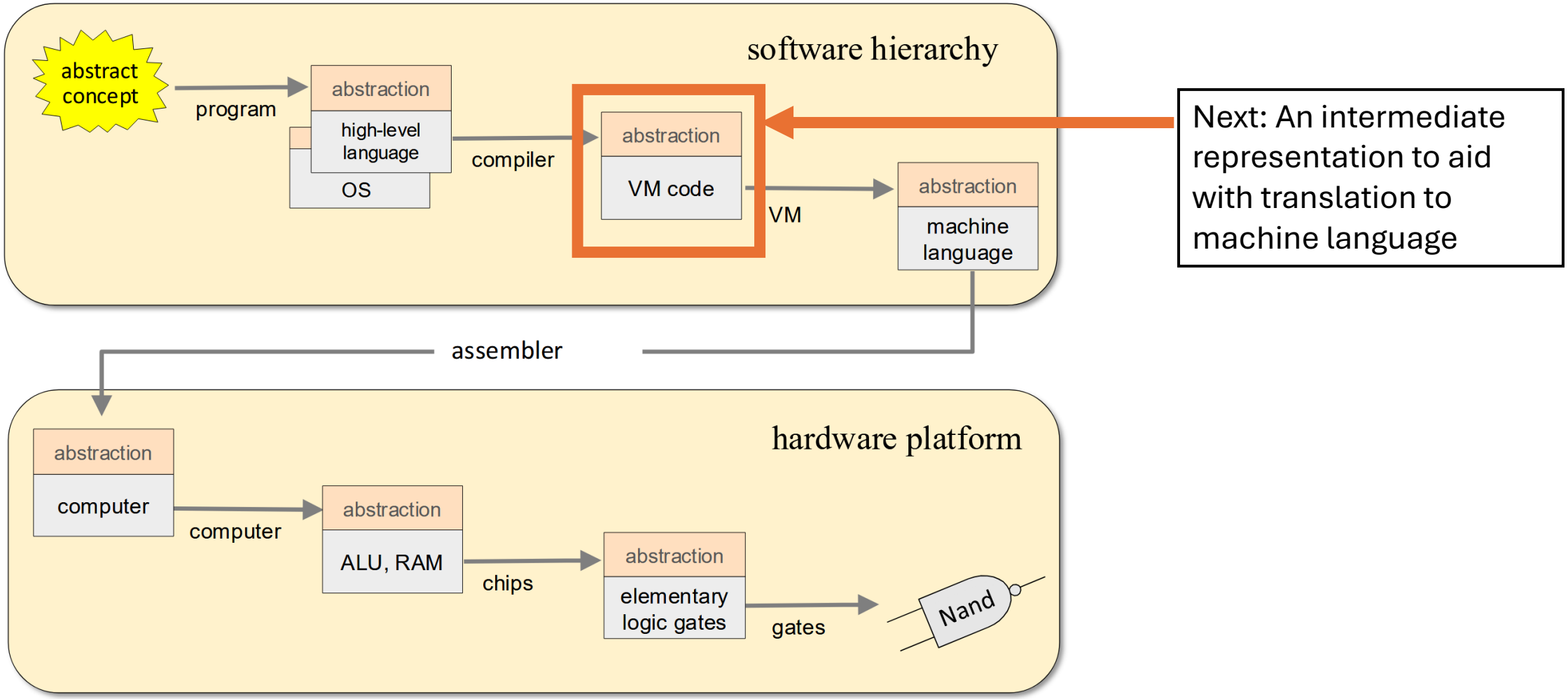
Virtual machines use a hardware-independent execution layer

- Example: The Java virtual machine can run on any hardware
- Example: VirtualBox allows you to simulate any operating system on your computer

Hack uses a virtual machine (VM) as an intermediary between high-level code and machine code

- VM Code: Generated by the compiler
- VM Translator: Translates VM code to machine code/assembly

The big picture: Hack



Next: An intermediate representation to aid with translation to machine language

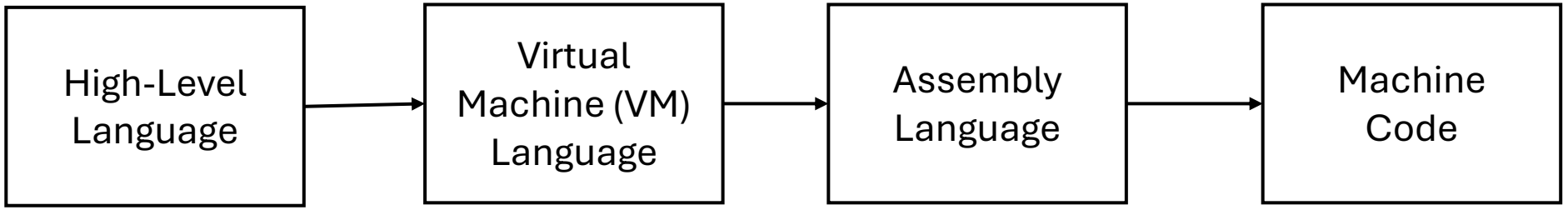
Recall: Compilation

How do we get from a high-level language to machine code?

Perform a series of translations that are increasingly low-level

Each level *abstracts* the details of the level below it

For example, assembly programs do not need to know the details of machine code



Writing a VM Translator for Hack

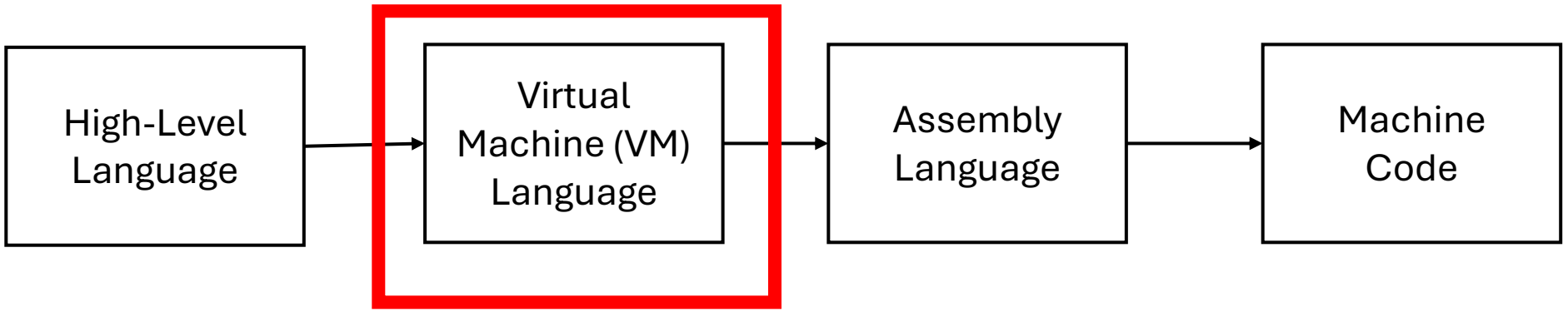
The VM translator converts high-level code to VM Code

