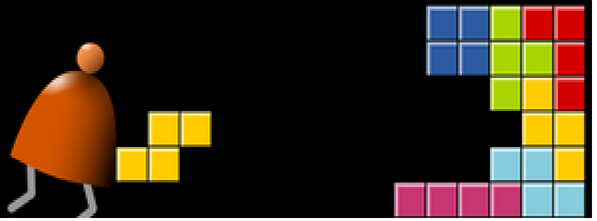
From Nand to Tetris

Building a Modern Computer from First Principles



Lecture 7

Virtual Machine I: Processing

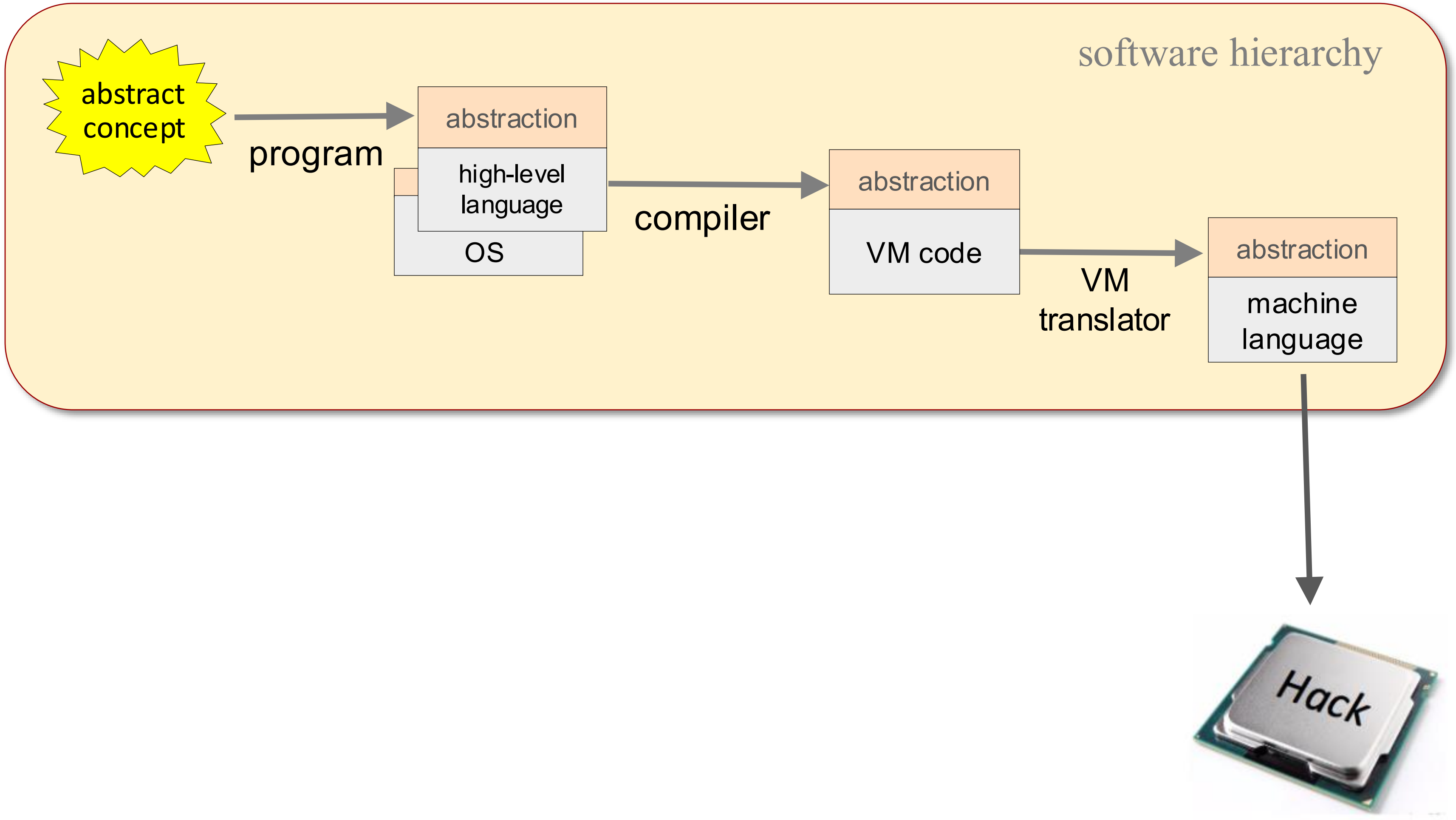
Slide deck for Chapter 7 of the book

The Elements of Computing Systems (2nd edition)

