Lecture 7

Virtual Machine, Part I

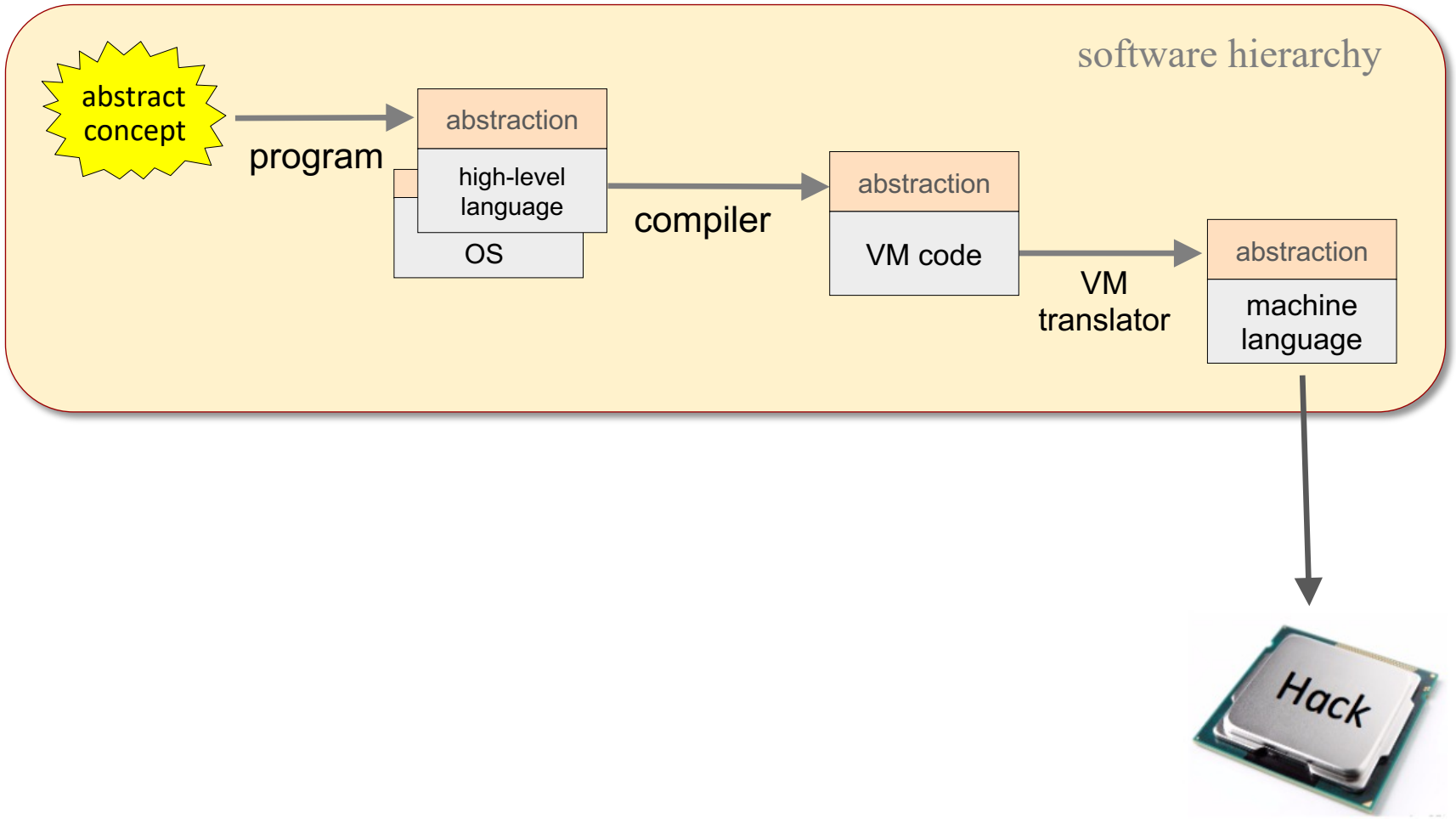
These slides support chapter 7 of the book

The Elements of Computing Systems

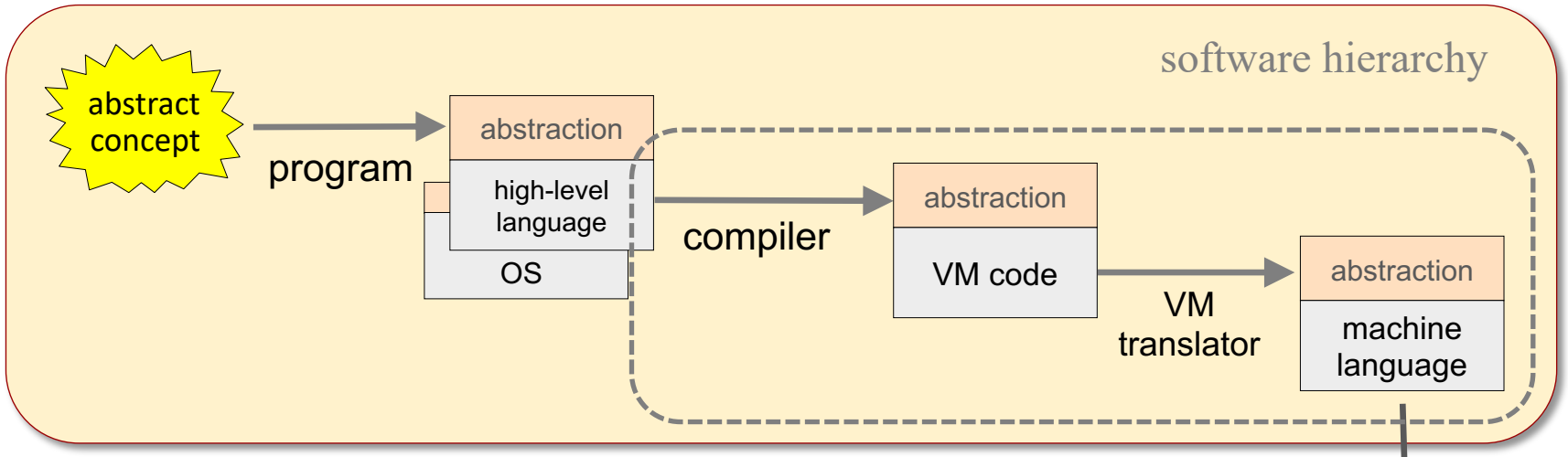
By Noam Nisan and Shimon Schocken

MIT Press, 2021

Nand to Tetris Roadmap: Part II



Nand to Tetris Roadmap: Part II

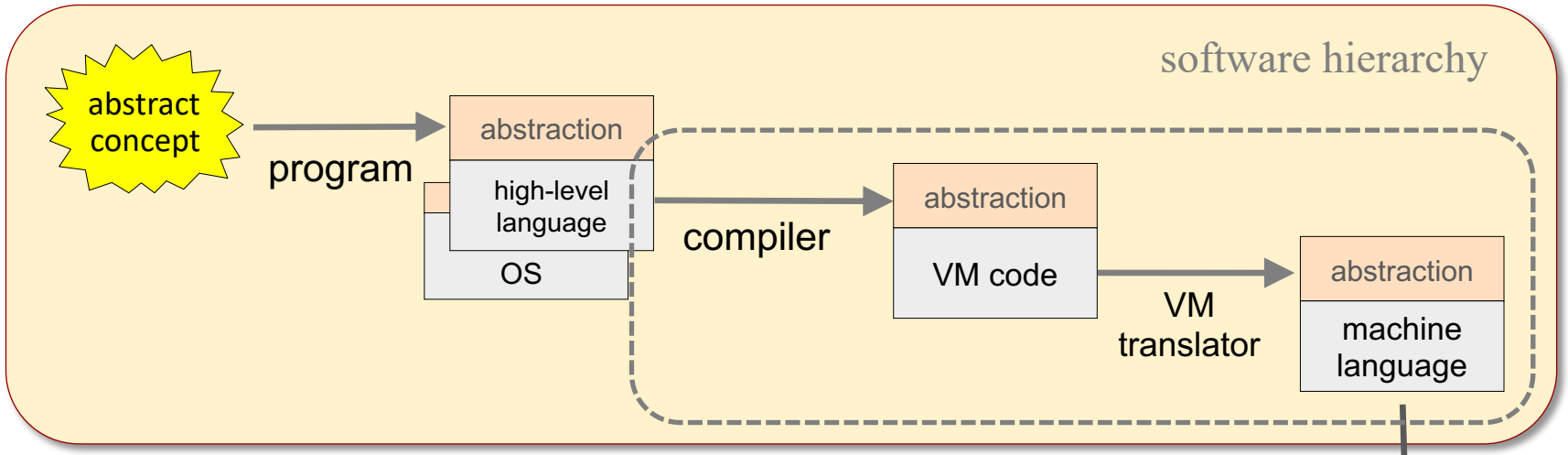


The VM code that the compiler generates is designed to operate on an *abstract stack machine*

The VM Translator translates this code into machine language

In other words: The VM translator realizes / implements the abstract stack machine on the host platform.

Nand to Tetris Roadmap: Part II



Lecture goals

- Understanding the VM *abstraction* (VM code)
- Building a VM *implementation* (VM translator)

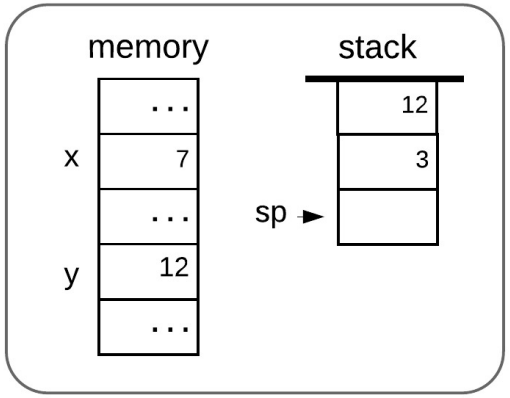
Stack

Basic operations

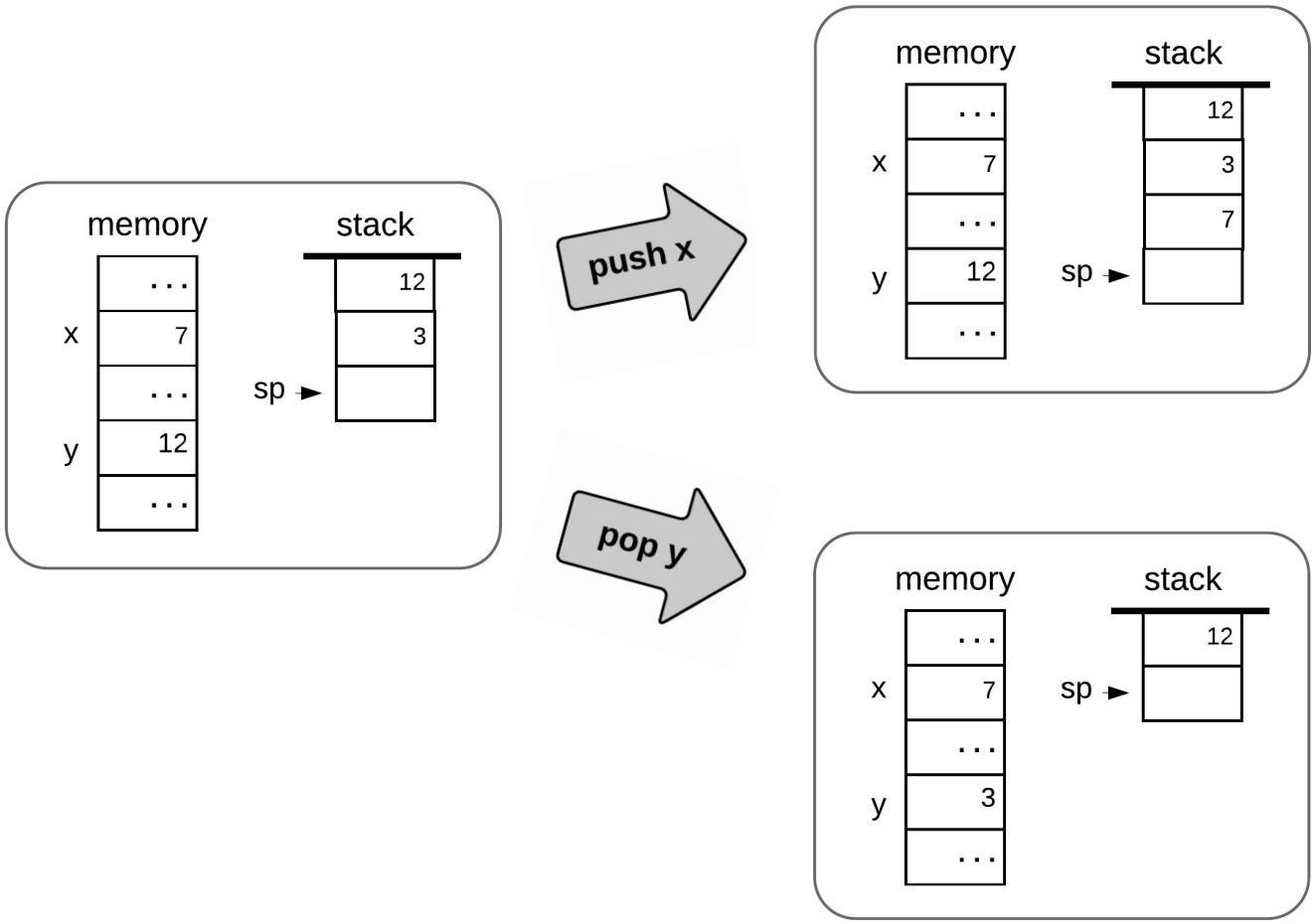
`push`: adds an element at the stack's top

`pop`: removes the top element

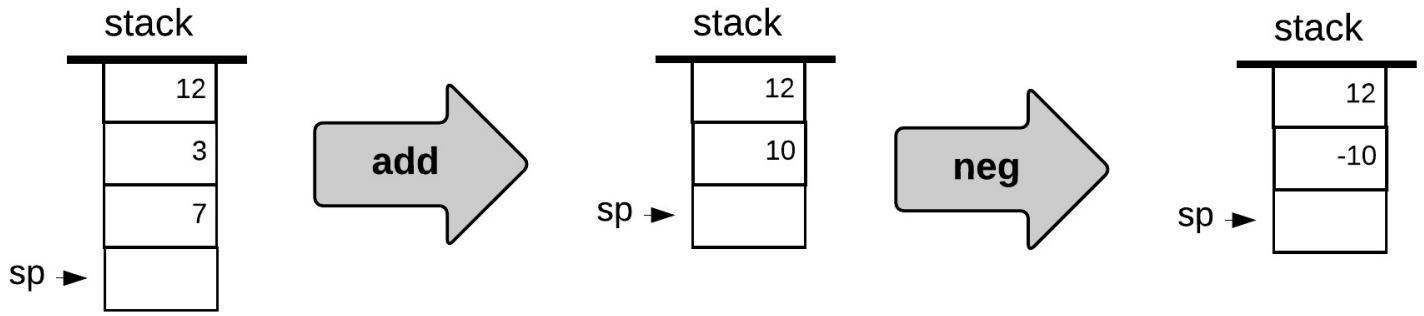
Stack



Stack



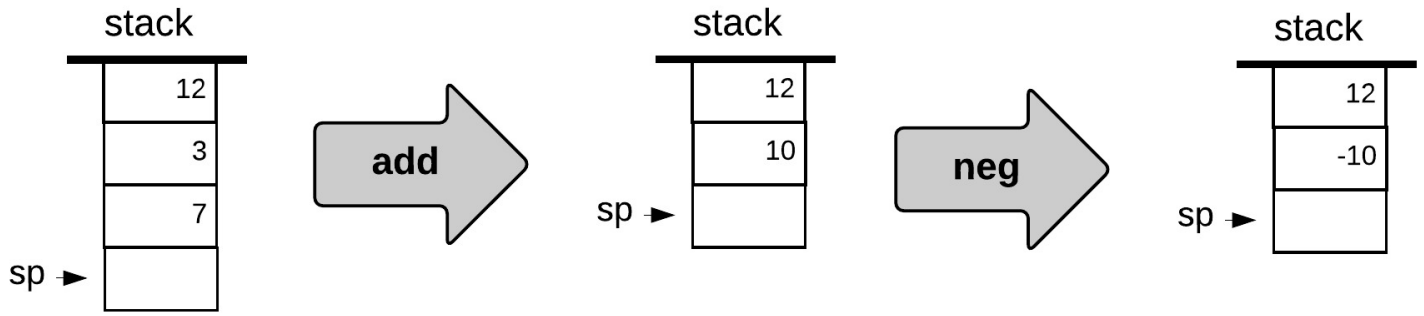
Stack arithmetic



Applying a function *f* (that has *n* arguments)