VM code is entirely stack-based

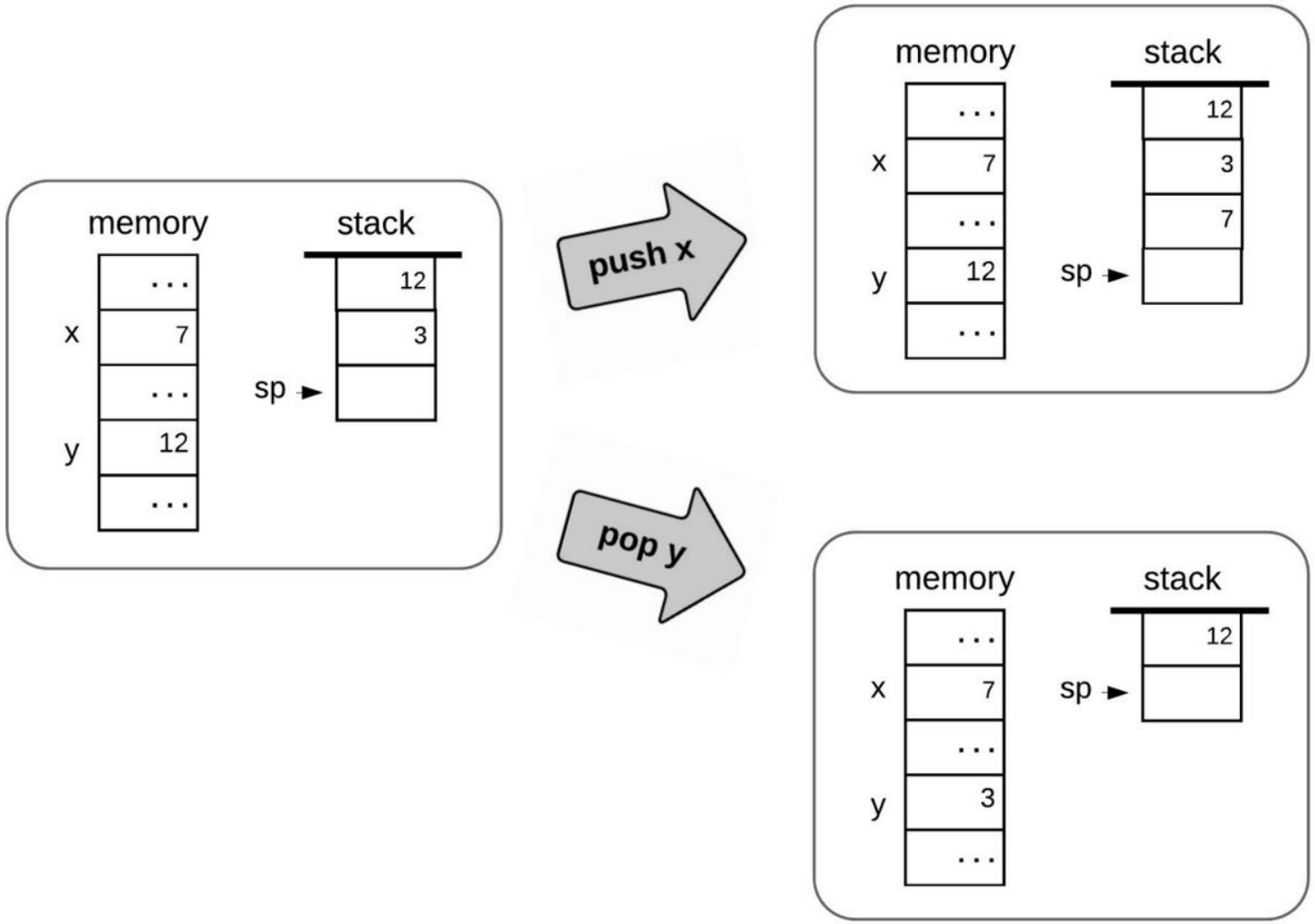
- Only uses push and pop
- Different stacks are used to store different types of variables

Two primary features:

- Expressions (arithmetic, conditionals, etc)
- Function calls



Stacks



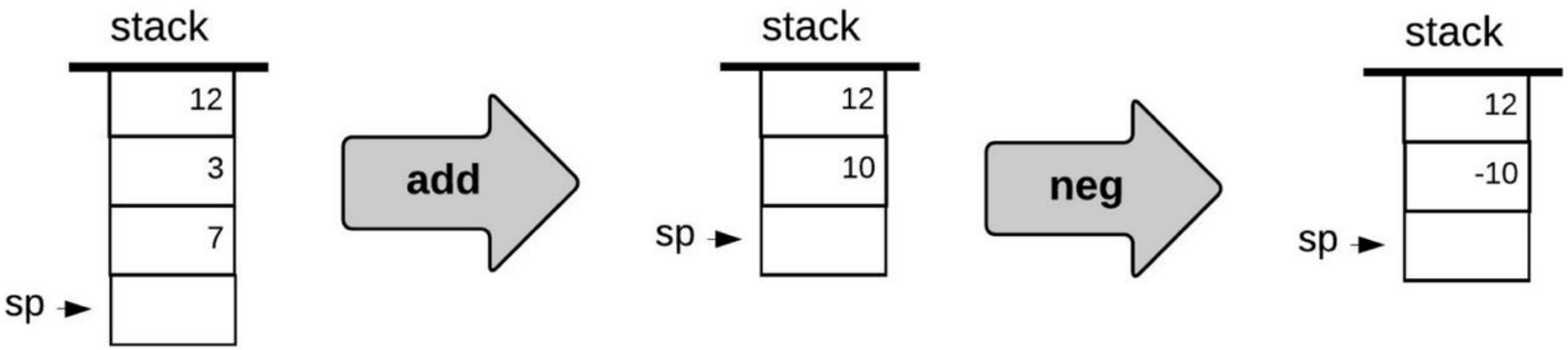
SP = Stack Pointer

Pop removes the topmost element

Push adds an element

Our stacks grow downwards

Idea: Implementing expressions with stacks



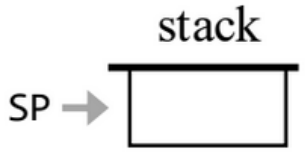
Applying a function f (that has n arguments)

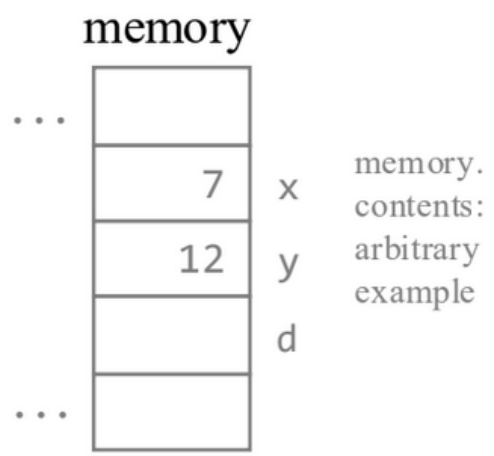
- pops n values (arguments) from the stack,
- Computes f on the values,
- Pushes the resulting value onto the stack.