By Noam Nisan and Shimon Schocken

MIT Press

Nand to Tetris Roadmap: Part II



software hierarchy

abstract concept

program

abstraction

high-level language

OS

compiler

abstraction

VM code

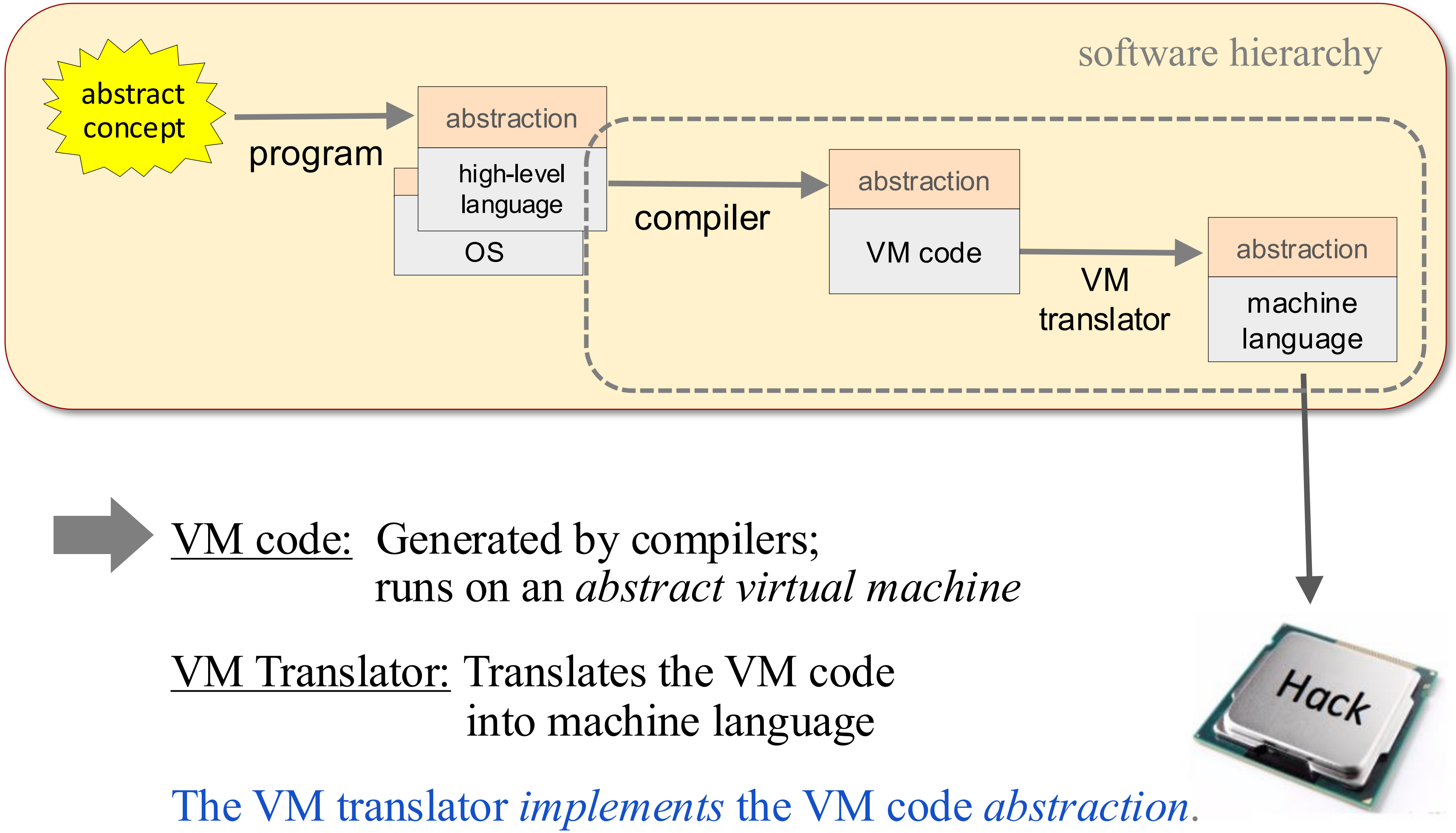
VM translator

abstraction

machine language

Hack

Nand to Tetris Roadmap: Part II



VM code: Generated by compilers;
runs on an *abstract virtual machine*

VM Translator: Translates the VM code
into machine language

The VM translator *implements* the VM code *abstraction*.