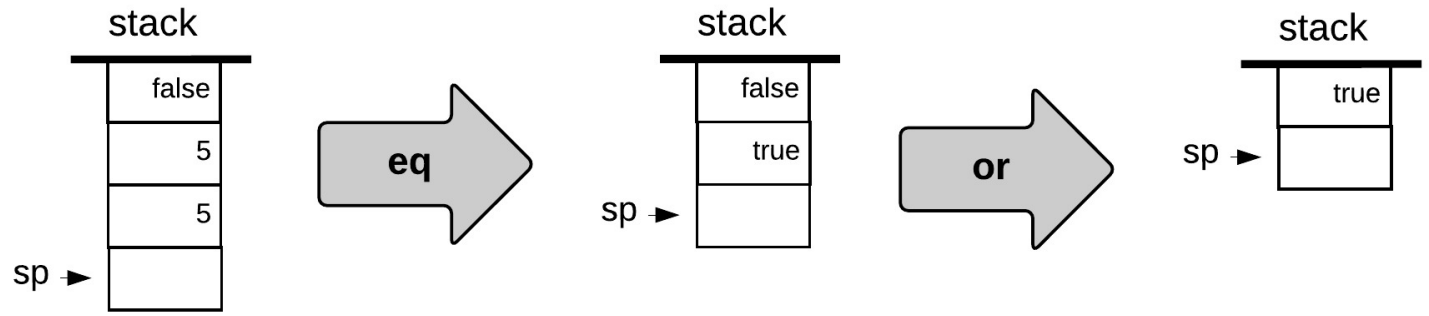
- pops *n* values (arguments) from the stack,
- Computes *f* on the values,
- Pushes the resulting value onto the stack.

Stack arithmetic



Applying a function *f* (that has *n* arguments)

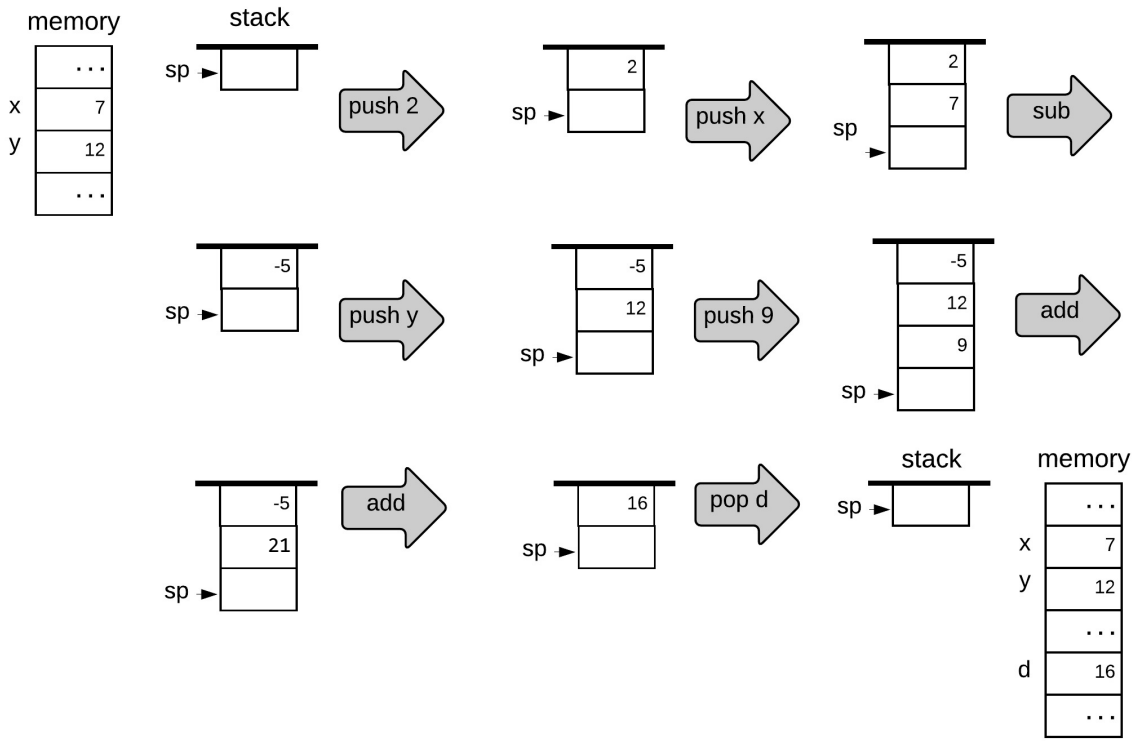
- pops *n* values (arguments) from the stack,
- Computes *f* on the values,
- Pushes the resulting value onto the stack.



Arithmetic operations

VM pseudocode (example)

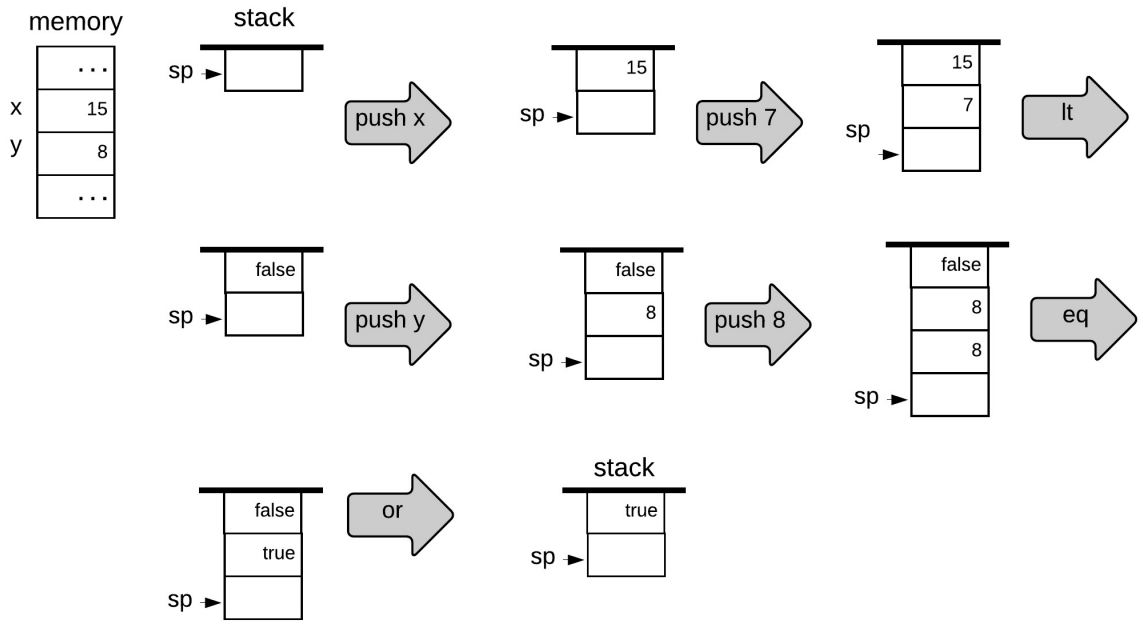
```
// d = (2 - x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Logical operations

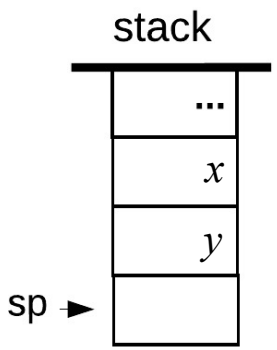
VM pseudocode (example)

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



The VM language: Arithmetic / Logical operations

Command	Return value	Return value
`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.

The big picture: Compilation

VM code is typically written by compilers;

Every high-level arithmetic-logical expression can be translated into a sequence of operations on a stack, using these VM commands.

The VM language

Push / pop commands

- `push` *segment i*
- `pop` *segment i*

Arithmetic / Logical commands

- `add`, `sub` , `neg`
- `eq` , `gt` , `lt`
- `and`, `or` , `not`

Branching commands

- `label` *label*
- `goto` *label*
- `if-goto` *label*

Function commands

- `Function` *functionName nVars*
- `Call` *functionName nArgs*
- `return`

The big picture: Compilation

Source code (Jack)

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

Compiled VM code