Exercise: Stack arithmetic

VM pseudocode (example)

```
// d = (2 - x) + (y + 9)
```





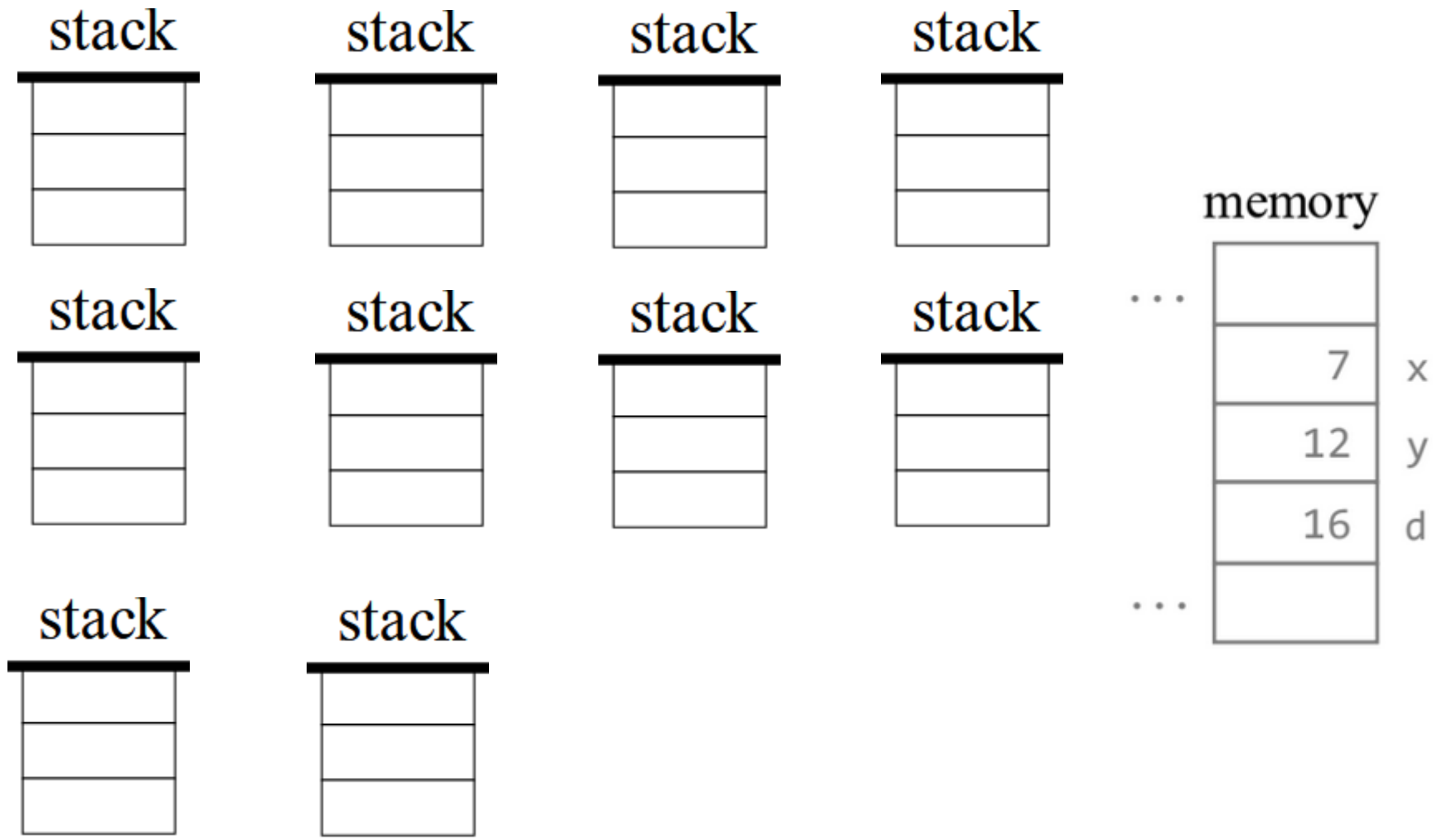
Convert the expression to VM code

Exercise: Stack arithmetic

Visualize the VM Code

VM pseudocode

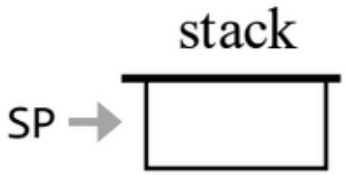
```
// d = (2 - x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```

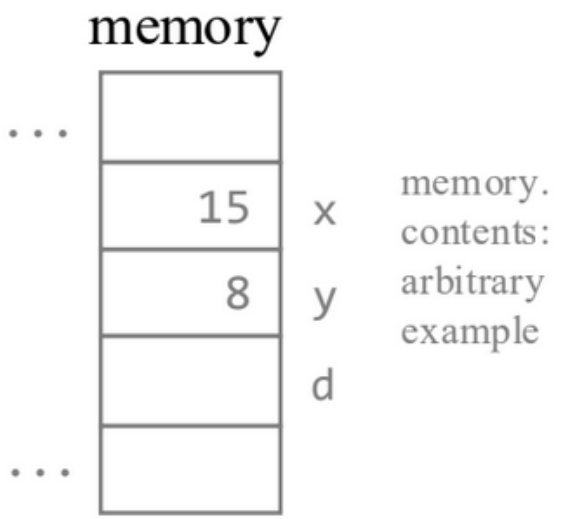


Exercise: Stack Logic

VM pseudocode (another example)

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





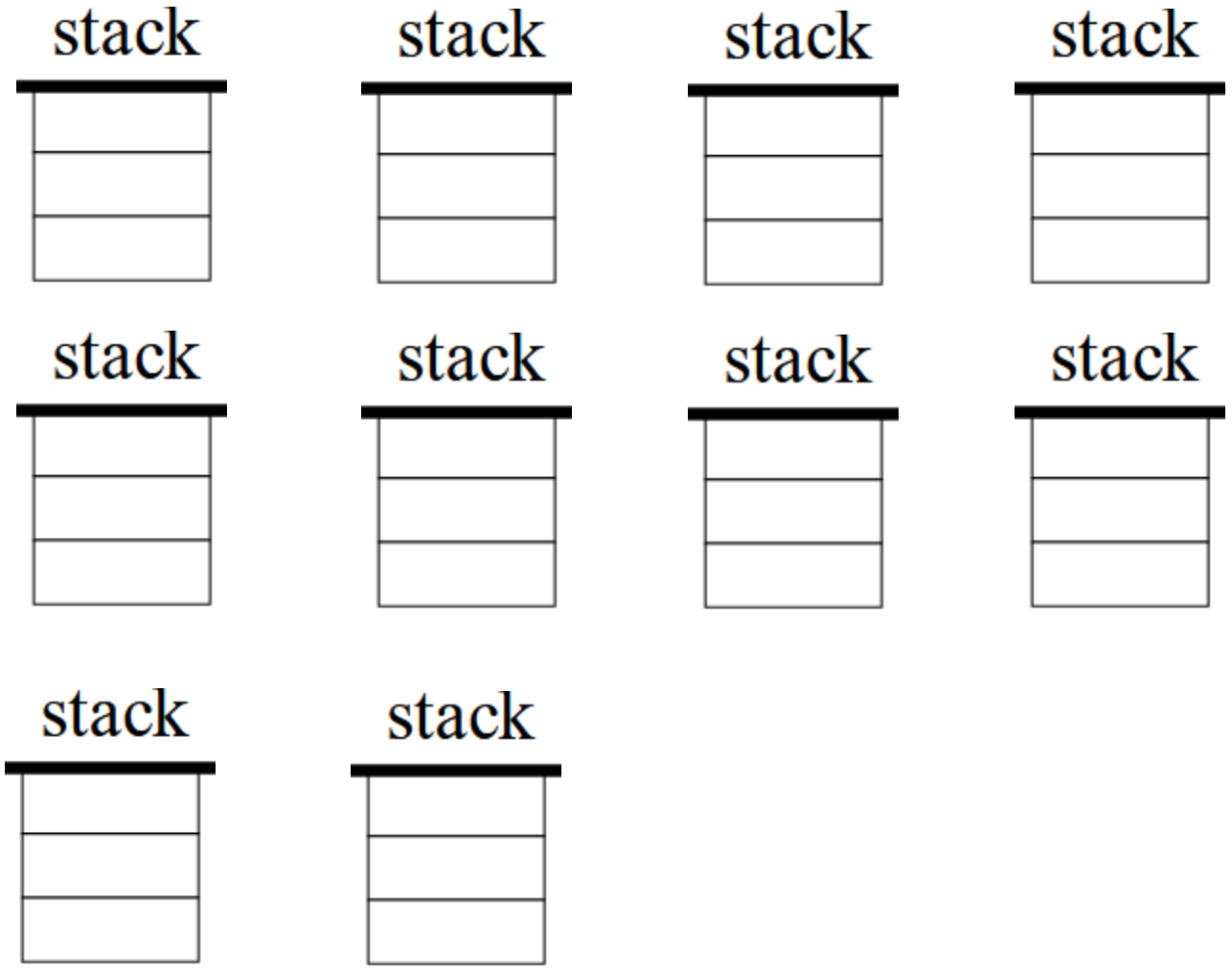
Write the VM Code

Visualize: Stack Logic

Visualize the VM Code

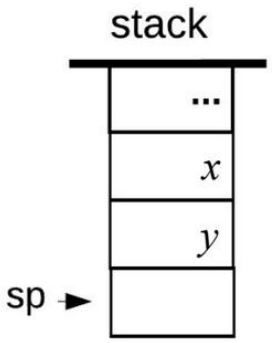
VM pseudocode (another

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



Stack functions: summary

command	operation	returns
add	$x + y$	integer
sub	$x - y$	integer
neg	$-y$	integer
eq	x == y	boolean
gt	$x > y$	boolean
lt	$x < y$	boolean
and	x And y	boolean
or	x Or y	boolean
not	Not x	boolean



Each command pops as many operands as it needs from the stack, computes the specified operation, and pushes the result onto the stack.

Every high-level arithmetic or logical expressions can be translated to a sequence of VM commands, operating on a stack.

But how to handle the arguments to our functions?

Virtual memory segments

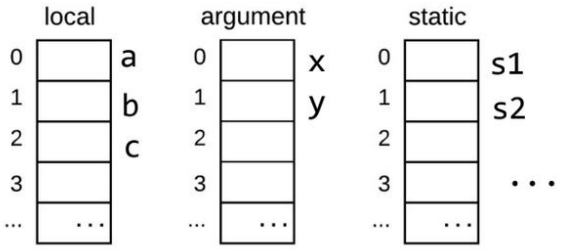
Source code (e.g. Java)

```
class Foo {
   static int s1, s2;
   public int bar(int x, int y) {
      int a, b, c;
      ...
      c = s1 + y;
      ...
      return c;
   }
}
```

compiler

Compiled VM code

```
...
...
...
...
push static 0
push argument 1
add
pop local 2
...
```



virtual memory segments

The compiler...

1. Represents variables by *virtual memory segments*, according to their *kinds*: `local`, `argument`, `static`, ...
2. Generates VM commands that operate on the stack and on the virtual memory segments.

Hack has 8 virtual memory segments:

- local
- argument
- static
- constant
- this
- that
- temp
- pointer

Syntax:

push / pop segment i

Exercise: Write VM Code that references virtual memory segments (VMS)