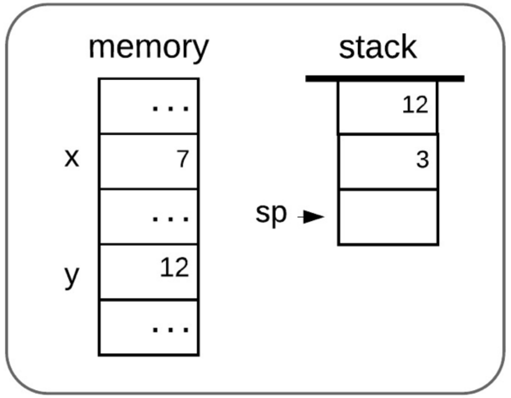
Our VM is *stack-based*

<u>Basic operations</u>

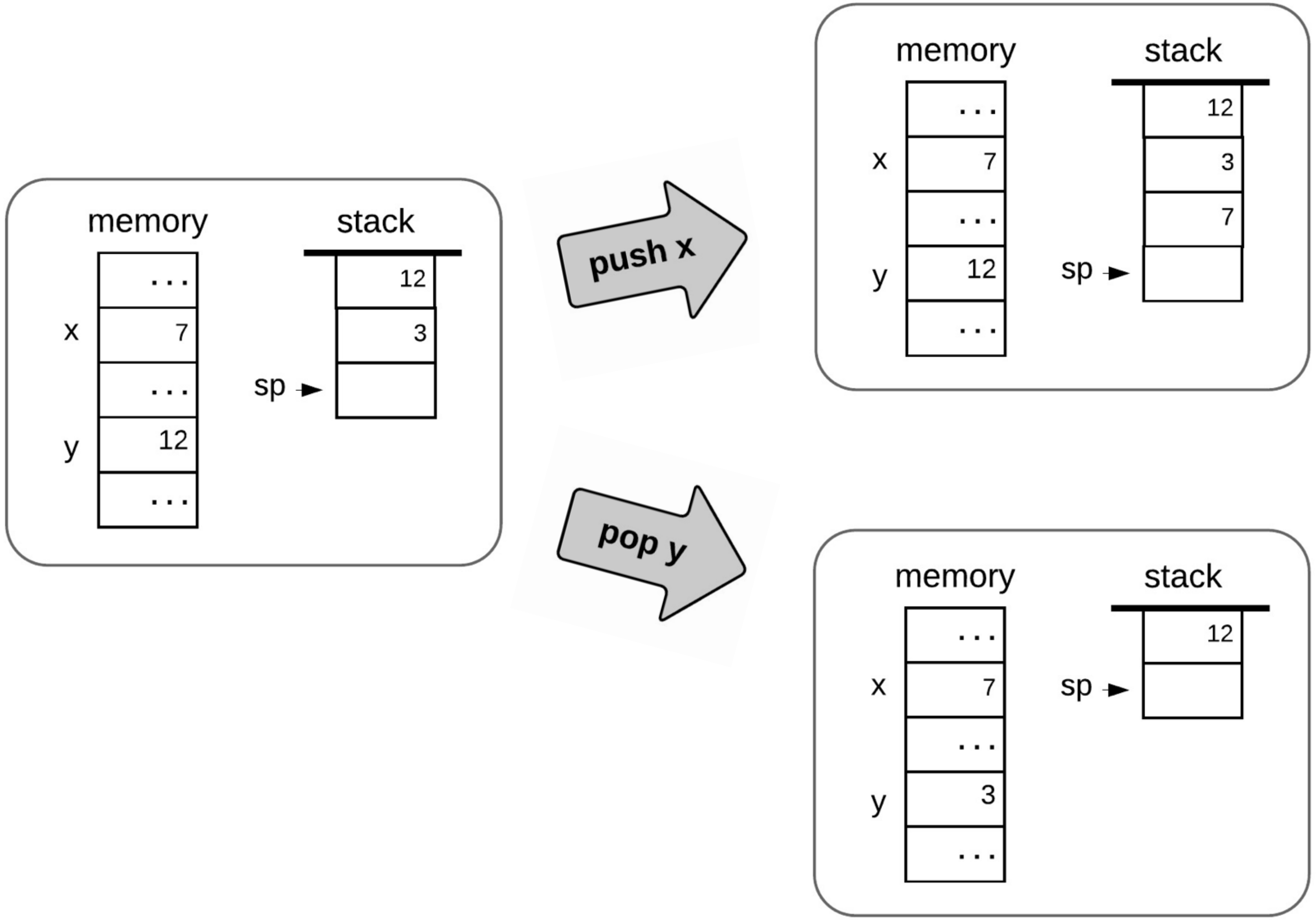
`push`: adds an element at the stack's top