```
...
...
...
...
...
...
...
...
...
```

The big picture: Compilation

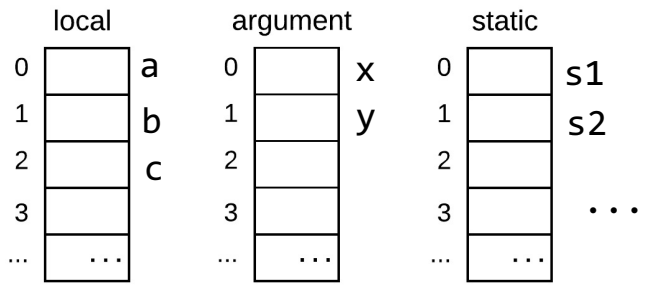
Source code (Jack)

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

compiler

Compiled VM code

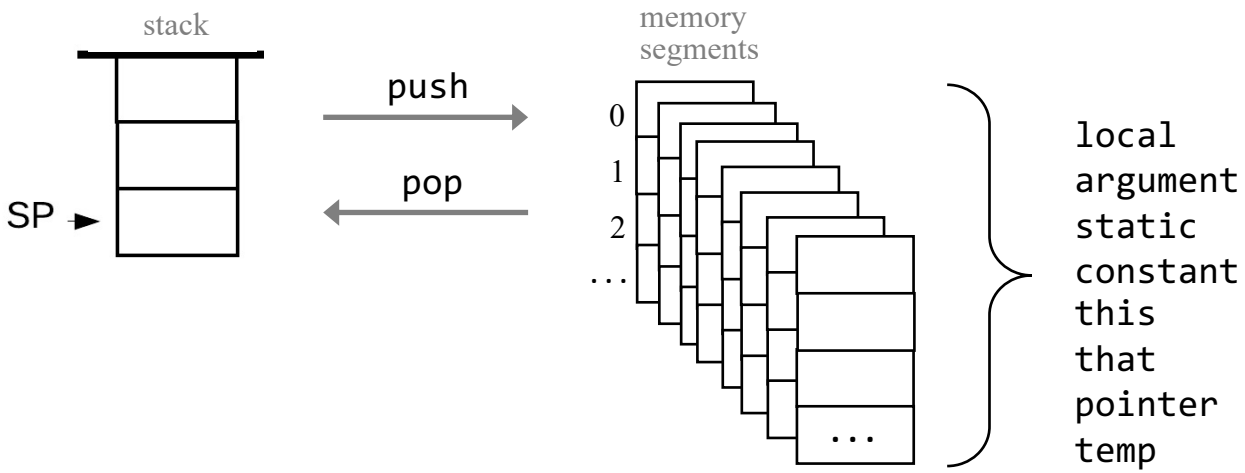
```
...
...
...
...
push static 0
push argument 1
add
pop local 2
...
```



virtual memory segments

In the VM abstraction, each variable is represented as an entry in a virtual memory segment dedicated to the variable's *kind:* `local`, `argument`, `static`.

Memory segments



Our VM abstraction features 8 virtual memory segments;

The role of each segment: Later.

All segments are accessed the same way:

`push` / `pop` *segment i*

where *i* is a non-negative integer and *segment* is `local`, `argument`, `static`, ..., `temp`

The VM language

✓ Push / pop commands

- `push` *segment i*
- `pop` *segment i*

✓ Arithmetic / Logical commands

- `add`, `sub` , `neg`
- `eq` , `gt` , `lt`
- `and`, `or` , `not`

Project 7

Question

How can we implement this VM abstraction?

Answers

Hardware implementation: We can extend the computer's hardware with modules that represent the stack and the virtual memory segments, and extend the computer's instruction set with versions of the VM commands; We also have to extend the CPU

Software implementation: We can write a program in a high level language in which the stack and the virtual memory segments are implemented as arrays, and the VM commands as methods that operate on these arrays

Compilation: We can implement the stack and the virtual memory segments as memory blocks in a host RAM, and translate each VM command into machine language instructions that operate on these blocks

The VM language

Push / pop commands

- `push` *segment i*
- `pop` *segment i*

Arithmetic / Logical commands

- `add`, `sub` , `neg`
- `eq` , `gt` , `lt`
- `and`, `or` , `not`

Project 7

Question

How can we implement this VM abstraction?

Answer

Hardware implementation: We can extend the computer's hardware with modules that represent the stack and the virtual memory segments, and extend the compute[illegible]he VM comman[illegible]

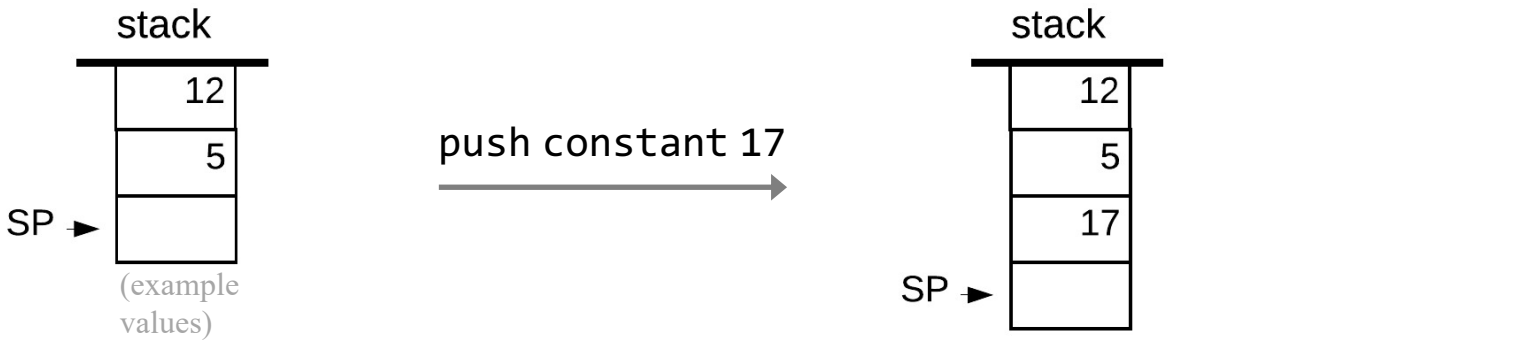
Let's start with implementing the push / pop commands

Softwar[illegible]program in a high level language in which the stack and the virtual memory segments are implemented as arrays, and the VM commands as methods that operate on these arrays

Compilation: We can implement the stack and the virtual memory segments as memory blocks in a host RAM, and translate each VM command into machine language instructions that operate on these blocks

Implementing `push constant` *i*

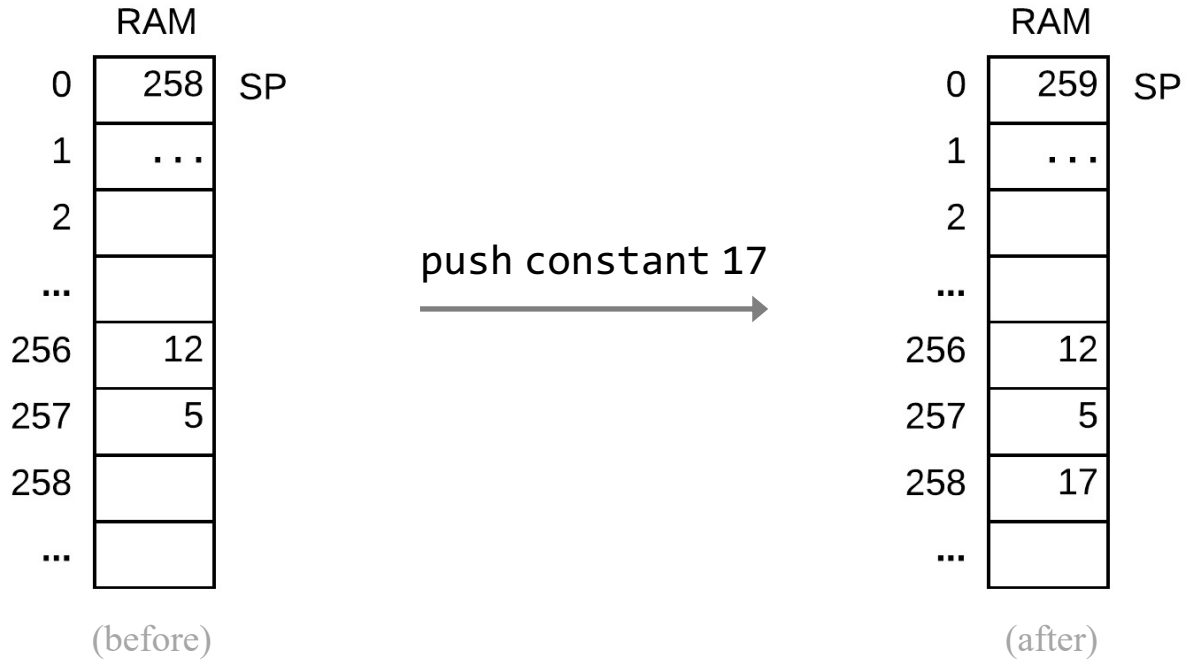
Abstraction



Implementation

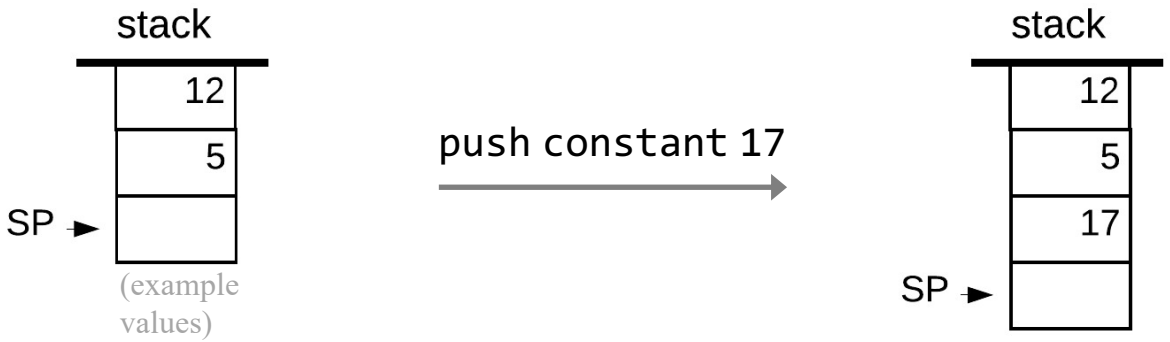
Assumed convention:
The *stack* is stored in the RAM, from address 256 onward

The *stack pointer* is stored in `RAM[0]`



Implementing `push constant i`

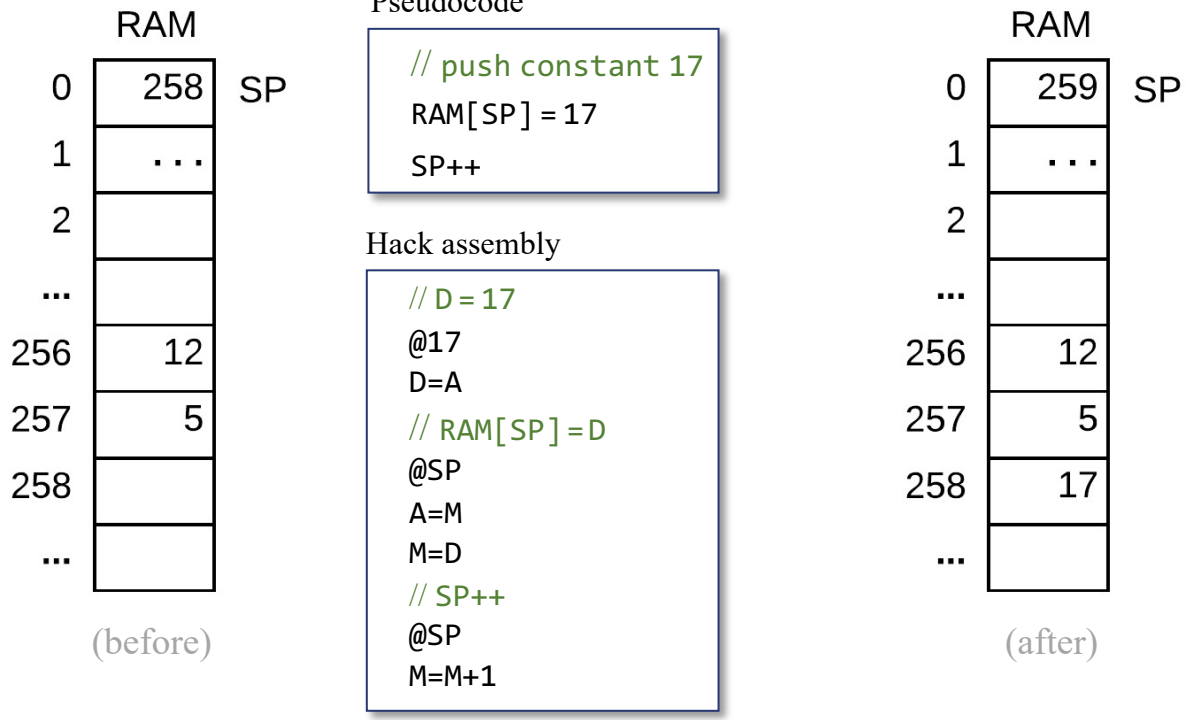
Abstraction



Implementation

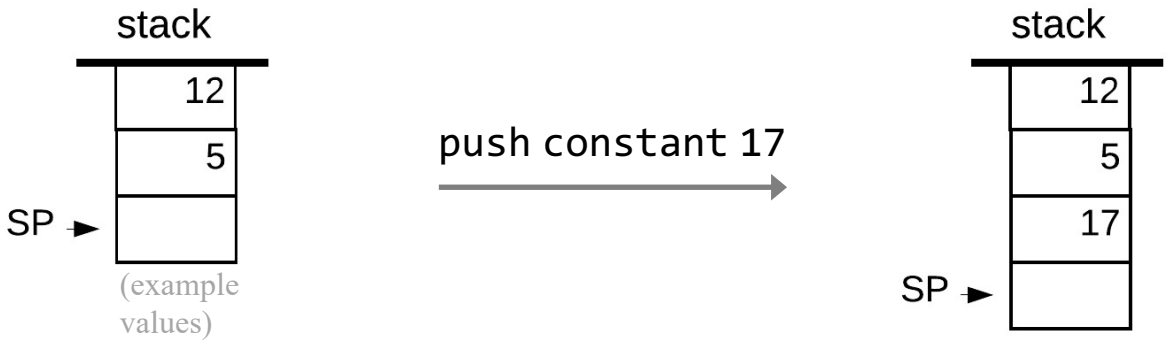
Assumed convention:
The *stack* is stored in the RAM, from address 256 onward

The *stack pointer* is stored in `RAM[0]`



Implementing `push constant i`

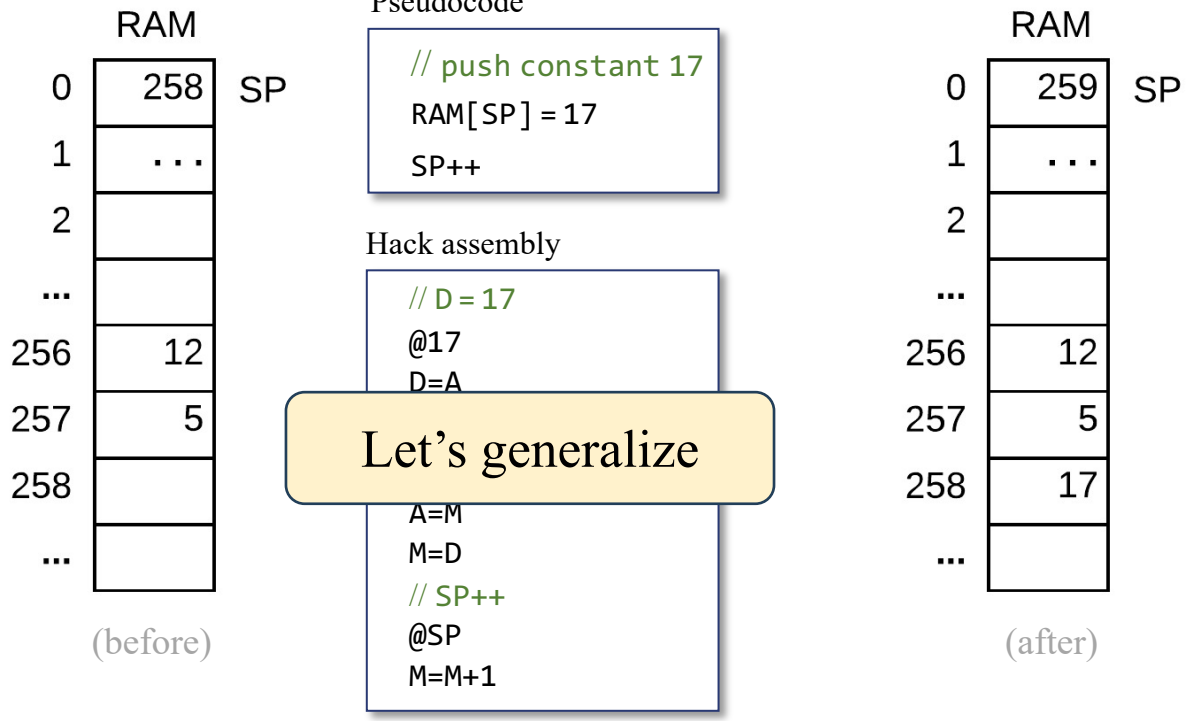
Abstraction



Implementation

Assumed convention:
The *stack* is stored in the RAM, from address 256 onward

The *stack pointer* is stored in `RAM[0]`



Implementing `push constant i`

Abstraction

VM code

```
push constant i
```

VM translator

Implementation

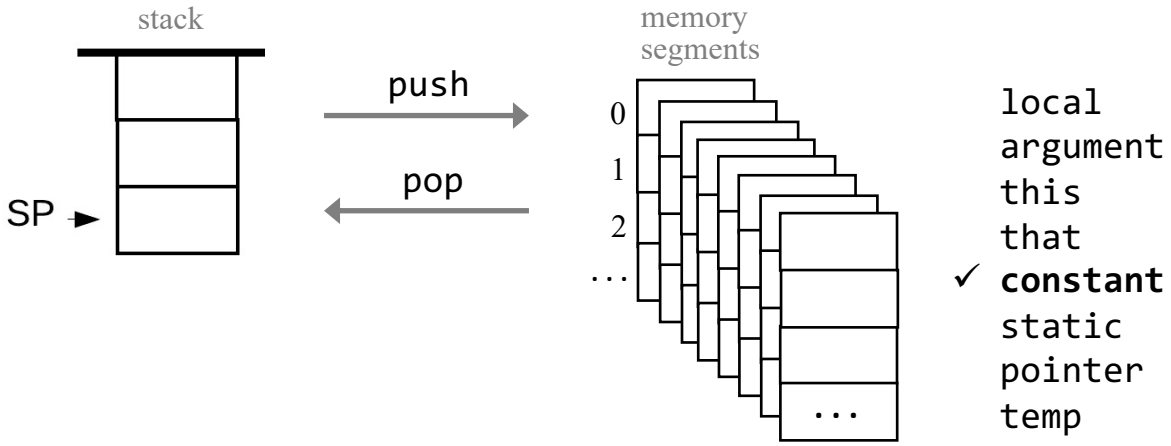
Assembly code