```
int global = 0;

void foo(int a, int b)
{
   global = a + 5;
}
```

High-level Code

Recall Syntax: push/pop segment i

How should we implement VM Code?

Where/how should our stacks be stored? Three Approaches:

Native: Extend the computer's hardware with modules that represent the stack, the stack pointer, and other VM constructs; Extend the computer's instruction set with primitive versions of the VM commands;

Emulation: Write a program in a high level language that represents the stack and the virtual memory segments as ADTs (abstract data type); Implement the VM commands as methods that operate on these ADTs;

Translation: Translate each VM command into machine language instructions that operate on a host RAM; Use an addressing contract that realizes the stack and the virtual address segments

How should we implement VM Code?

Where/how should our stacks be stored? Three Approaches:

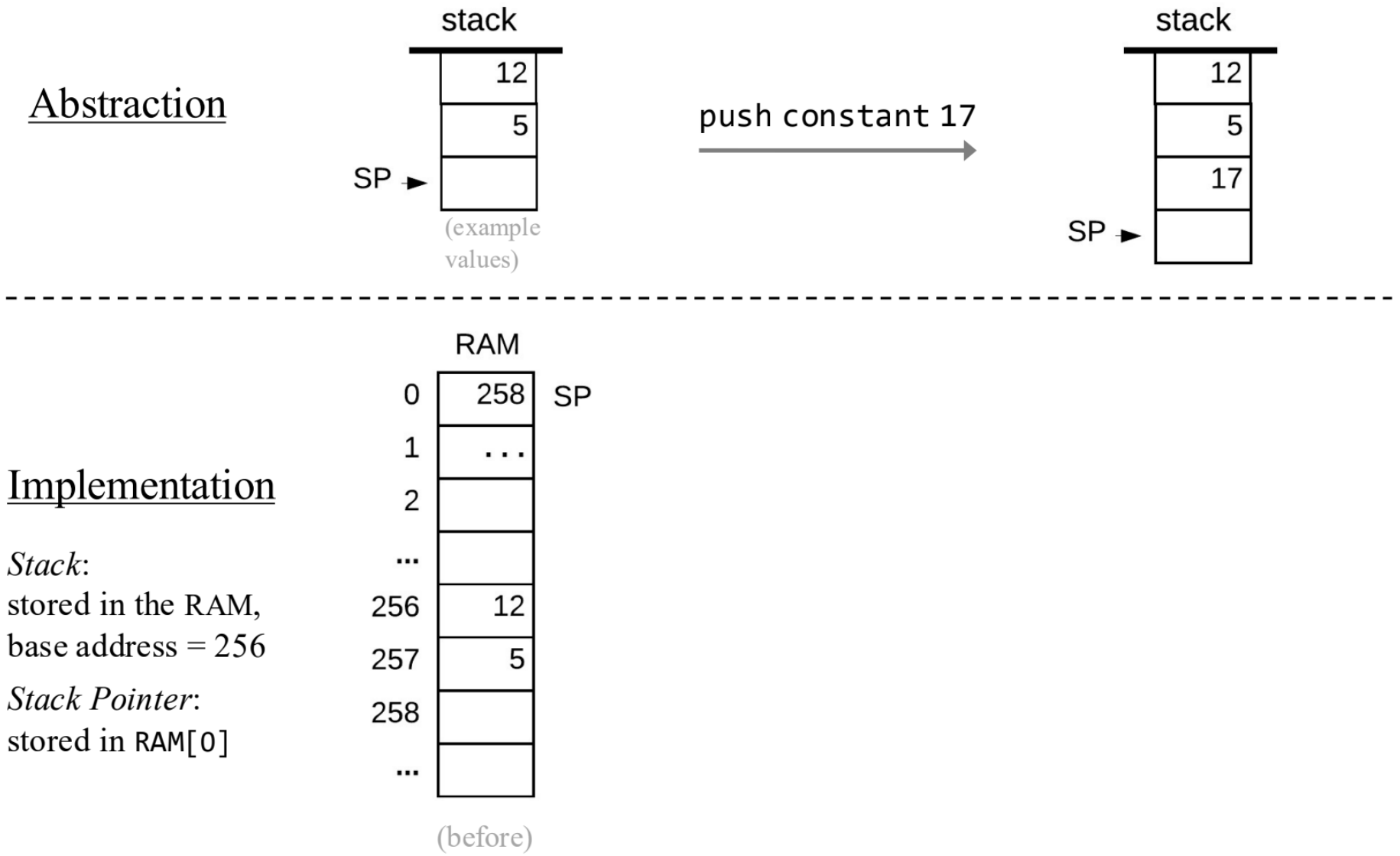
Native: Extend the computer's hardware with modules that represent the stack, the stack pointer, and other VM constructs; Extend the computer's instruction set with primitive versions of the VM commands;

Emulation: Write a program in ...ts the stack and the virtual memory s... ...VM commands as methods tha...

The approach taken by:
- Java, C#, Python, Ruby, Scala, …
- Jack (designed in Nand to Tetris)