`pop`: removes the top element

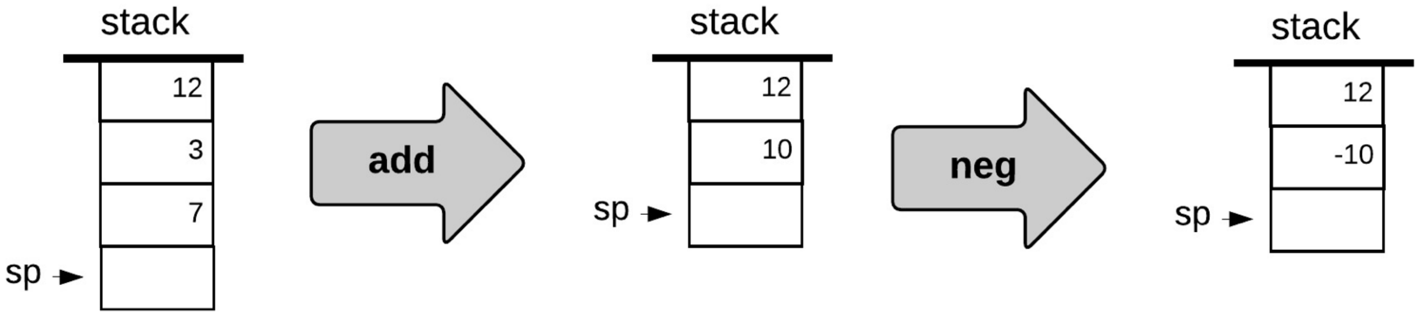
Stack



Stack


memory
stack
...
x
7
...
y
12
...
12
3
sp
push x
pop y
memory
stack
...
x
7
...
y
12
...
12
3
7
sp
memory
stack
...
x
7
...
y
3
...
12
sp