```
// D = i
@i
D=A
// RAM[SP] = D
@SP
A=M
M=D
// SP++
@SP
M=M+1
```

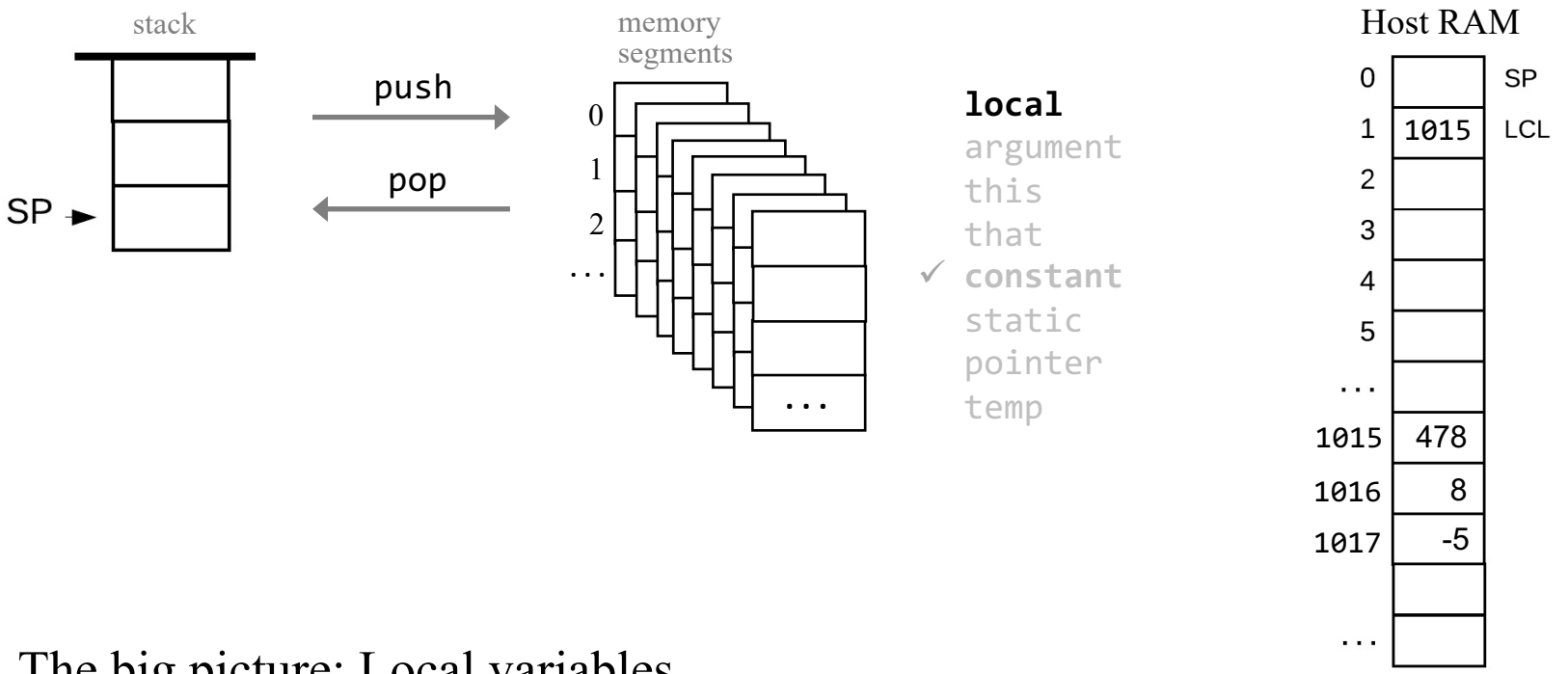
Notes

1. `constant` is not a "real segment"
   (used for VM syntax consistency)
2. There is no `pop` *constant i* command

Implementing `push/pop`



Implementing `push/pop local i`



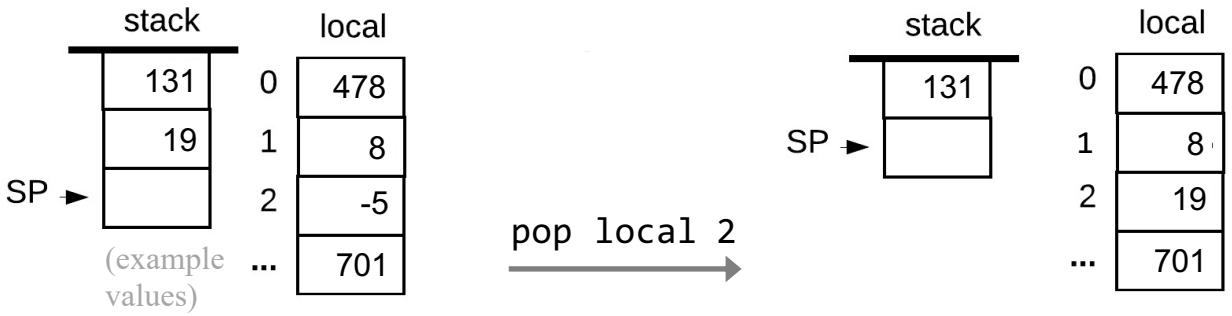
The big picture: Local variables

Abstraction: `local 0`, `local 1`, `local 2`, ...

Implementation: A RAM block whose base address is kept in a pointer named `LCL`

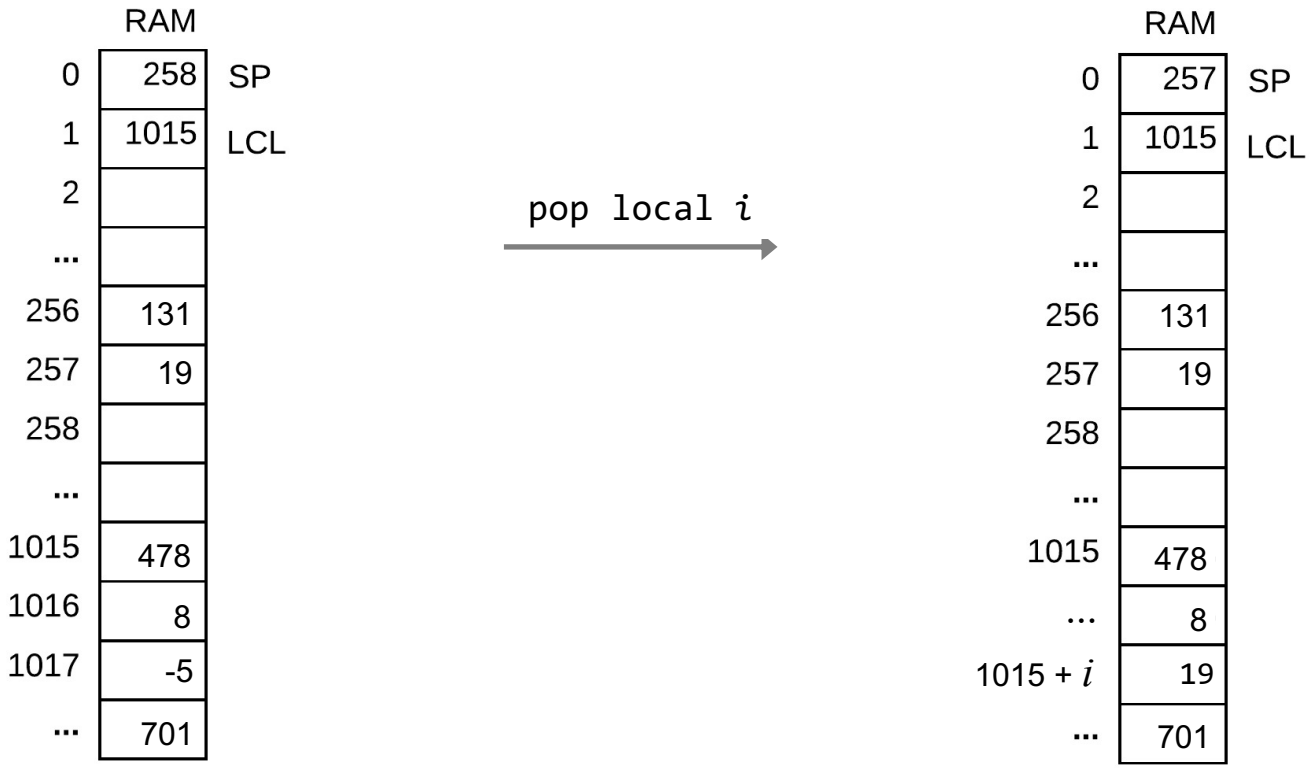
Implementing `push/pop local i`

Abstraction



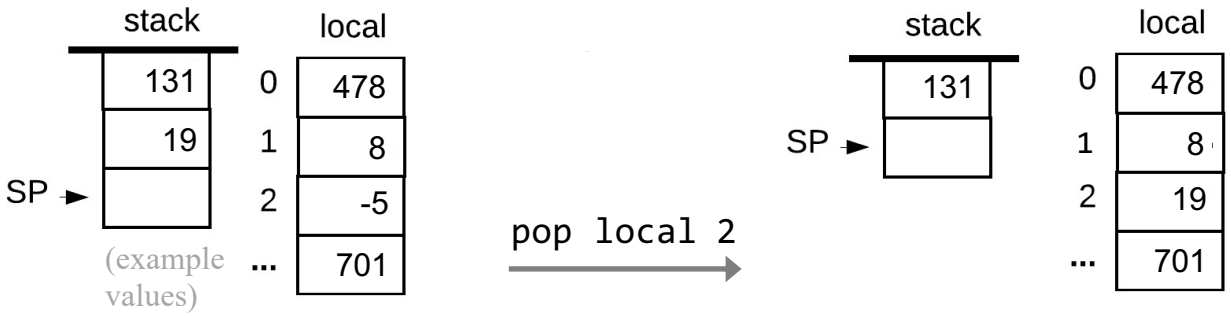
Implementation

Here, the base address of the RAM block that represents the local variables happens to be 1015 (arbitrary example)



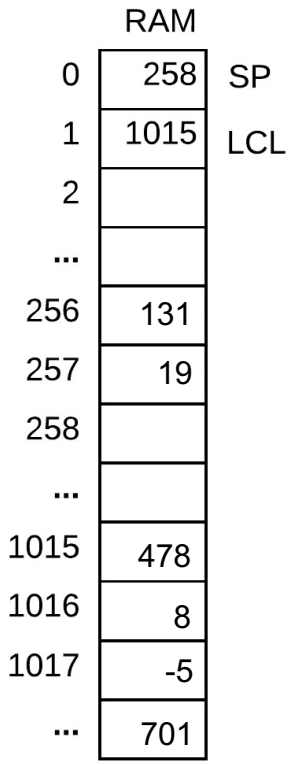
Implementing `push/pop local i`

Abstraction



Implementation

Here, the base address of the RAM block that represents the local variables happens to be 1015 (arbitrary example)

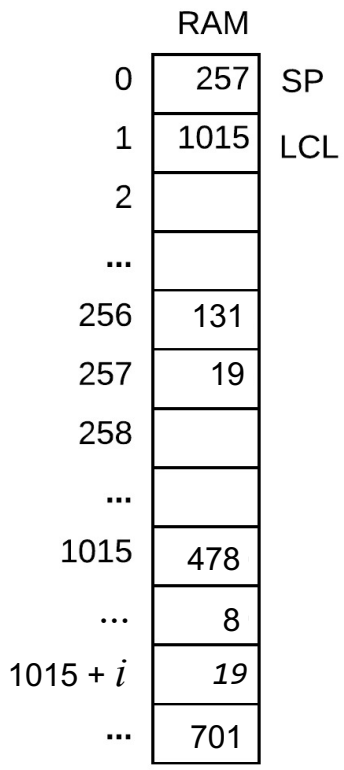


Pseudocode

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

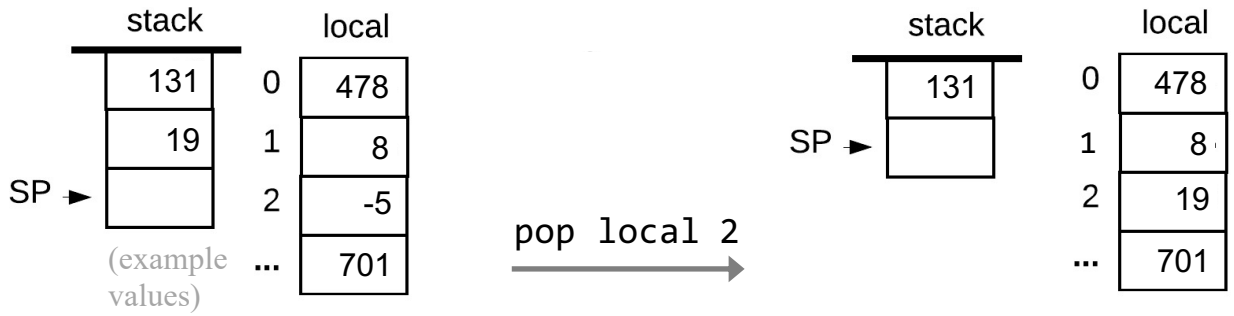
Hack assembly

You figure it out!



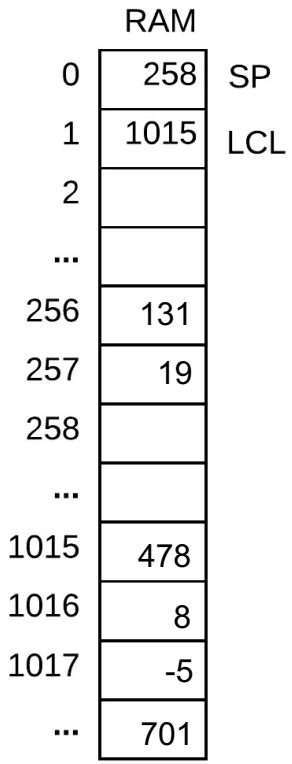
Implementing `push/pop local i`

Abstraction



Implementation

Here, the base address of the RAM block that represents the local variables happens to be 1015 (arbitrary example)

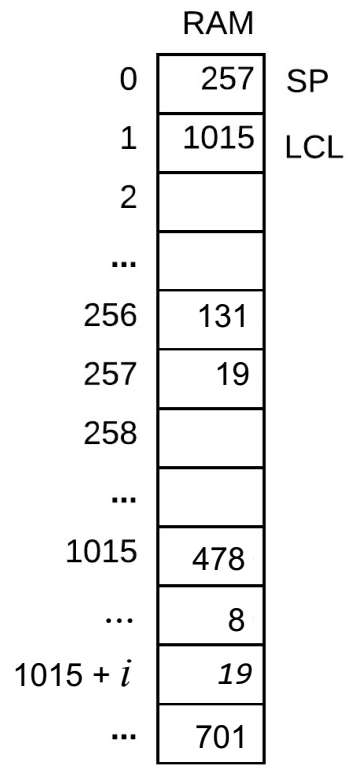


Pseudocode

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

Hack assembly

Let's generalize



Implementing `push/pop` {`local` | `argument` | `this` | `that`} *i*

VM code

`pop` *segment i*

VM translator

Assembly pseudo code

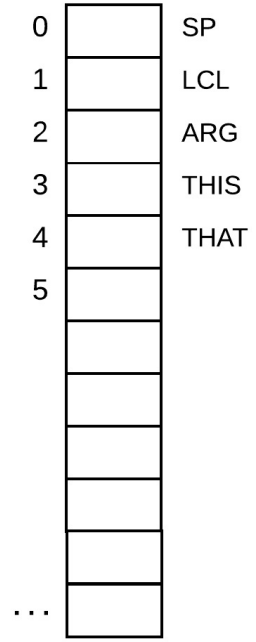
```
// pop segment i
addr ← segmentPointer + i
SP--
RAM[addr] ← RAM[SP]
```

`push` *segment i*

VM translator