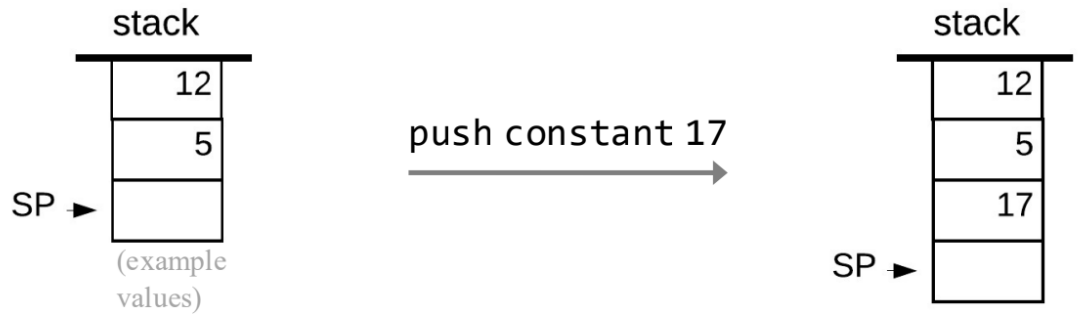
Translation: Translate each VM command into machine language instructions that operate on a host RAM; Use an addressing contract that realizes the stack and the virtual address segments

Constants (integers)



Implementing constants

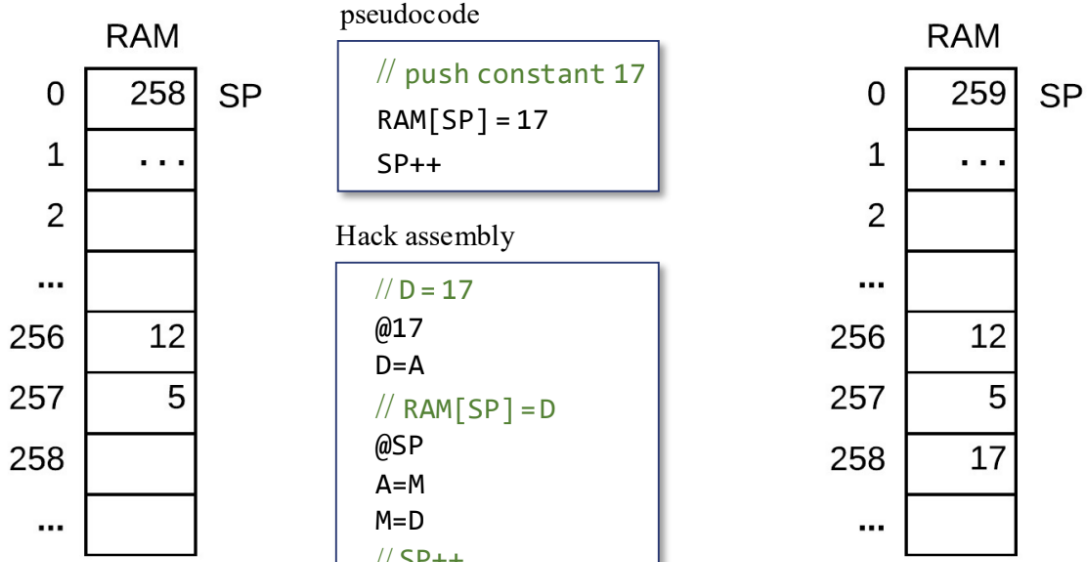
Abstraction



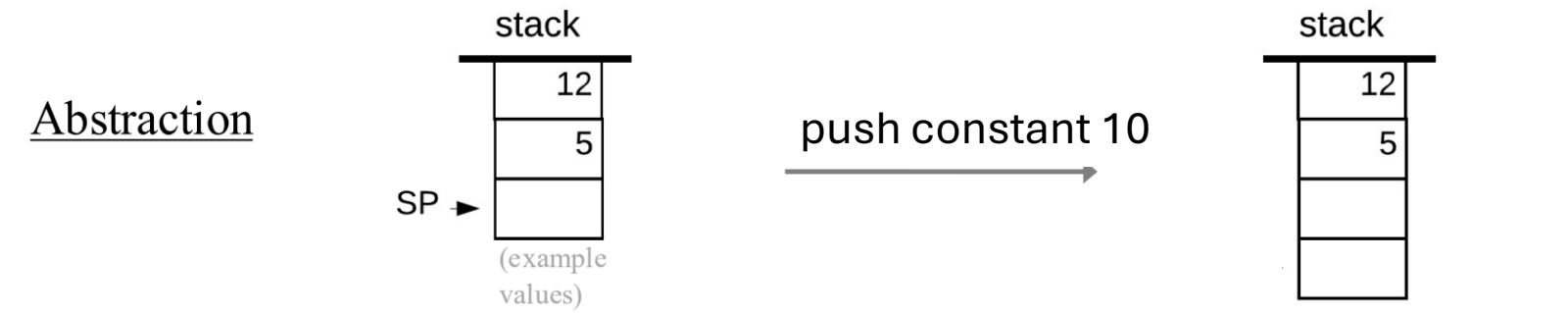
Implementation

Stack:
stored in the RAM,
base address = 256

Stack Pointer:
stored in `RAM[0]`



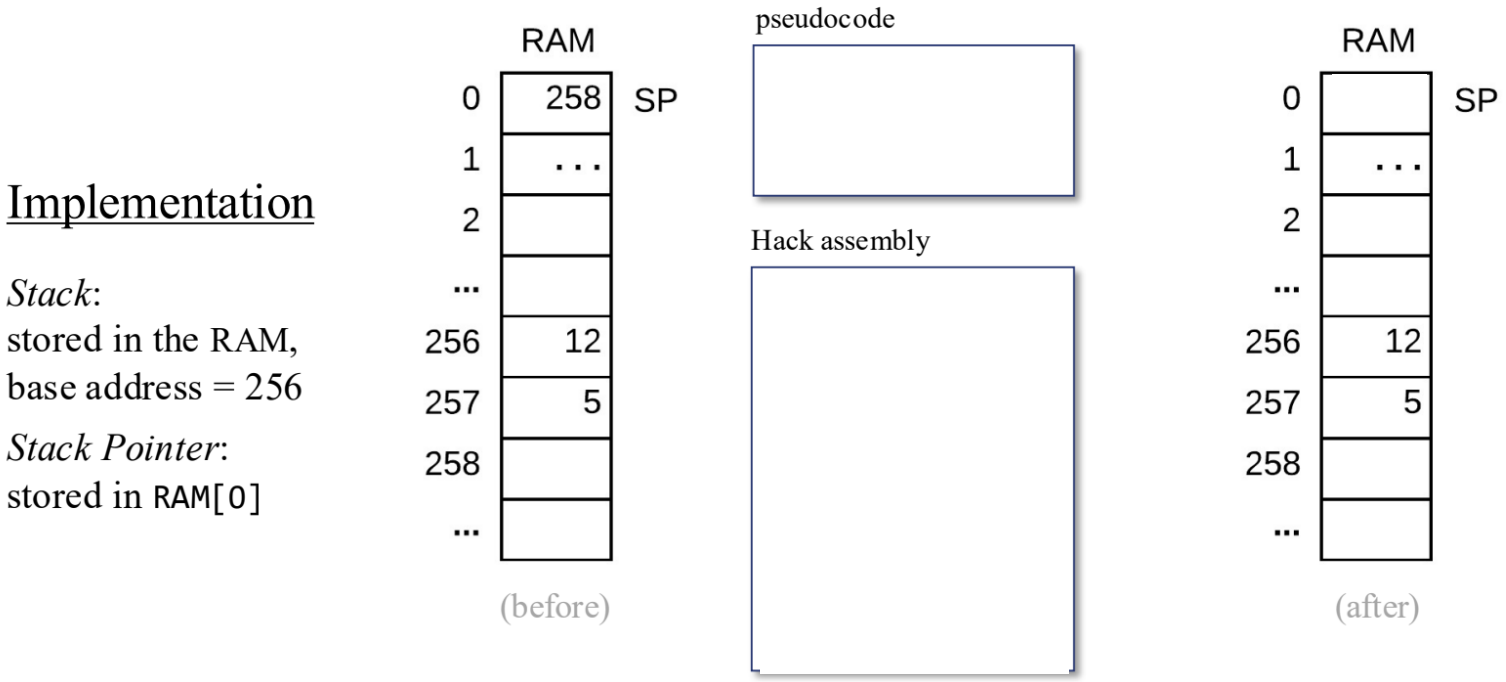
Exercise: Implementing constants



a = 10 → push constant 10

1: Convert to assembly pseudocode

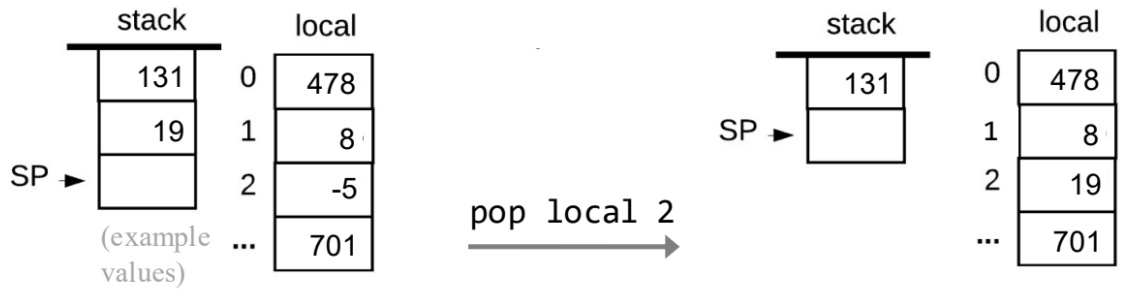
2: Convert to assembly



Abstraction: *push/pop local i*

Local variables

Abstraction



pop local i pops the value at the working stack and stores it in local i

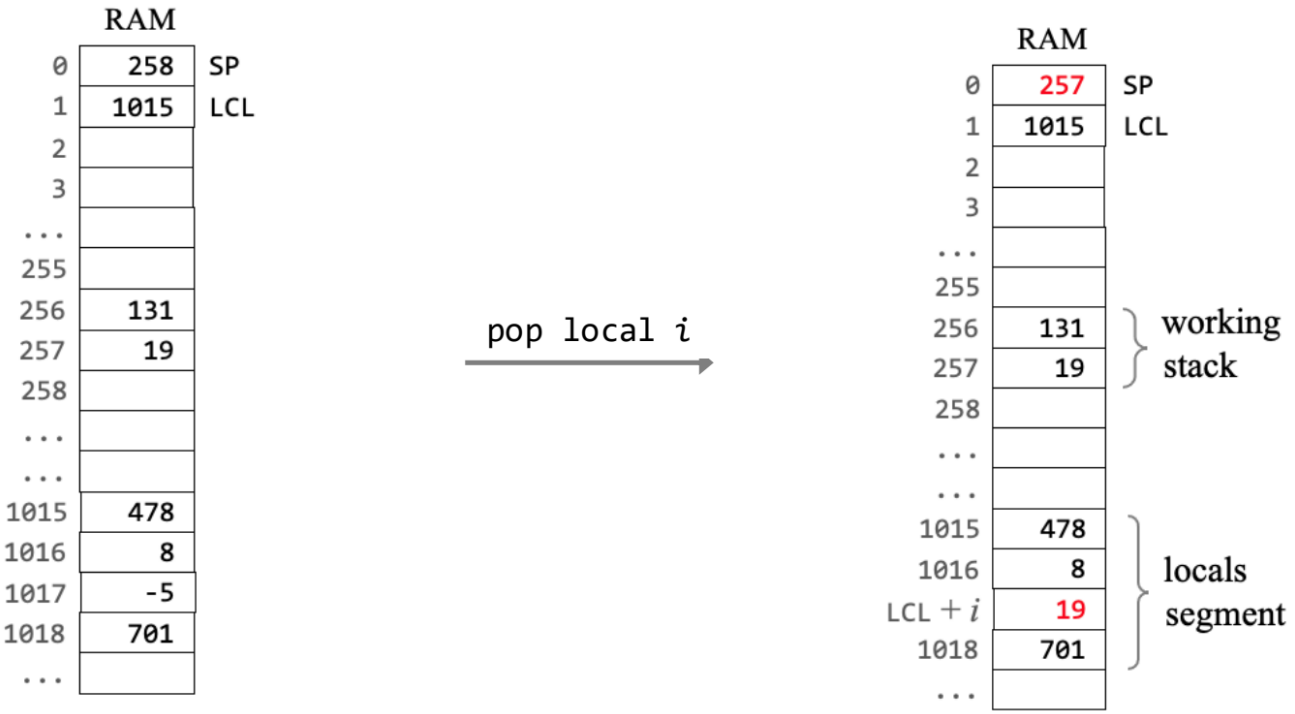
The local stack pointer (LCL is RAM[1]) stores the local stack segment location

Implementation

locals segment: stored somewhere in the RAM;

LCL = base address

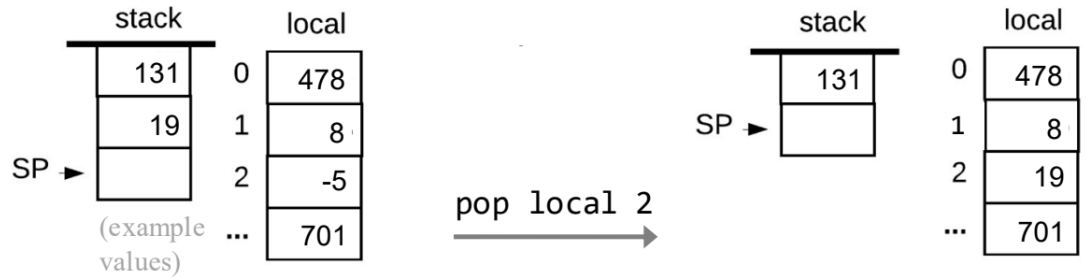
(1015 is an example)



Write the Hack Assembly code

Exercise: Local variables

Abstraction

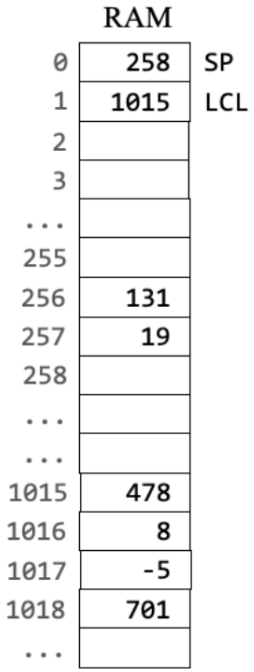


Implementation

locals segment:
stored somewhere in the RAM;

`LCL` = base address

(1015 is an example)



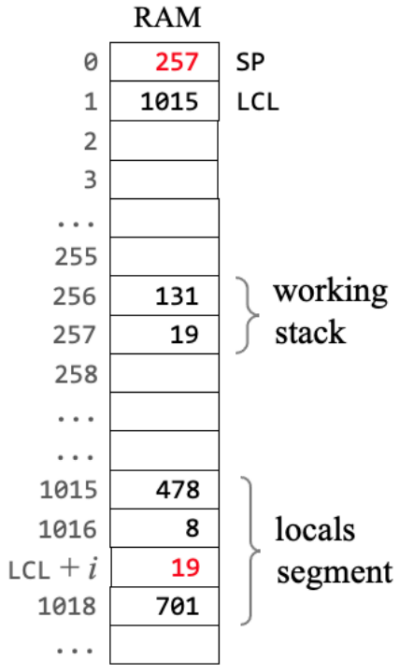
Pseudocode