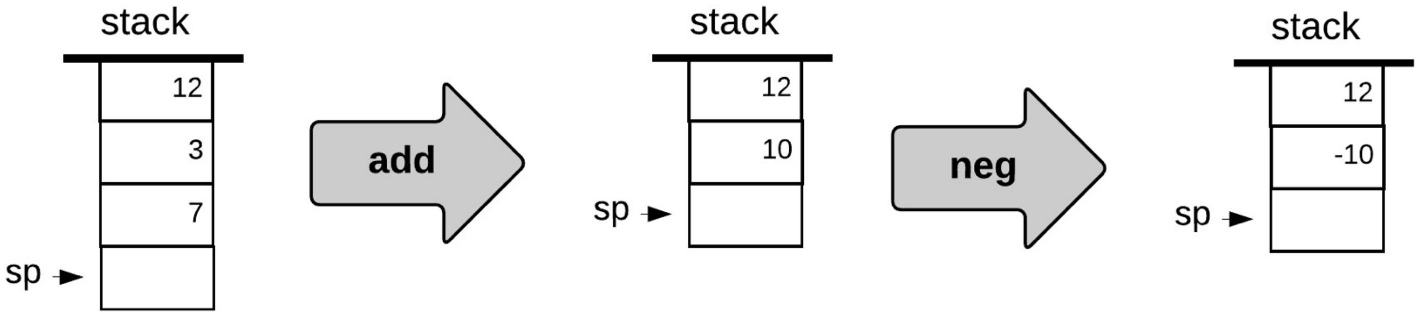
Stack arithmetic



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

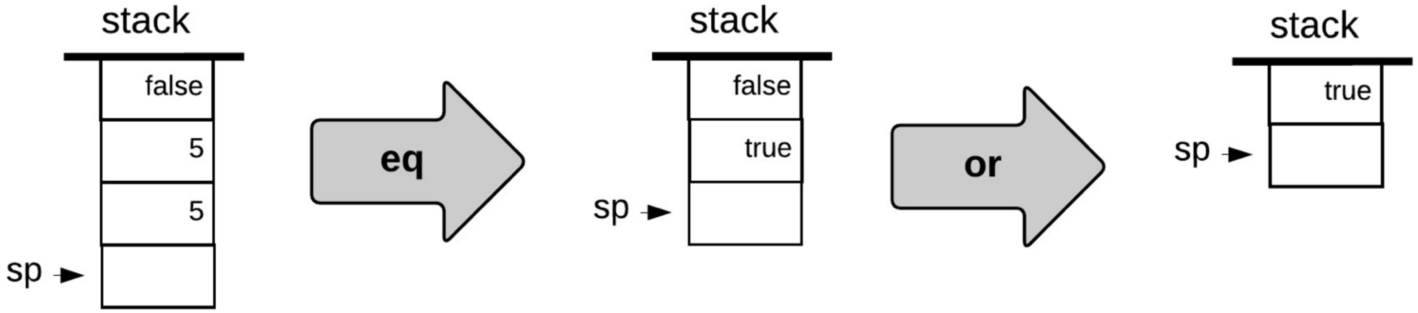
- pops *n* values (arguments) from the stack,
- Computes *f* on the values,
- Pushes the resulting value onto the stack.

Stack arithmetic



<u>Applying a function</u> *f* (that has *n* arguments)

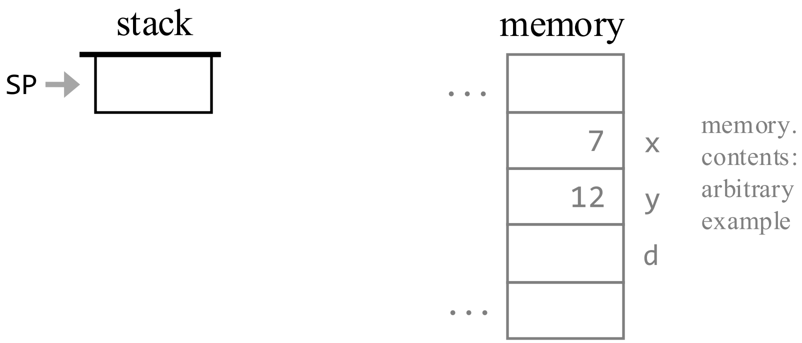
- pops *n* values (arguments) from the stack,
- Computes *f* on the values,
- Pushes the resulting value onto the stack.



Arithmetic operations

VM pseudocode (example)

```
// d = (2 – x) + (y + 9)
```



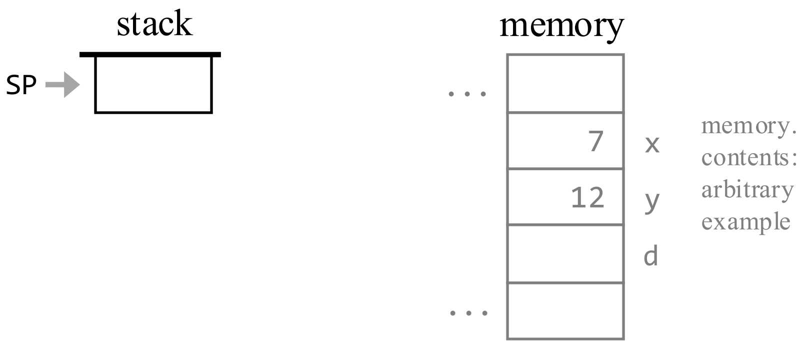
memory.
contents:
arbitrary
example

Arithmetic operations

VM pseudocode (example)