```
// push segment i
addr ← segmentPointer + i
RAM[SP] ← RAM[addr]
SP++
```

Host RAM

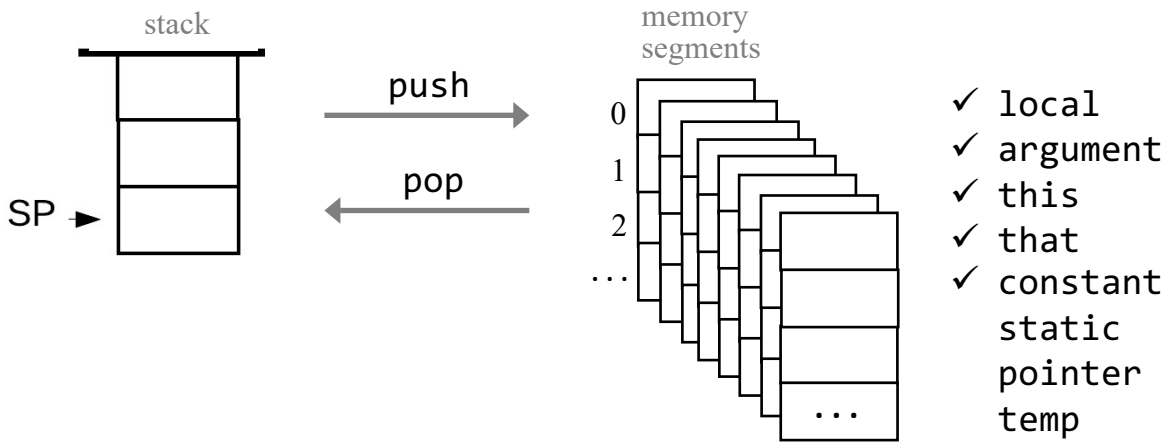


This pseudo code handles all the VM commands

`push` / `pop` *segment i*

where *segment* is `local`, `argument`, `this`, or `that`,
segmentPointer is `LCL`, `ARG`, `THIS`, or `THAT`, *and*
i is a non-negative integer.

Implementing `push/pop`



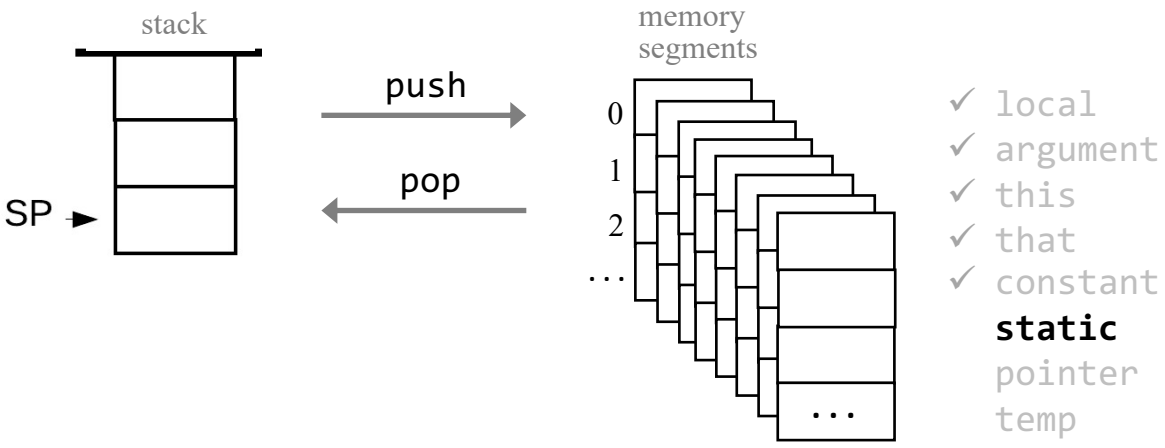
The big picture / Recap

When the compiler compiles a high-level method, it generates code that represents…

- the method's *local variables* in the virtual segment `local`
- the method's *arguments* in the virtual segment `argument`
- the *fields* of the current object in the virtual segment `this`
- the *elements* of the current array in the virtual segment `that`

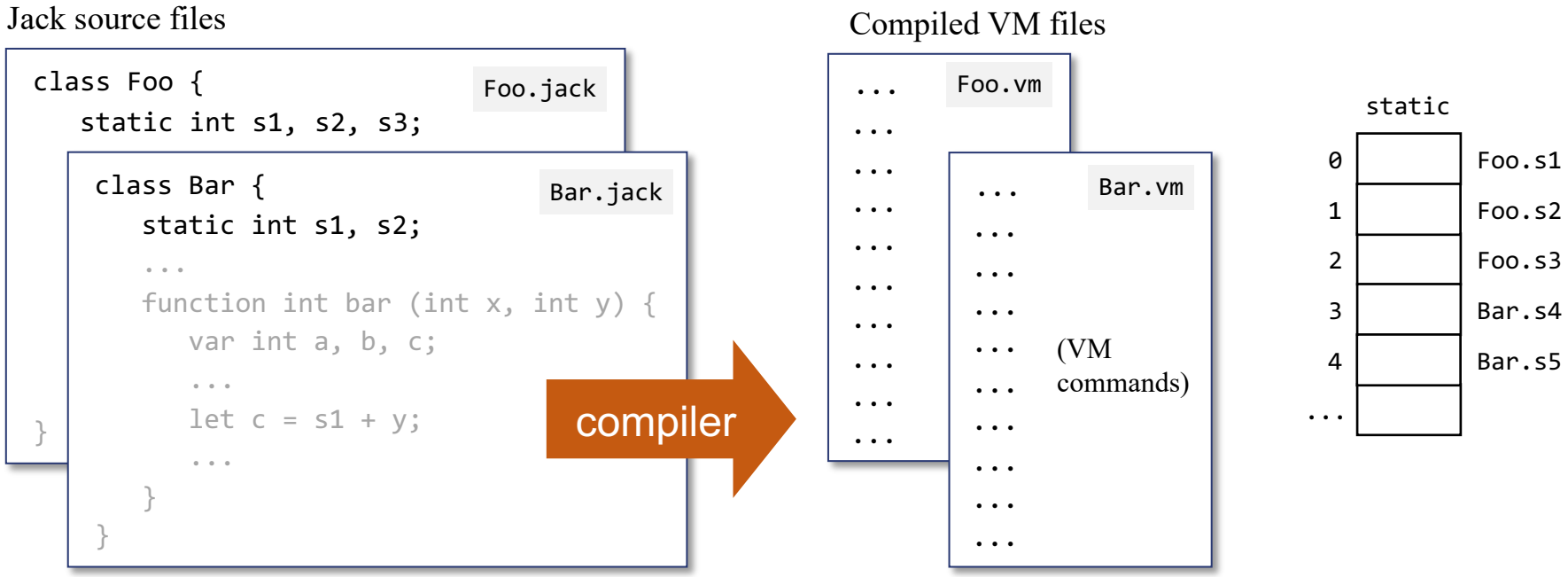
All abstracted and implemented on the VM exactly the same way

Implementing `push/pop static i`



- ✓ `local`
- ✓ `argument`
- ✓ `this`
- ✓ `that`
- ✓ `constant`
- **`static`**
- `pointer`
- `temp`

Implementing `push/pop static i`



The Big Picture: Static variables

The compiler maps the *static variables* of all the classes onto one VM segment named `static`

Implementing `push/pop static` *i*

Implementation

Assumed convention: The `static` segment is stored in a fixed RAM block, starting at address 16;

Compilation trick:

When translating a VM file named `Foo.vm`,
we translate each VM command: `push/pop static` *i* into
into assembly code that realizes: `push/pop Foo.`*i*

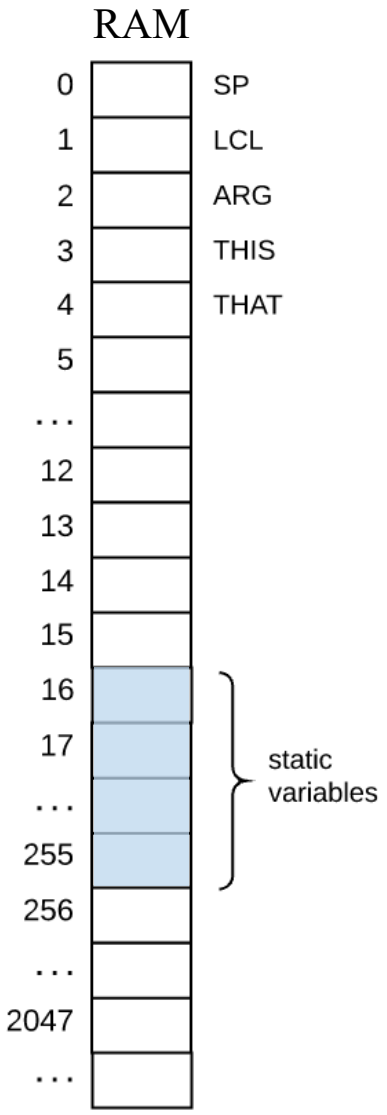
Explanation

(why this works on the Hack platform):

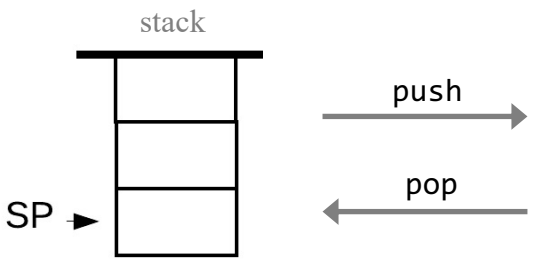
The VM code generated by the VM translator will be translated further by the Hack assembler into Hack instructions;

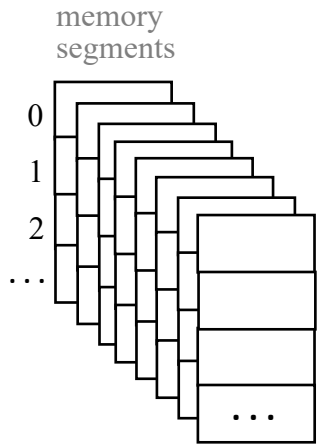
And, by convention, the Hack assembler maps every symbolic variable `Foo.`*i* onto `RAM[16]`, `RAM[17]`, ..., `RAM[255]`

(exactly what we want).

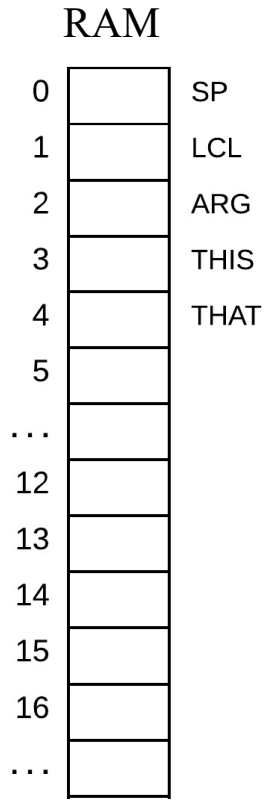


Implementing `push/pop`

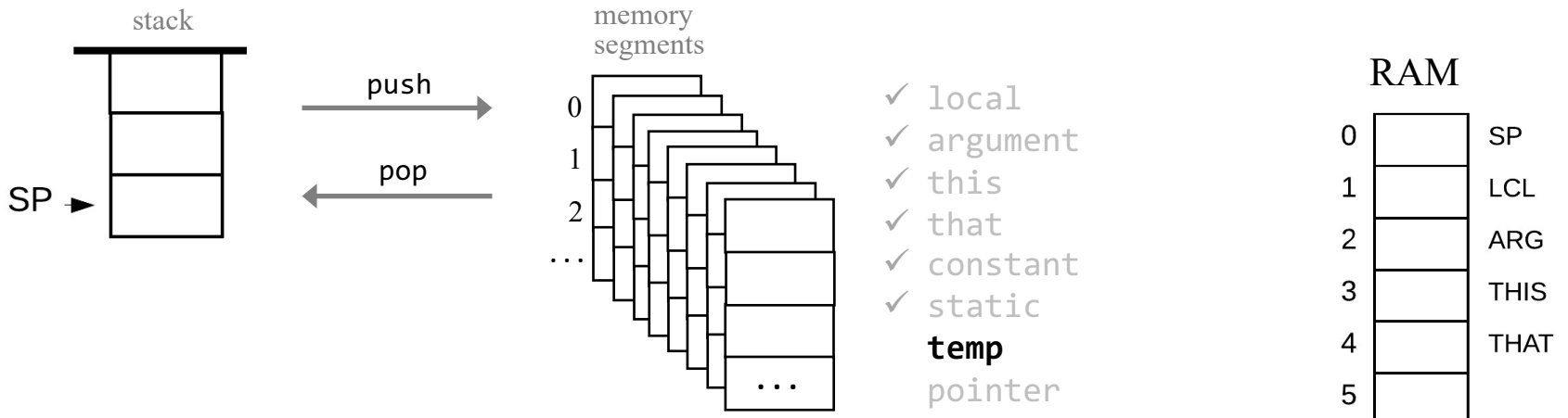




- ✓ `local`
- ✓ `argument`
- ✓ `this`
- ✓ `that`
- ✓ `constant`
- ✓ `static`
- `temp`
- `pointer`



Implementing `push/pop temp i`

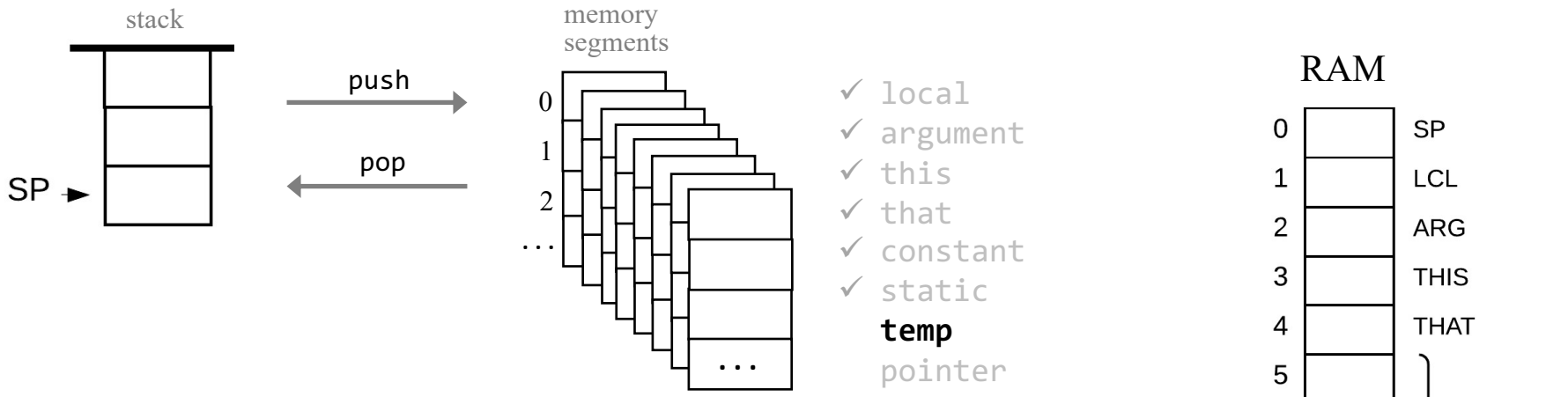


The Big Picture: Working variables

When translating high-level programs, compilers sometimes have to to add to the generated VM code some working variables of their own

Abstraction: A fixed, 8-entry segment: `temp 0`, `temp 1`, ..., `temp 7`

Implementing `push/pop temp i`



The Big Picture: Working variables

When translating high-level programs, compilers sometimes have to to add to the generated VM code some working variables of their own

Abstraction: A fixed, 8-entry segment: `temp 0`, `temp 1`, ..., `temp 7`

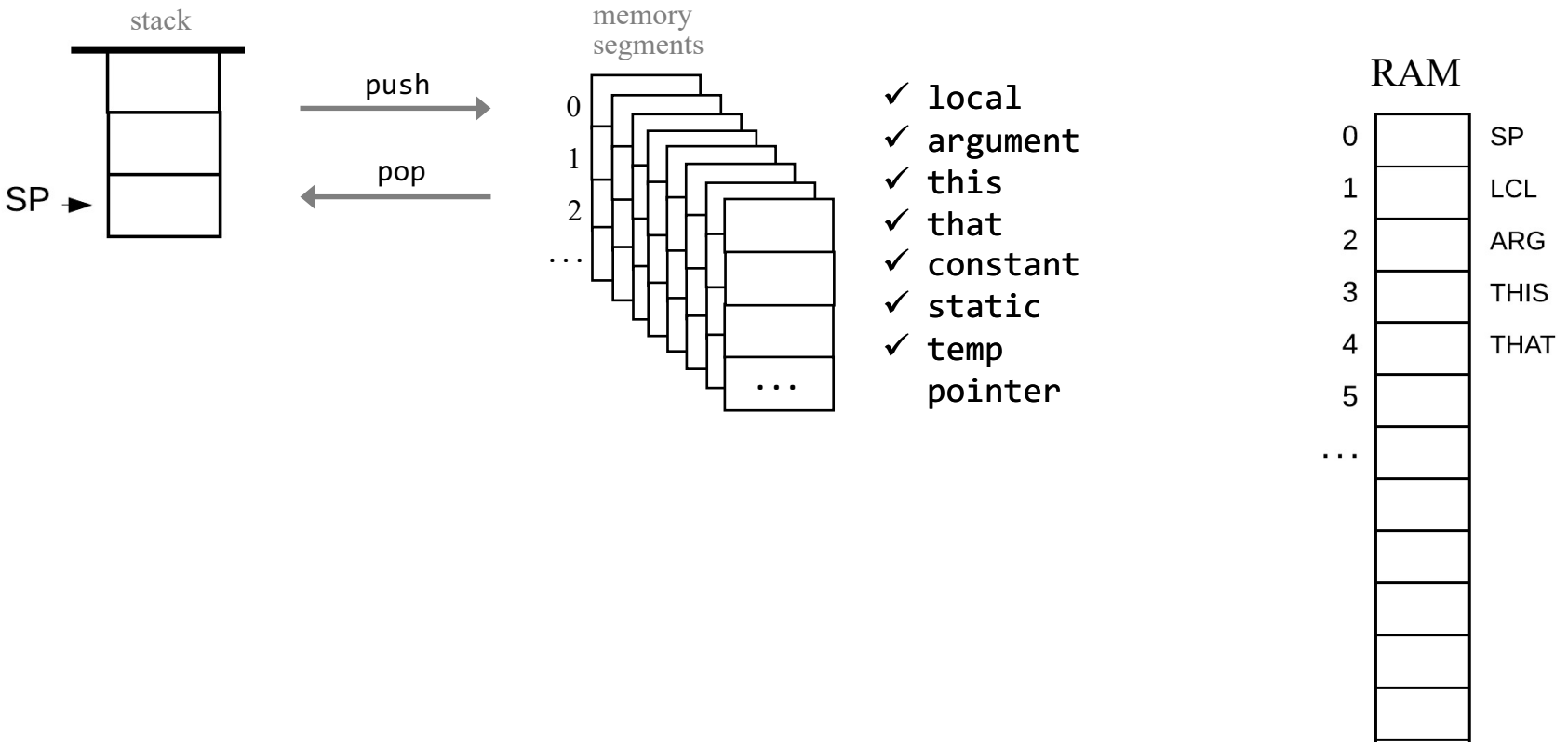
Implementation

The `temp` segment is stored in a fixed 8-word RAM block, starting at address 5;

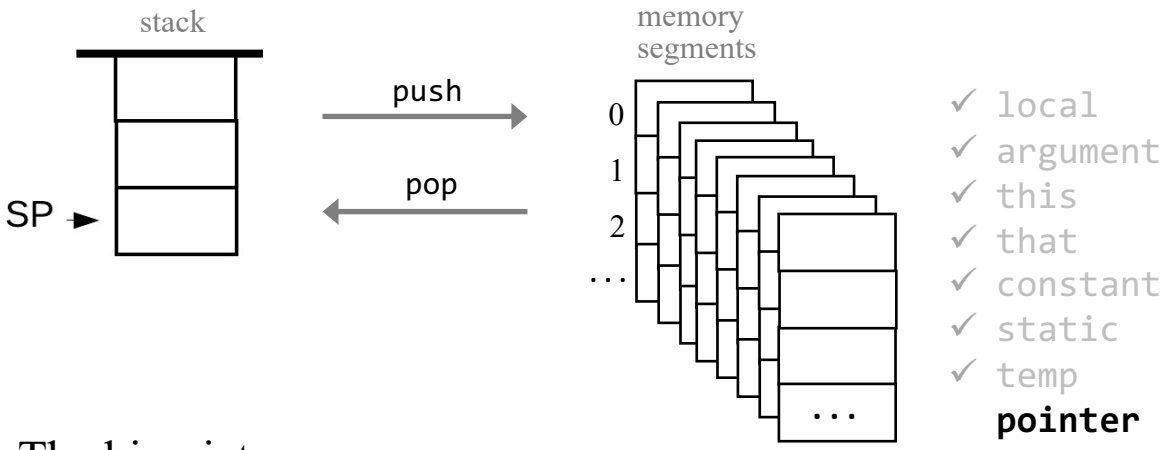
We translate each VM command: `push` / `pop temp` *i*

into assembly code that realizes: `push` / `pop RAM[5 + i]`

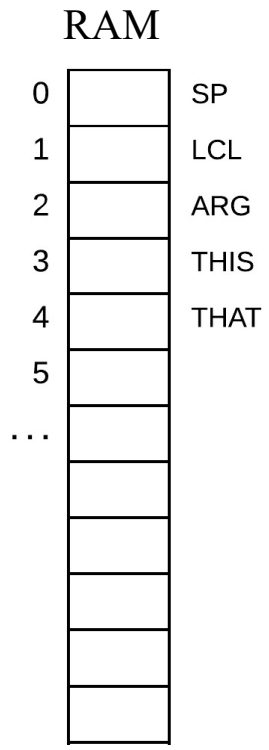
Implementing `push/pop`