```
// pop local i
addr ← LCL + i
SP--
RAM[addr] ← RAM[SP]
```

pop local i

Hack assembly

You do it!



Summary: Local variables

Abstraction

VM code

```
pop local i
```

```
push local i
```

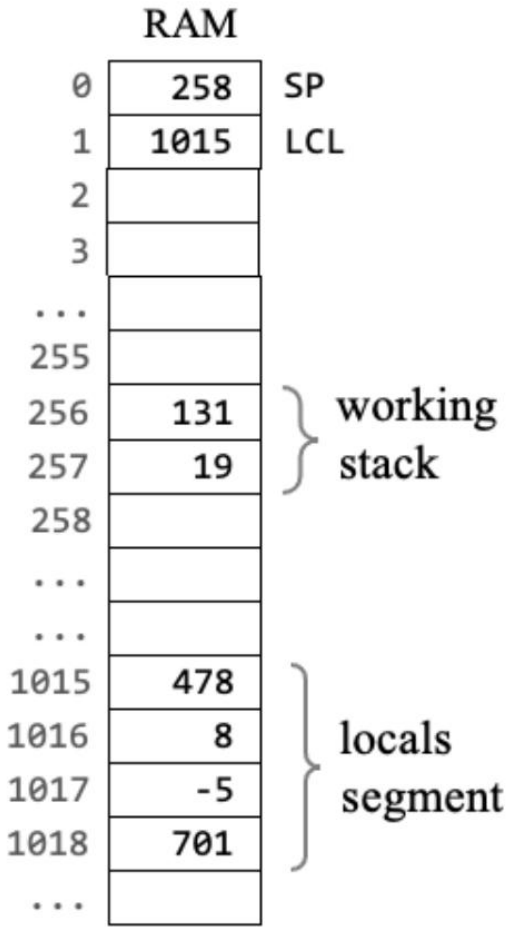
VM translator

Implementation

Assembly pseudo code

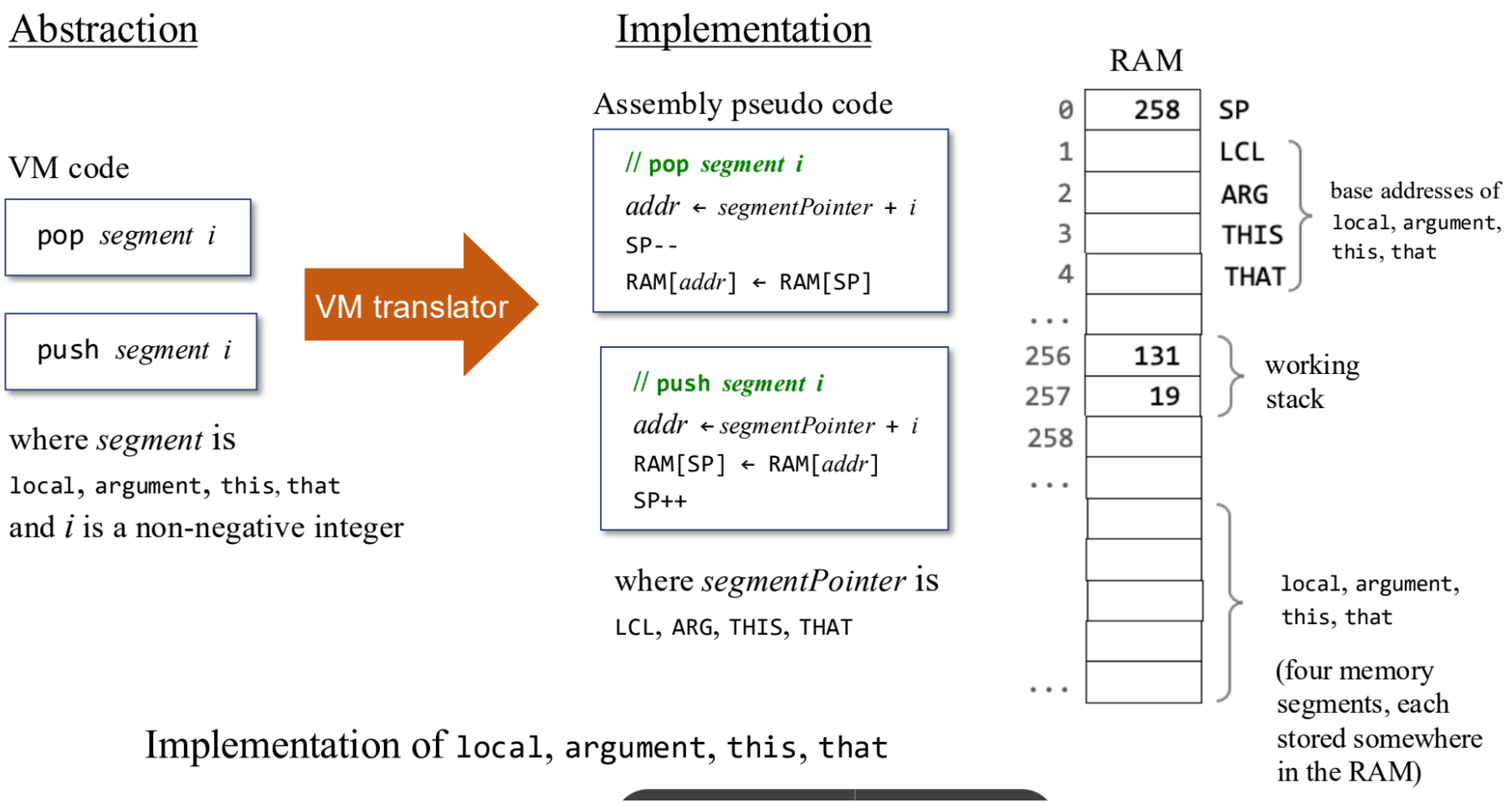
```
// pop local i
addr ← LCL + i
SP--
RAM[addr] ← RAM[SP]
```