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

typically, generated by a compiler

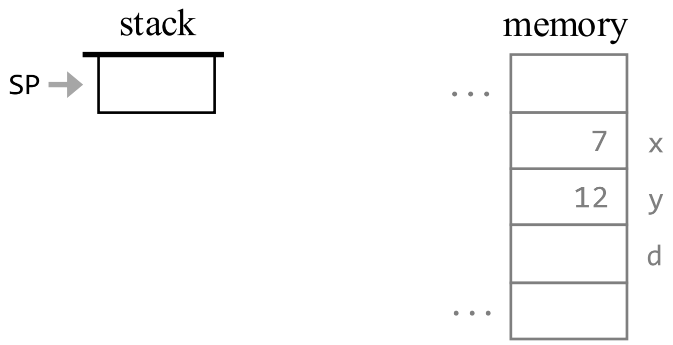


memory contents: arbitrary example

Arithmetic operations

VM pseudocode

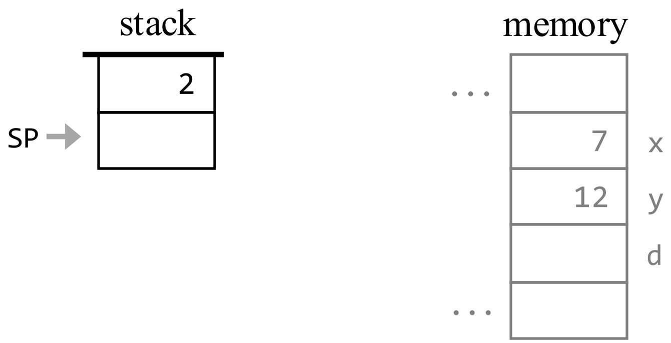
```
// d = (2 – x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic operations

VM pseudocode

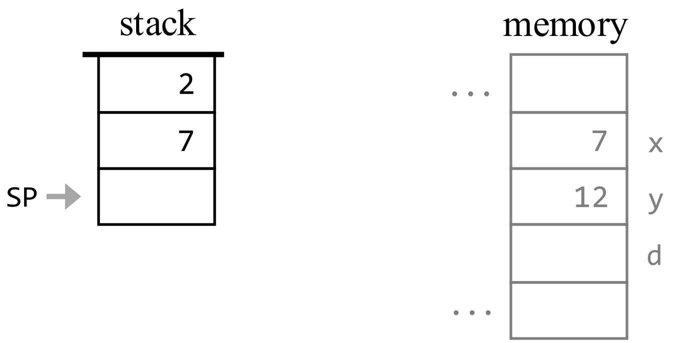
```
// d = (2 – x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic operations

VM pseudocode

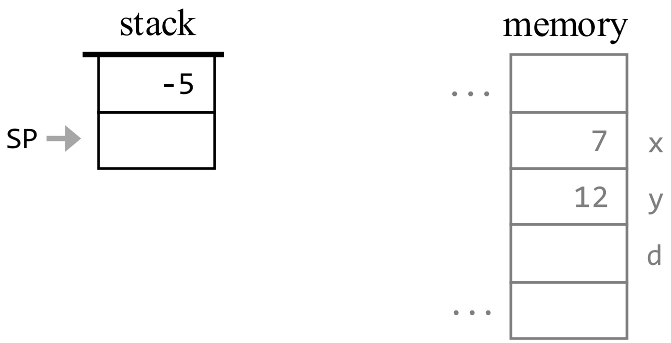
```
// d = (2 – x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic operations

VM pseudocode

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



stack

-5

SP

memory

...

7 x

12 y

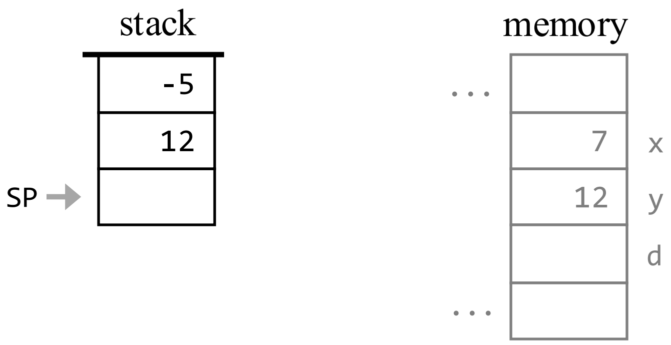
d

...

Arithmetic operations

VM pseudocode

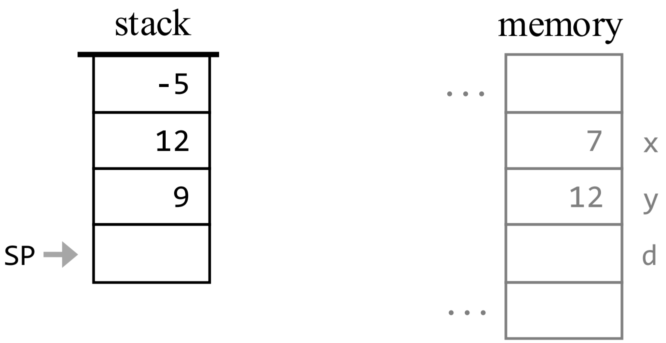
```
// d = (2 – x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic operations