Implementing `push/pop pointer i`



- ✓ `local`
- ✓ `argument`
- ✓ `this`
- ✓ `that`
- ✓ `constant`
- ✓ `static`
- ✓ `temp`
- **`pointer`**



The big picture

The fields of the *current object* are represented by the segment `this`, and the elements of the *current array* are represented by the segment `that`;

When executing a program that uses arrays and objects, we have to dynamically maintain the base addresses of these segments.
We store then in `pointer 0` and `pointer 1` (a 2-place segment)

(More about the `pointer` segment when we'll discuss the compiler in chapter 11)

Implementation

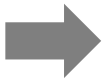
We translate each VM command: `push` / `pop pointer 0`
into assembly code that realizes: `push` / `pop THIS`

We translate each VM command: `push` / `pop pointer 1`
into assembly code that realizes: `push` / `pop THAT`

The VM language: Implementation

Push / pop commands

- `push` *segment i*
- `pop` *segment i*



Arithmetic / Logical commands

- `add`, `sub` , `neg`
- `eq` , `gt` , `lt`
- `and`, `or` , `not`

Implementing the VM arithmetic-logical commands

Command	Return value	Return value
`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

Abstraction

Each arithmetic/logical command `pops` one or two values from the stack, computes one of these functions on these values, and `pushes` the computed value onto the stack

Implementation

We've already discussed how to implement push and pop operations in assembly;

The implementation of `+`, `-`, `==`, `>`, `<`, `And`, `Or`, `Not` in assembly is simple

It follows that the implementation of the arithmetic / logical commands is straightforward.

The VM language

✓ Push / pop commands

- `push` *segment i*
- `pop` *segment i*

✓ Arithmetic / Logical commands

- `add`, `sub` , `neg`
- `eq` , `gt` , `lt`
- `and`, `or` , `not`

This lecture

Branching commands

- `label` *label*
- `goto` *label*
- `if-goto` *label*

Function commands

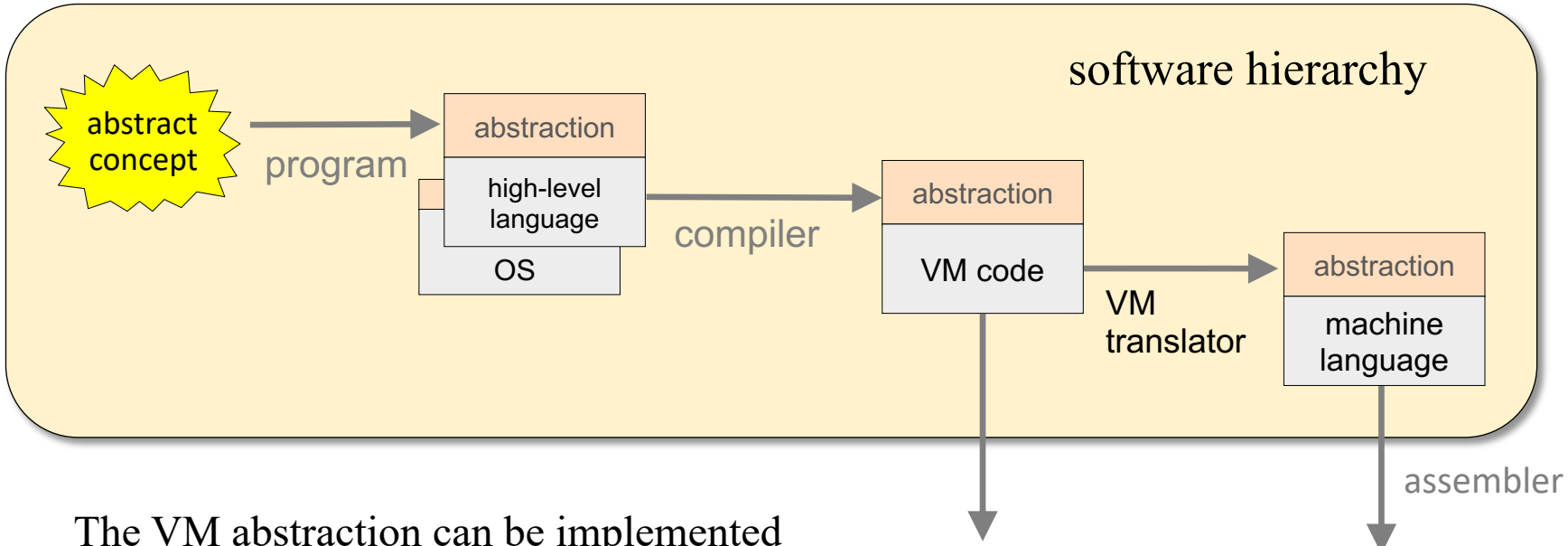
- `Function` *functionName nVars*
- `Call` *functionName nArgs*
- `return`

Next lecture

Lecture plan

- Overview
- VM Language (part 1): Abstraction
- VM Language (part 1): Implementation
- VM Emulator
- Standard Mapping
- VM Translator
- Project 7

VM implementations

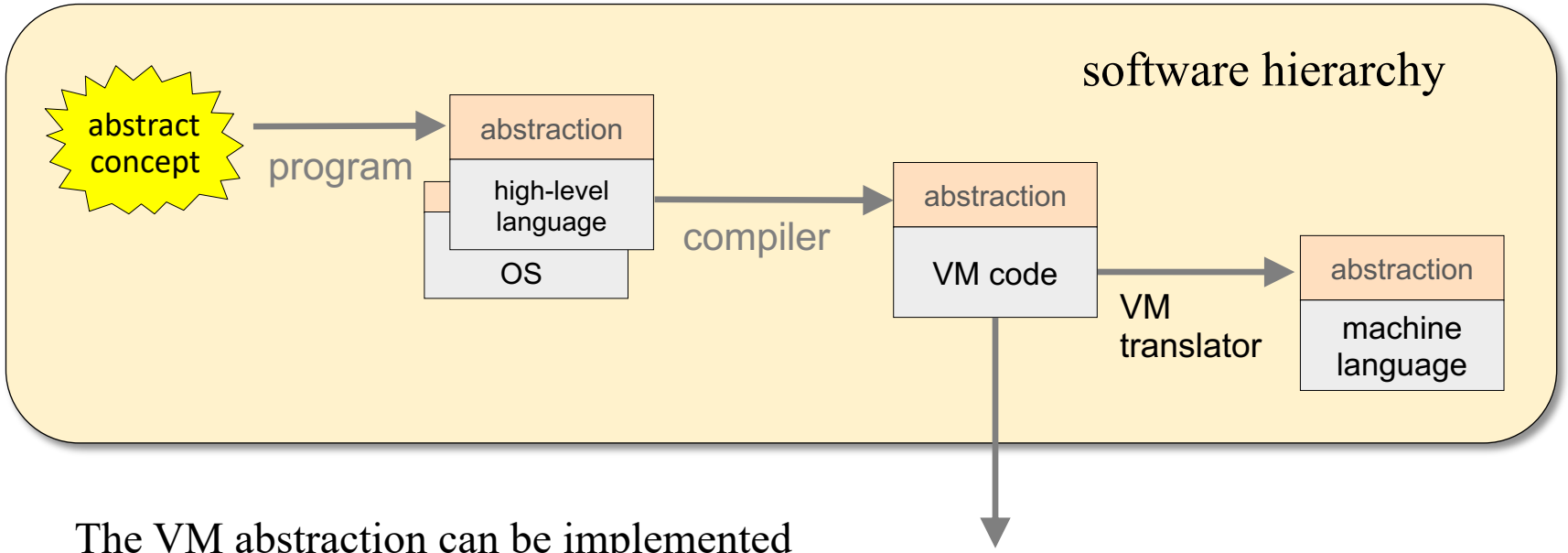


The VM abstraction can be implemented in several ways, including:

VM translator: Translates VM commands into the machine language of a target platform (project 7)

VM emulator: Uses a high-level language to execute VM commands.

VM implementations



The VM abstraction can be implemented in several ways, including:

VM emulator: Uses a high-level language to execute VM commands.

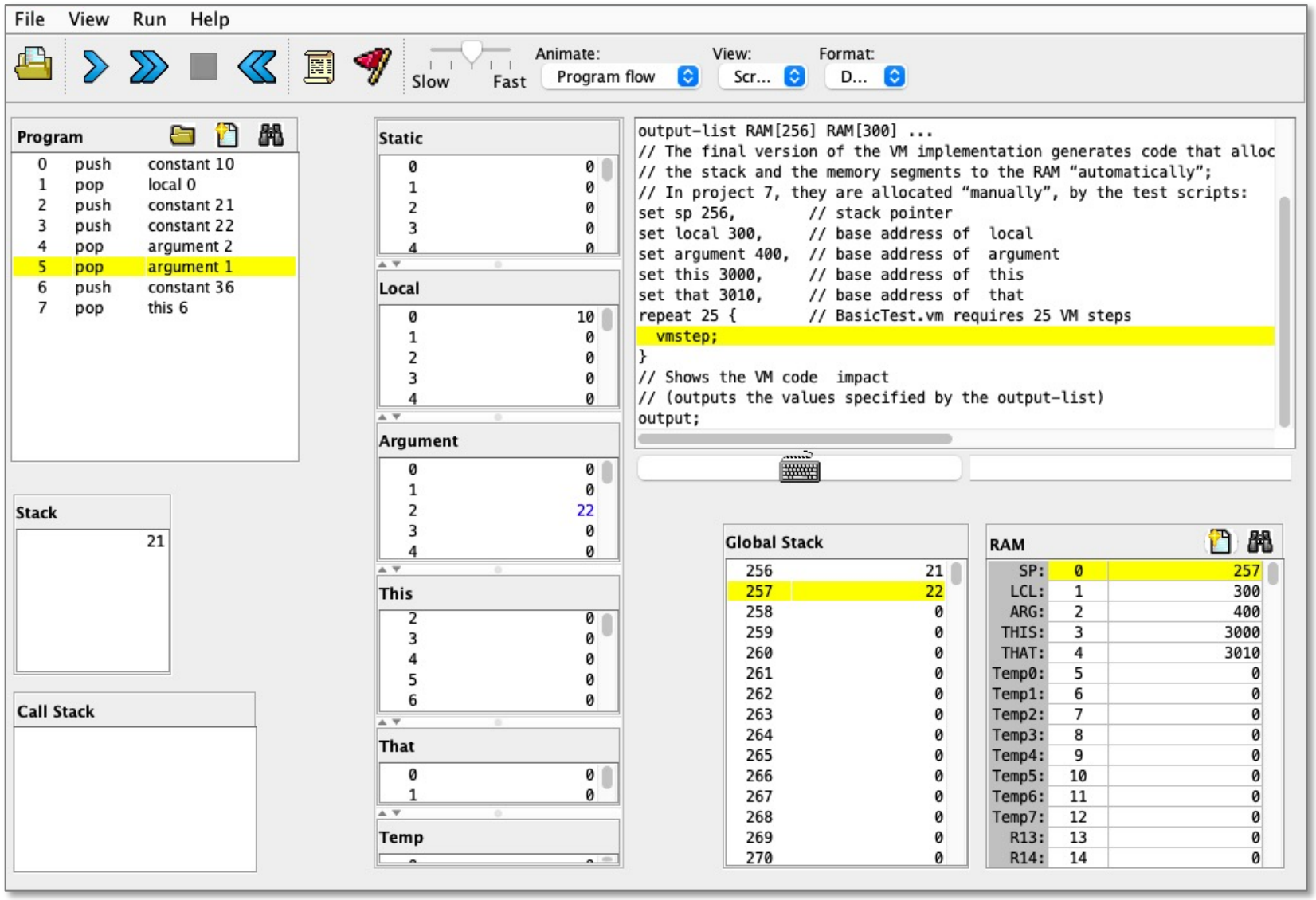
VM emulator

Purpose

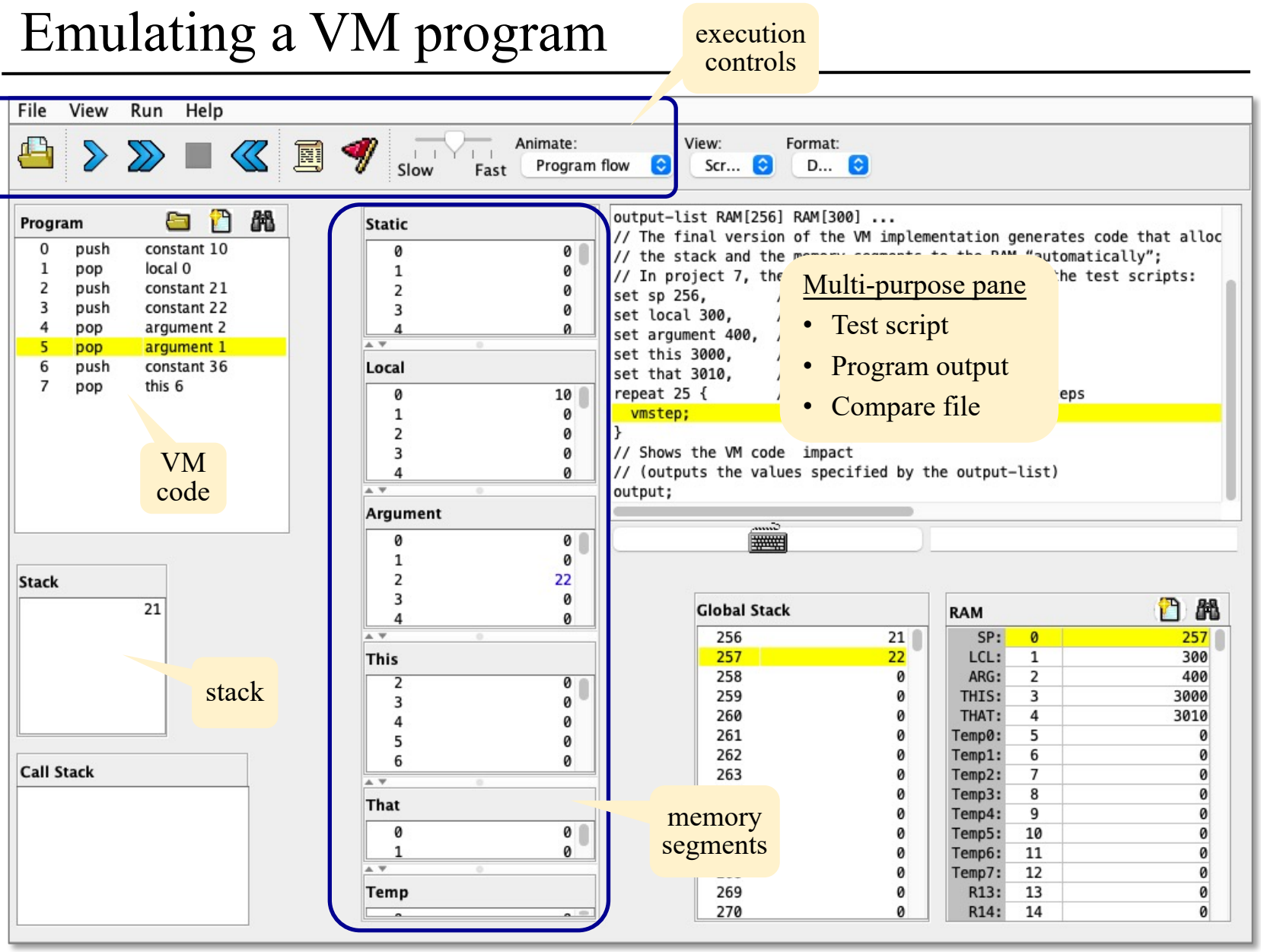
Visualizes...

- How VM commands work
- How the *stack* and the *virtual segments* are mapped on the host RAM.

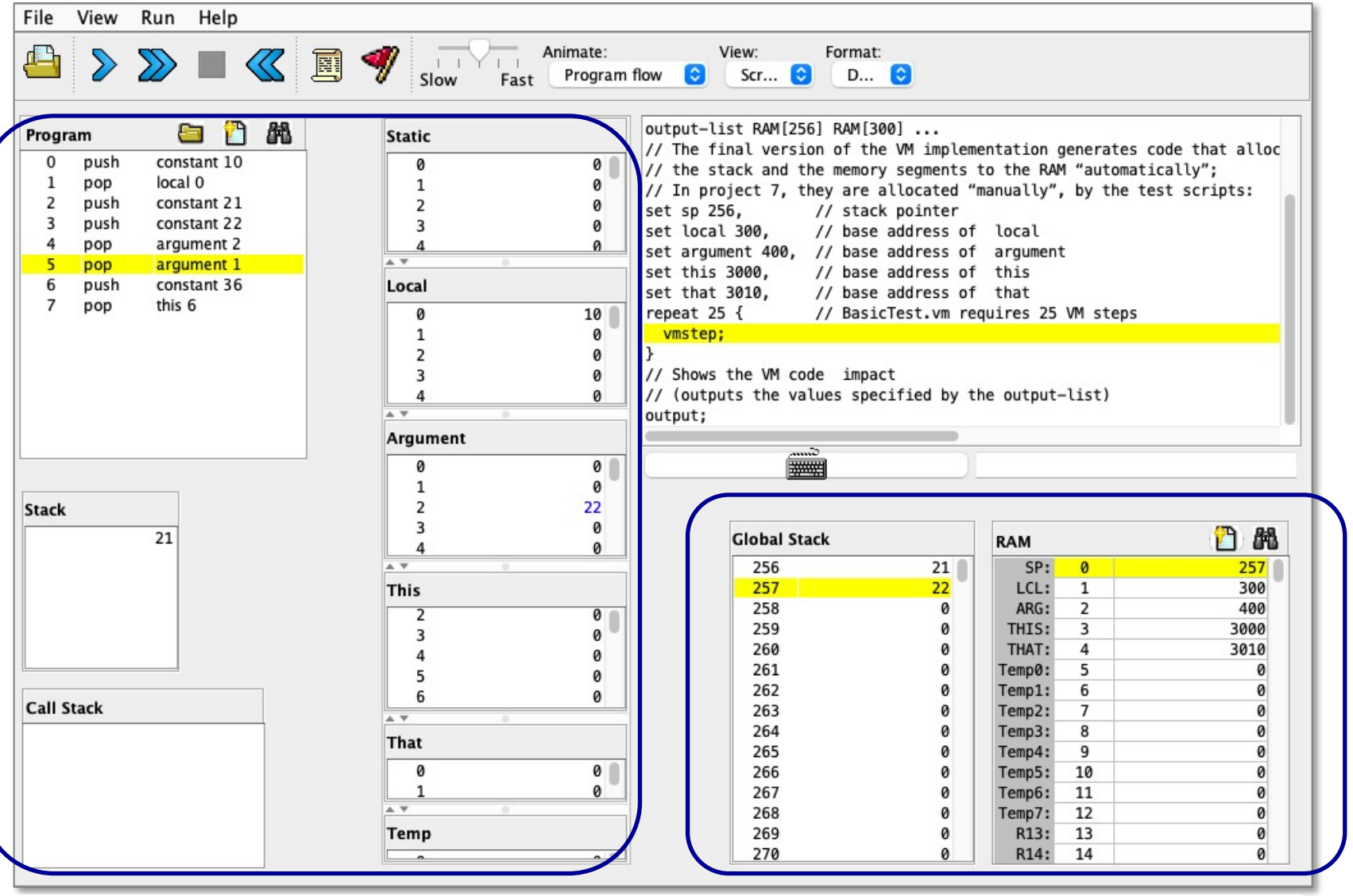
Emulating a VM program



Emulating a VM program



Emulating a VM program



Abstraction

How the abstraction is realized

Emulating a VM program: Testing

`BasicTest.vm` (example)