```
// push local i
addr ← LCL + i
RAM[SP] ← RAM[addr]
SP++
```



Arguments, this, that, and local are all implemented the same way

Implementation of local, argument, this, that



Implementation of `local`, `argument`, `this`, `that`

Abstraction: *push/pop static i*

Static variables

Location of class static variables

Standard mapping (contract)

The `static` segment is stored in a fixed RAM block, starting at address 16 and ending at address 255

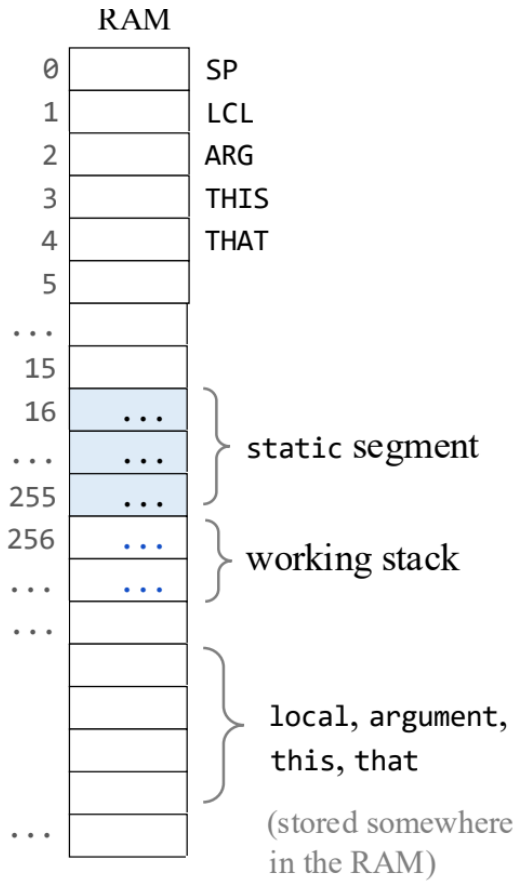
To translate `push/pop static i`

(when translating a VM file named `Xxx.vm`)

Generate assembly code that realizes:

`push/pop Xxx.i`

(Explanation: When this assembly code will be further translated to executable code, the Assembler will map these variables on RAM addresses 16, 17, 18, ..., exactly what we want).



More Later...

Abstraction: *push/pop temp i*

Temporary variables

Sometimes we will need to define temporary variables that are not defined by the original code.

Stored in a fixed-size 8 variable stack, corresponding to *temp 0, temp 1, ..., temp 7*

The temporary stack segment is stored at RAM[5], Ram[6], ..., Ram[12]

Standard mapping (contract)

The `temp` segment is stored in a fixed RAM block, starting at address 5 and ending at address 12:

`temp 0` is stored in `RAM[5]`

`temp 1` is stored in `RAM[6]`

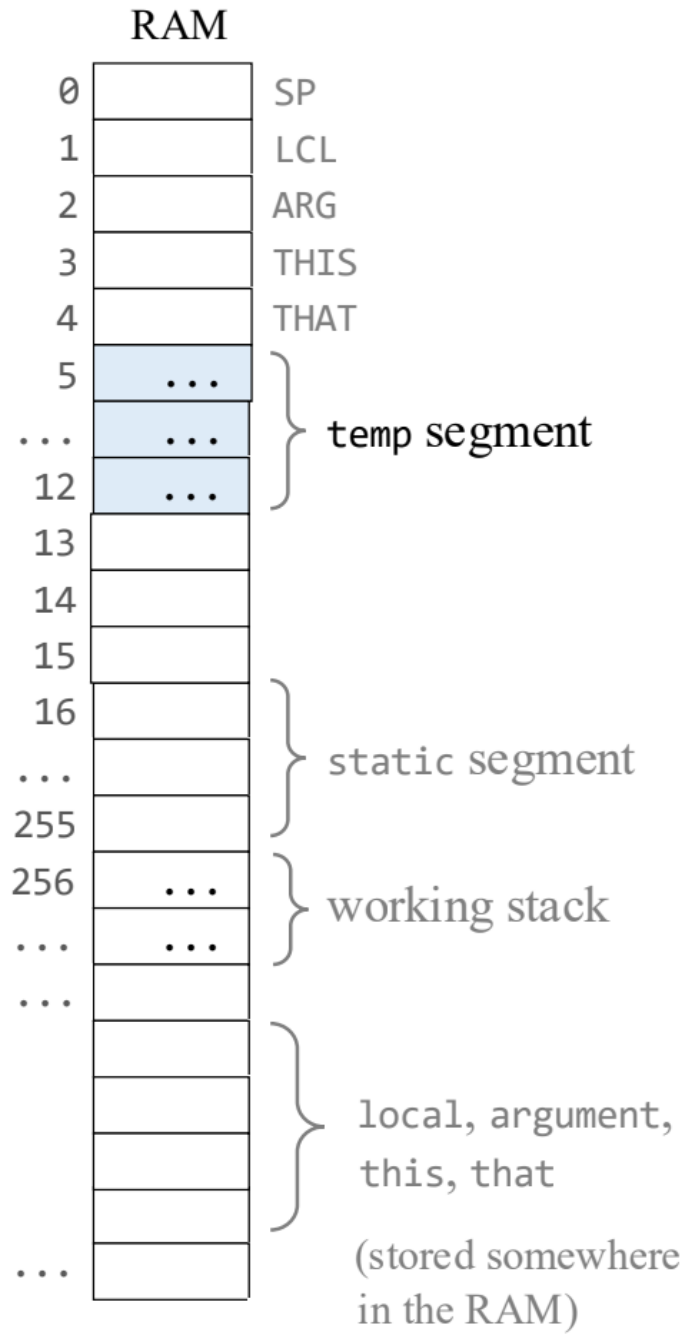
...

`temp 7` is stored in `RAM[12]`

Implementing `push/pop temp` i

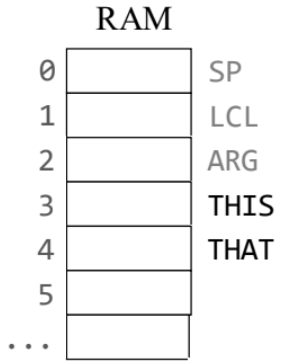
Generate assembly code that realizes:

`push/pop RAM[5 + i]`