VM pseudocode

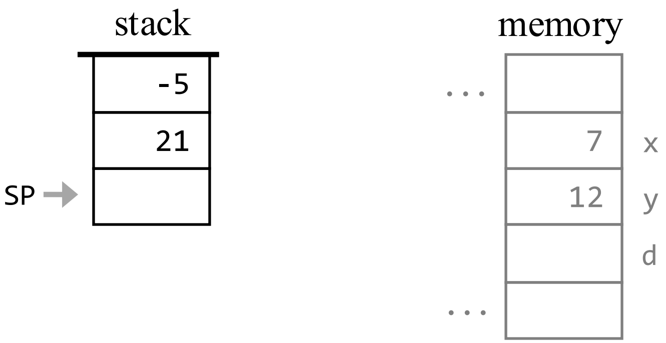
```
// d = (2 – x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic operations

VM pseudocode

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



stack

-5

21

SP

memory

...

7 x

12 y

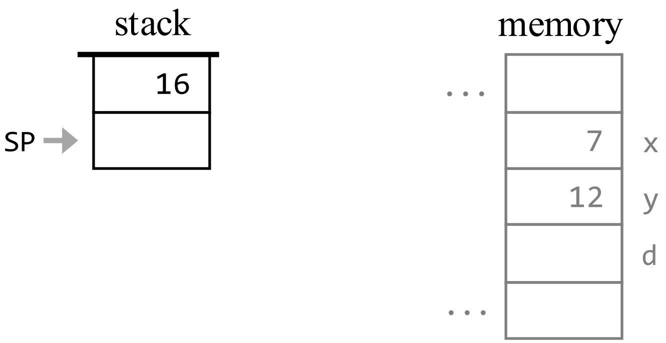
d

...

Arithmetic operations

VM pseudocode

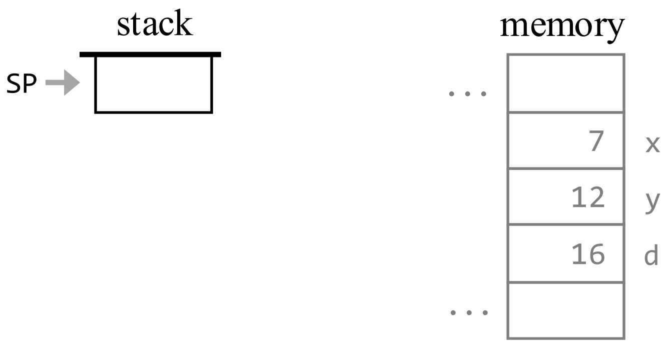
```
// d = (2 - x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic operations