```
push constant 10
pop local 0
push constant 21
push constant 22
pop argument 2
pop argument 1
push constant 36
pop this 6
...
```

```
load BasicTest.vm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
// Specifies selected RAM values to output
// (contents of selected pointers, segment base addresses, etc.):
output-list RAM[256] RAM[300] ...

// The final version of the VM implementation generates code that allocates
// the stack and the memory segments to the RAM "automatically";
// In project 7, they are allocated "manually", by the test scripts:

set sp 256,          // stack pointer
set local 300,       // base address of local
set argument 400,    // base address of argument
set this 3000,       // base address of this
set that 3010,       // base address of that

repeat 25 {          // BasicTest.vm requires 25 VM steps
  vmstep;
}

// Shows the VM code impact
// (outputs the values specified by the output-list)
output;
```

The *progName*`VME.tst` scripts enable experimenting with the VM test programs in the VM emulator

Lecture plan

- Overview
- VM Language: Abstraction
- VM Language: Implementation
- VM Emulator
- Standard Mapping
- VM Translator
- Project 7

Standard VM mapping

Background

We've introduced a virtual machine (VM)

The VM can be implemented on various target platforms.

Standard VM mapping

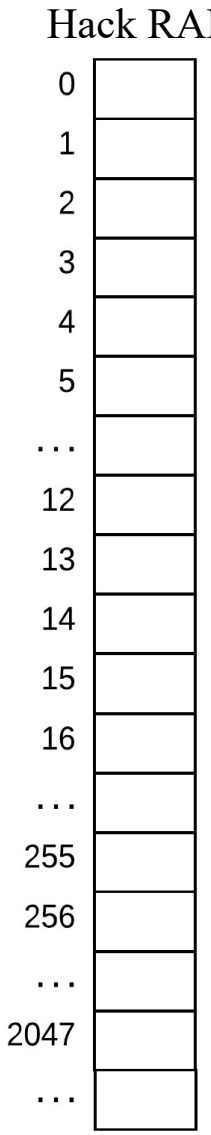
Recommends a *specific way* for realizing the VM on a specific target platform

Benefits

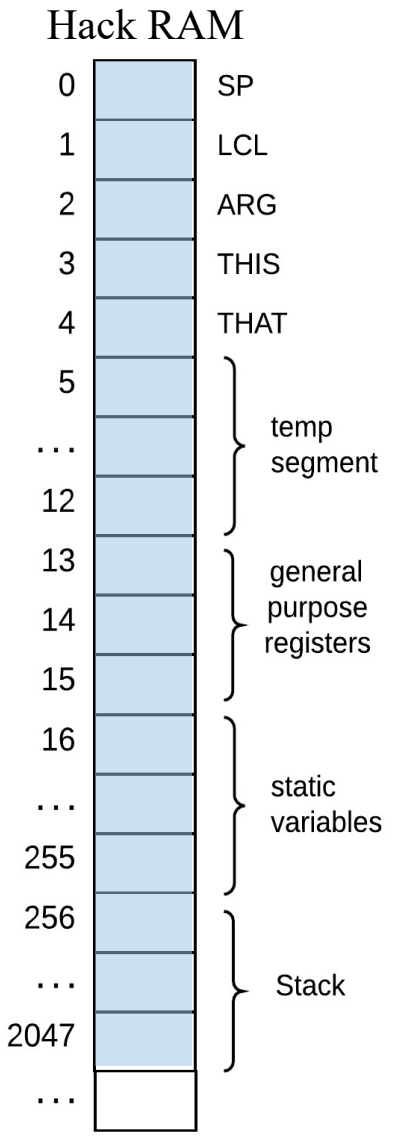
Compatibility with other tools / libraries that conform to this standard:

- VM emulators, OS
- Testing systems / test scripts
- Etc.

Standard VM mapping on the Hack platform



Standard VM mapping on the Hack platform



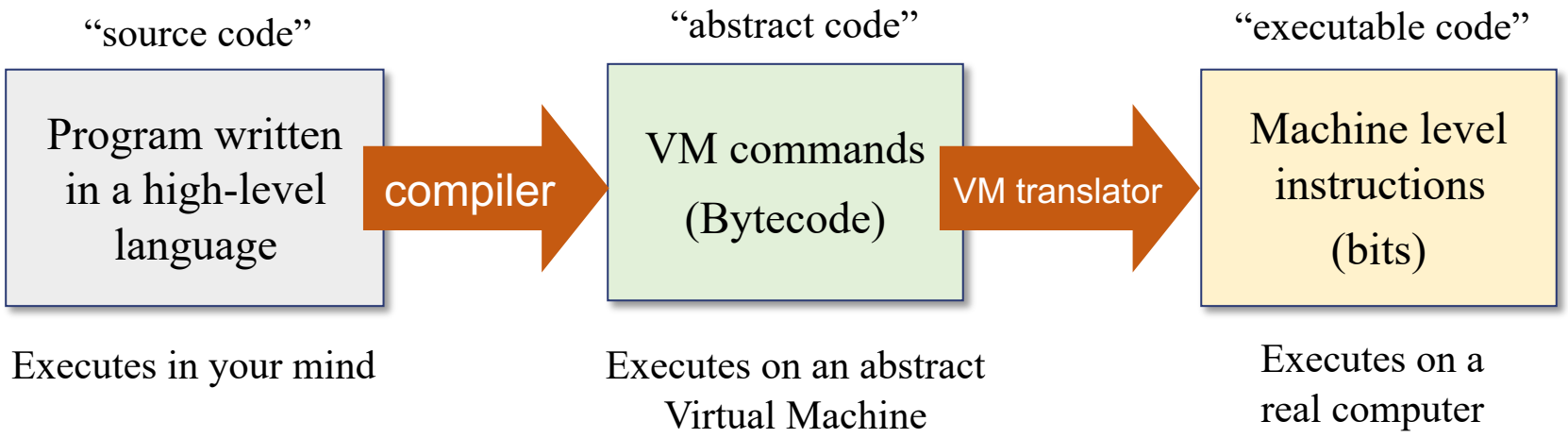
To realize this standard mapping, the assembly code generated by the VM translator must conform to the mapping shown on the left, and use the following symbols:

Symbol	***Usage***
`SP`	This predefined symbol points to the memory address within the host RAM just following the address containing the topmost stack value.
`LCL`, `ARG`, `THIS`, `THAT`	These predefined symbols point, respectively, to the base addresses within the host RAM of the virtual segments `local`, `argument`, `this`, and `that` of the currently running VM function.
`R13`–`R15`	These predefined symbols can be used for any purpose.
`Xxx.`*i* symbols	The `static` segment is implemented as follows: each static variable *i* in file `Xxx.vm` is translated into the assembly symbol `Xxx.`*i*. In the subsequent assembly process, these symbolic variables will be allocated to the RAM by the Hack assembler.

Lecture plan

- Overview
- VM Language: Abstraction
- VM Language: Implementation
- VM Emulator
- Standard Mapping
- VM Translator
- Project 7

The Big Picture: Program compilation



The VM translator

VM code (*fileName*.vm)

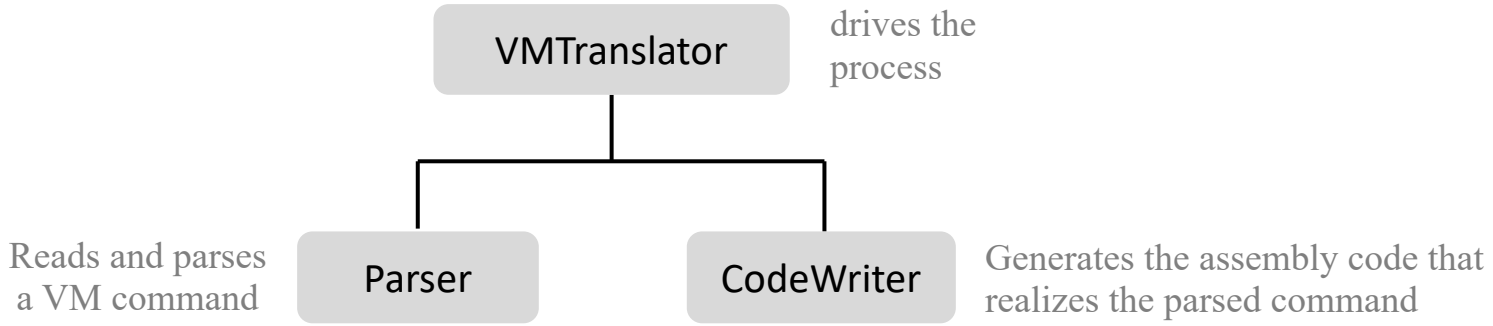
```
...
push constant 17
push local 2
add
pop argument 1
...
```

Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
... additional assembly code that completes the
    implementation of push constant 17
// push local 2
... assembly code that implements push local 2
// add
... assembly code that implements add
// pop argument 1
... assembly code that implements push argument 1
...
```