Abstraction: *push/pop pointer i*

Pointer variables



Abstraction

`pointer`: A two-element segment, containing the base addresses of segments `this` and `that`

Implementation

(a truly virtual segment, not stored anywhere)

To translate: `push/pop pointer 0`

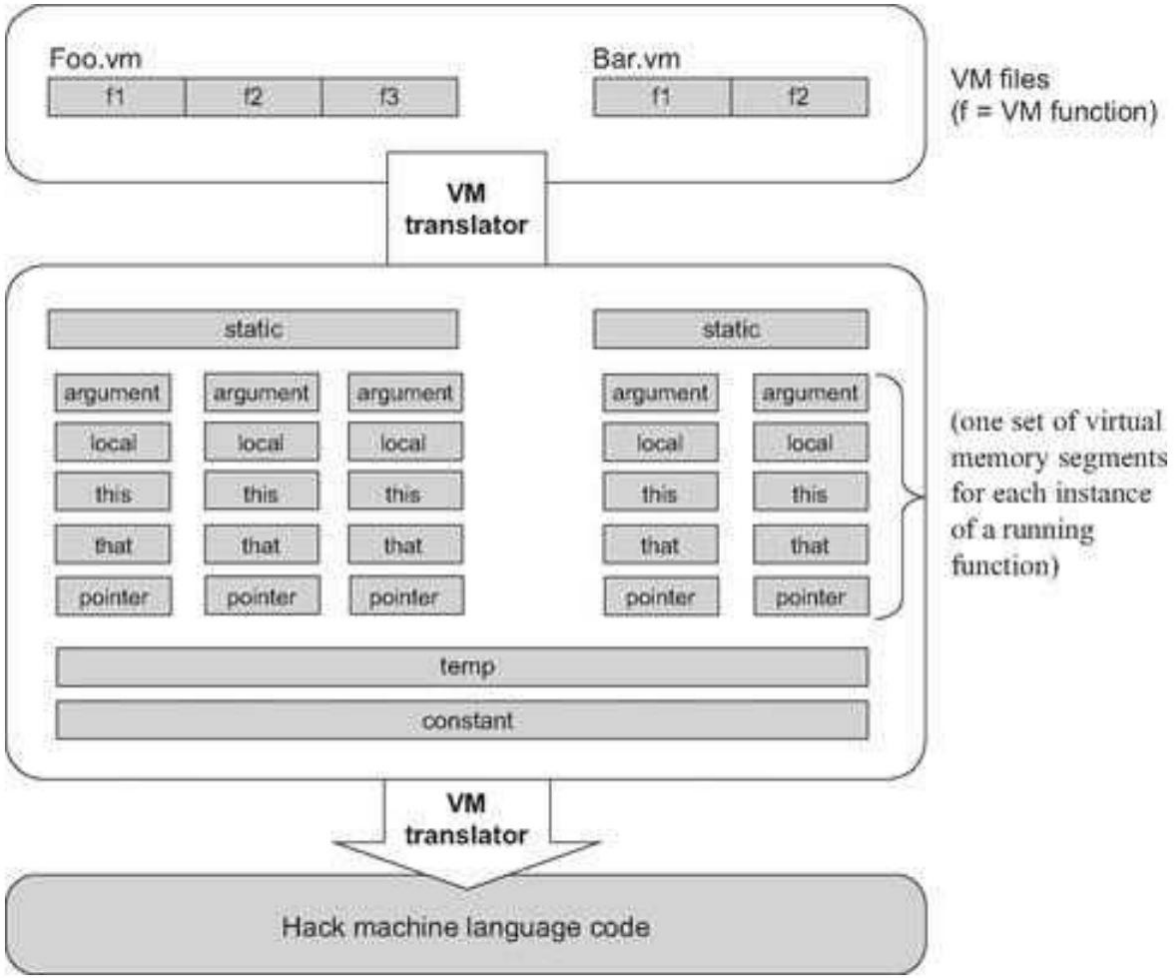
generate assembly code that realizes `push/pop THIS`

To translate: `push/pop pointer 1`

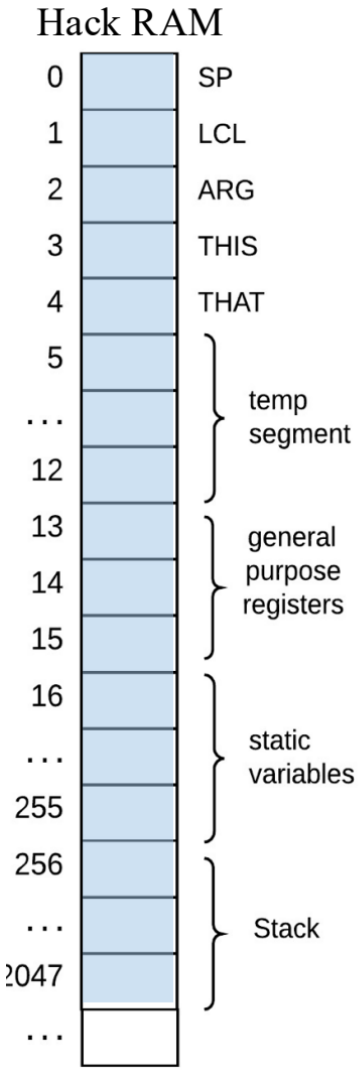
generate assembly code that realizes `push/pop THAT`

Used for objects and arrays (more later)…

Virtual memory segments for multiple files



Summary: Memory Access Commands



push segment index

- Push the value of segment[index] onto the stack.

pop segment index

- Pop the top stack value and store it in segment[index].

where *segment* ϵ {*temp, static, this, that, local, constant, argument, pointer*}

Exercise: Write VM code

Suppose x1 = 10 and y1 = 2

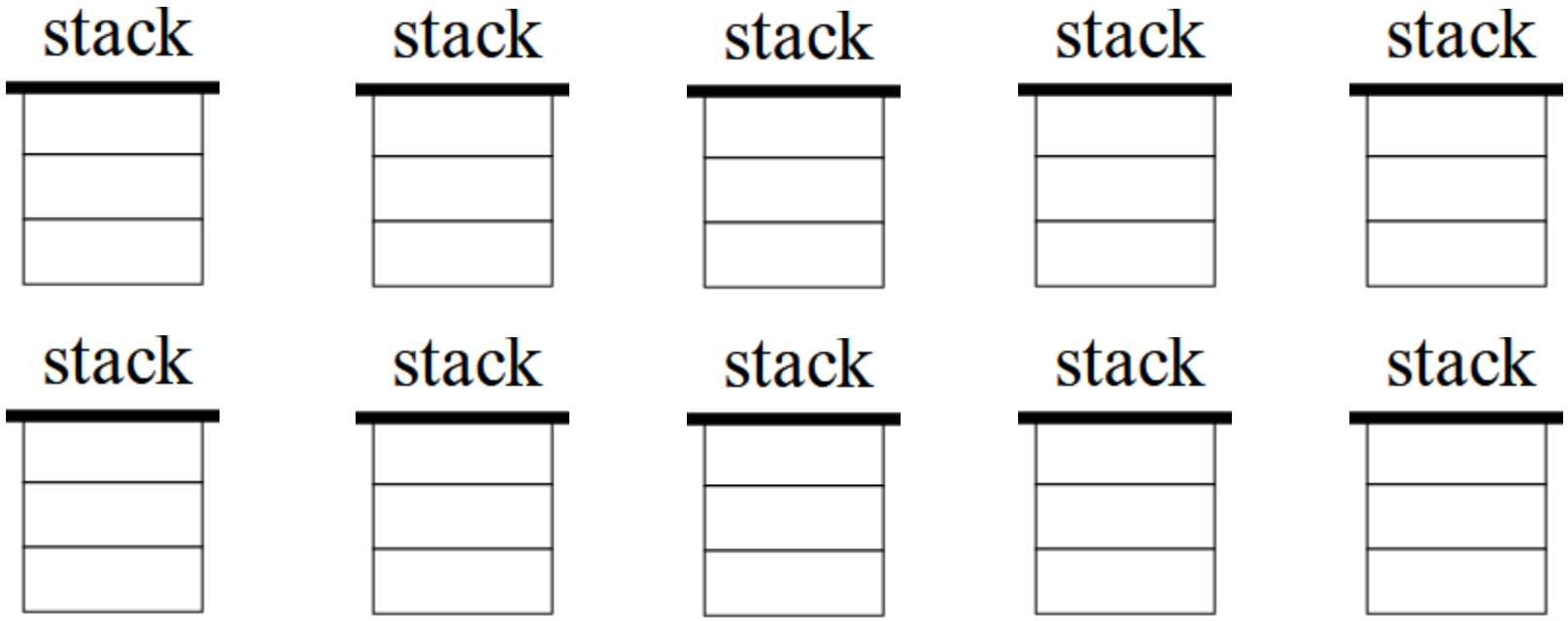
What virtual memory segments do we need?

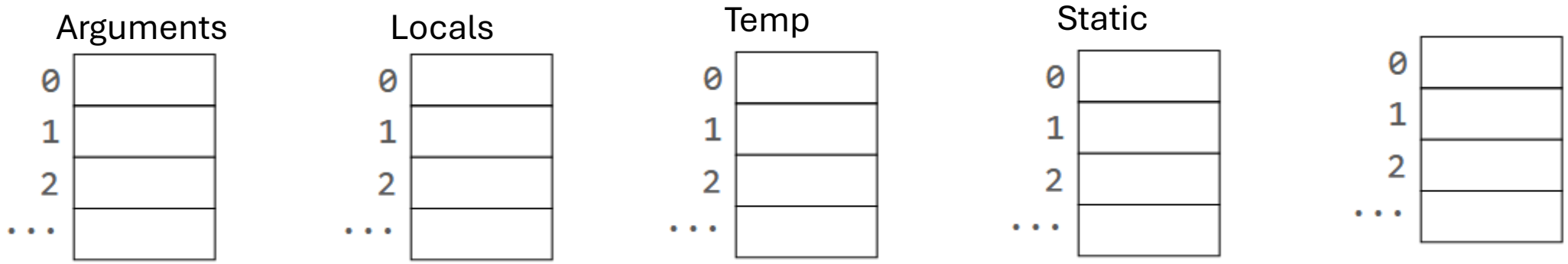
```
static int factor = 2;

int foo(int x1, int y1)
{
   int delta = y1 – x1;
   int tmp = factor + delta + 4;
   ....

}
```

Exercise: Visualize the VM code execution

Suppose argument0 = 10 and argument1 = 2

```
push argument 1
push argument 0
sub
pop local 0
push static 0
push local 0
push constant 4
add
add
pop local 1
```





Exercise: Convert to hack pseudocode

```
push argument 1
push argument 0
sub
pop local 0
```

Exercise: Convert to hack assembly

// sub
SP--
D ← RAM[SP]
SP--
D ← RAM[SP] – D
RAM[SP] ← D
SP++