The VM translator creates an output `.asm` file, parses the source VM commands line by line, and emits their translation into the output file.

The VM translator: Design and Usage



Usage: (if the translator is implemented in Java)

```
$ java VMTranslator fileName.vm
```

(the *fileName* may contain a file path; the first character of *fileName* must be an uppercase letter).

Output: An assembly file named *fileName*`.asm`

Action

- Constructs a `Parser` to handle the input file
- Constructs a `CodeWriter` to handle the output file
- Iterates through the input file, parsing each line and generating code from it, using the services of the `Parser` and a `CodeWriter`.

Parser

Routines

- Constructor / initializer: Creates a `Parser` and opens the input (source VM code) file
- Getting the current instruction:
  - **`hasMoreLines()`:** Checks if there is more work to do (boolean)
  - **`advance()`:** Gets the next instruction and makes it the *current instruction* (string)
- Parsing the *current instruction*:
  - **`commandType()`:** Returns the type of the current command (string constant):
    `C_ARITHMETIC` if the current command is an arithmetic-logical command;
    `C_PUSH`, `C-POP`, `C_LABEL`, `C_GOTO`, `C_IF`, `C_FUNCTION`, `C_RETURN`, `C_CALL`
    if the current command is any of these command types
  - **`arg1()`:** Returns the first argument of the current command;
    In the case of `C_ARITHMETIC`, the command itself is returned (string)
  - **`arg2()`:** Returns the second argument of the current command (int);
    Called only if the current command is `C_PUSH`, `C-POP`, `C_FUNCTION`, or `C_CALL`

Examples:

current command	
`add, neg, eq, ...`	`commandType()` returns `C_ARITHMETIC`; `arg1()` returns `"add"`, `"neg"`, `"eq"`,...
`push local 3`	`commandType()` returns `C_PUSH`; `arg1()` returns `"local"`; `arg2()` returns `3`
`call foo 17`	`commandType()` returns `C_CALL`; `arg1()` returns `"foo"`; `arg2()` returns `17`

Parser API (detailed)

- Handles the parsing of a single `.vm` file
- Reads a VM command, parses the command into its lexical components, and provides convenient access to these components
- Ignores white space and comments

Routine	***Arguments***	***Returns***	***Function***
constructor	input file / stream	—	Opens the input file/stream, and gets ready to parse it.
`hasMoreLines`	—	boolean	Are there more lines in the input?
`advance`	—	—	Reads the next command from the input and makes it the *current command*. This method should be called only if `hasMoreLines` is true. Initially there is no current command.

(continues in the next slide)

Parser API (detailed)

- Handles the parsing of a single `.vm` file
- Reads a VM command, parses the command into its lexical components, and provides convenient access to these components
- Ignores white space and comments

Routine	*Arguments*	*Returns*	*Function*
`commandType`	—	`C_ARITHMETIC`, `C_PUSH`, `C_POP`, `C_LABEL`, `C_GOTO`, `C_IF`, `C_FUNCTION`, `C_RETURN`, `C_CALL` (constant)	Returns a constant representing the type of the current command. If the current command is an arithmetic-logical command, returns `C_ARITHMETIC`.
`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 API

Generates assembly code from the parsed VM command

Routine	*Arguments*	*Returns*	*Function*
constructor	output file / stream	—	Opens an output file / stream and gets ready to write into it.
`writeArithmetic`	command (string)	—	Writes to the output file the assembly code that implements the given arithmetic-logical command.
`WritePushPop`	command (`C_PUSH` or `C_POP`), segment (string), `index` (int)	—	Writes to the output file the assembly code that implements the given push or pop command.
`close`	—	—	Closes the output file.

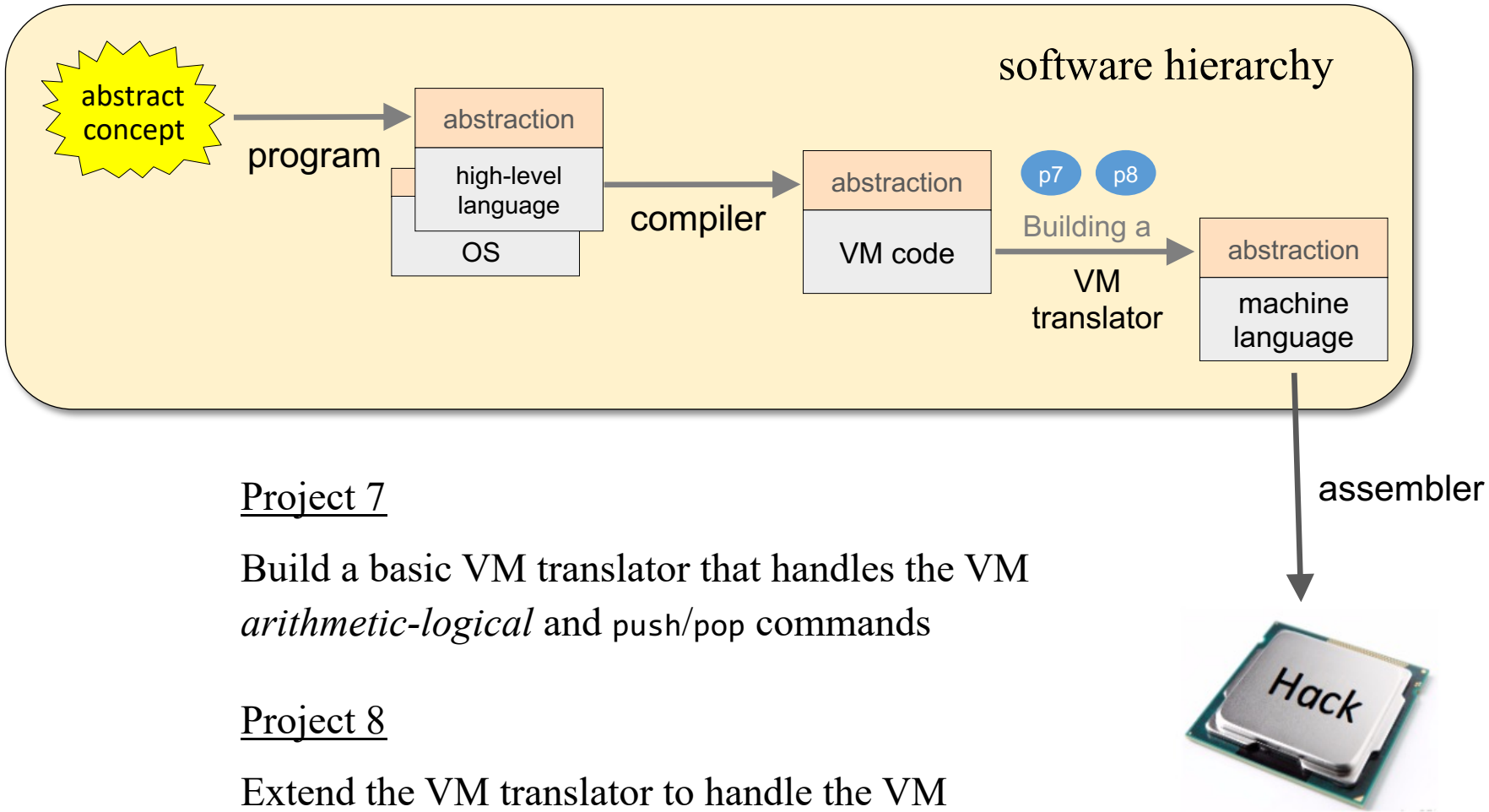
Notes

- The components/fields of each VM command are supplied by the Parser routines;
- Before committing it to code, write and debug *on paper* the assembly code that each VM command should generate;
- More routines will be added to this module in Project 8, for handling all the commands of the VM language.

Virtual Machine I

- Overview
- VM Language: Abstraction
- VM Language: Implementation
- VM Emulator
- Standard Mapping
- VM Translator
- Project 7

Project 7



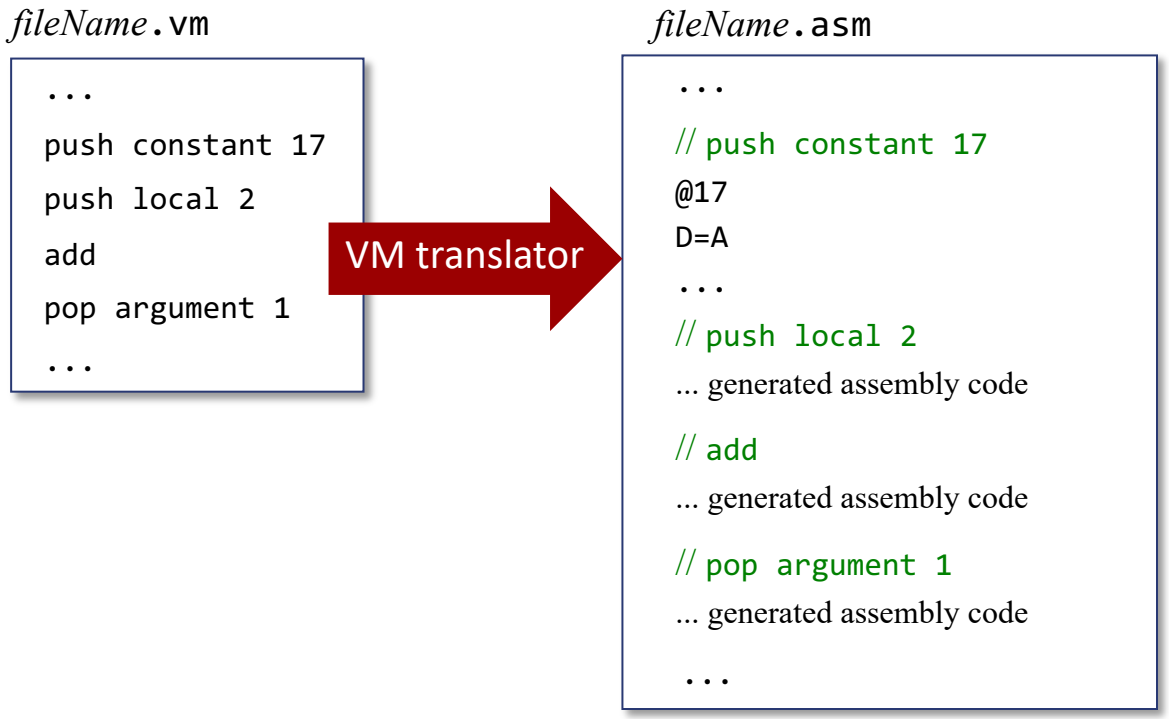
Project 7

Build a basic VM translator that handles the VM *arithmetic-logical* and `push`/`pop` commands

Project 8

Extend the VM translator to handle the VM *branching* and *function* commands.

Project 7



Testing option 1: Translate the generated assembly code into machine language: run the binary code on the Hack computer

Testing option 2 (simpler): Run the generated assembly code on the CPU emulator.

Project 7

Test programs

`SimpleAdd.vm`
`StackTest.vm`
`BasicTest.vm`
`PointerTest.vm`
`StaticTest.vm`

Example:

`BasicTest.vm`

```
...
push constant 510
pop temp 6
push local 0
push that 5
add
push argument 1
sub
...
```

Given

`BasicTest.asm`

```
...
// push constant 510
@510
D=A
...
```

Generated by your VM translator

For each test program *xxx*`.vm`

We supply three files:
xxx`VME.tst`, *xxx*`.tst` and *xxx*`.cmp`

0. (optional) Load and run the *xxx*`VME.tst` test script in the *VM emulator*; This will cause the emulator to load and execute *xxx*`.vm`; Observe the program's operation, and how its stack and segments are managed on the host RAM
1. Use your VM translator to translate *xxx*.vm; The result will be a file named *xxx*`.asm`
2. Inspect the generated code; If there's a problem, fix your translator and go to stage 1
3. Load and run the *xxx*`.tst` test script in the *CPUemulator,* This will cause the emulator to load and execute *xxx*`.asm`; Inspect the results
4. If there's a problem, fix your translator and go to stage 1.

Recap: The VM language

Push/pop commands

`push` *symbol n*

`pop` *symbol n*

Arith./logical commands

`add`, `sub`, `neg`, `eq`, `gt`, `lt`,
`and`, `or`, `not`

Branching commands

`label` *symbol*

`goto` *symbol*

`if-goto` *symbol*

Function commands

`function` *symbol n*

`call` *symbol n*

`return`

Where *symbol* is a string and *n* is a non-negative integer

`//` `comments`, indentation, and white space are allowed and ignored.

Notes

- This slide documents the syntax of all the VM commands, including the *branching* and *function* commands that will be presented in the next lecture
- Parsing-wise, we don't care what the commands do; We focus only on their syntax
- We suggest that the parser (but not the code generator) that you write in project 7 will handle *all* the VM commands presented here. You will have to do it anyway in project 8, and it can be done in project 7 with minimal extra effort.