VM pseudocode

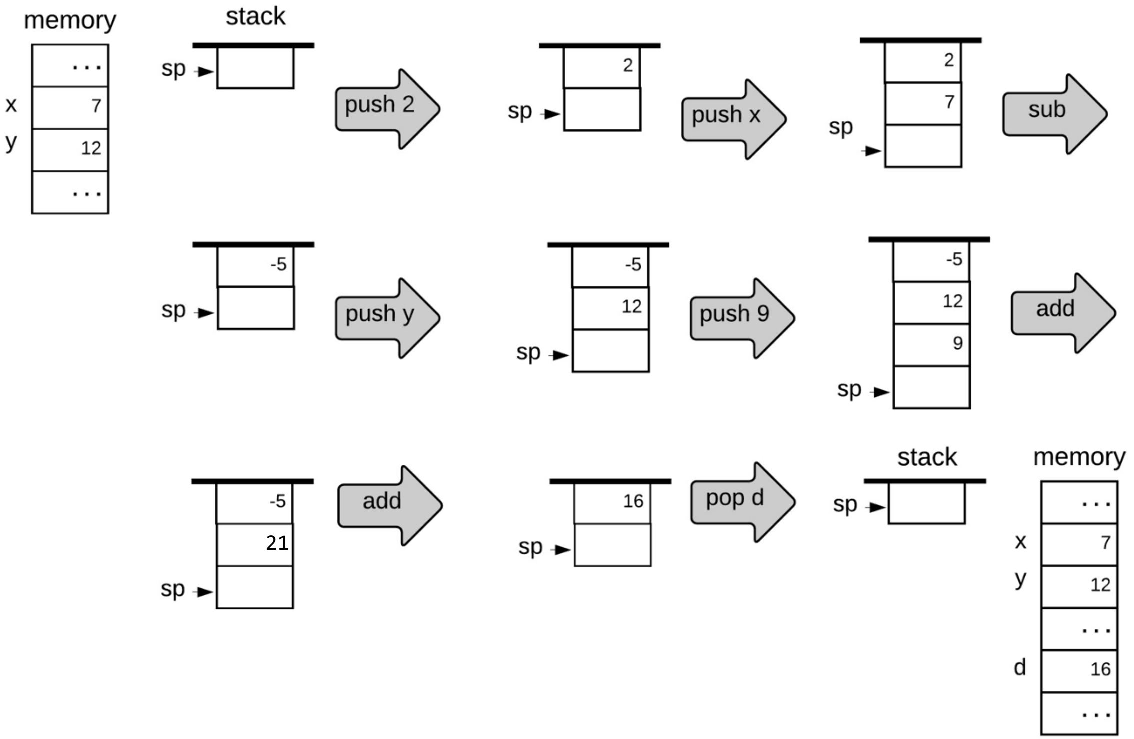
```
// d = (2 – x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic operations (example recap)

VM pseudocode

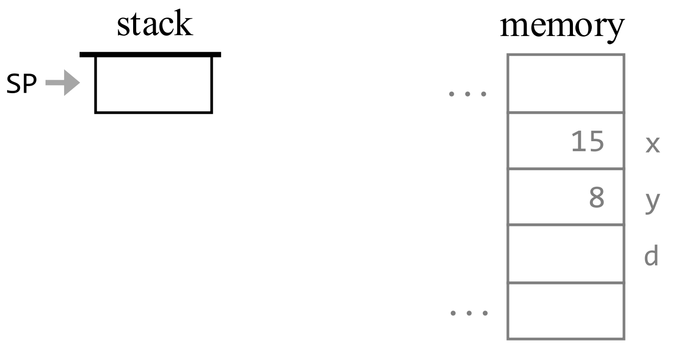
```
// d = (2 − x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Logical operations

VM pseudocode (another example)

```
// (x < 7) or (y == 8)
```



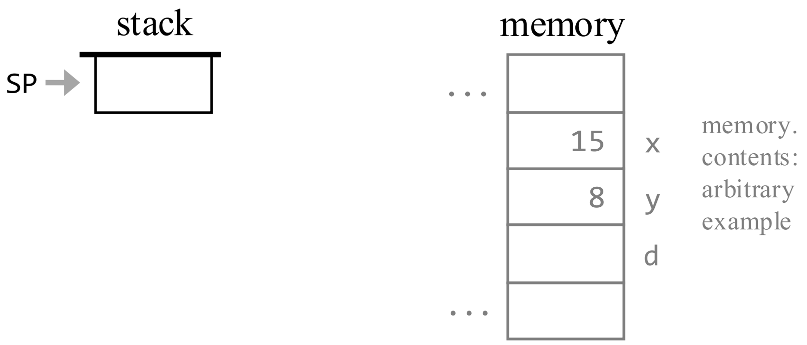
memory. contents: arbitrary example

Logical operations

VM pseudocode (another example)

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

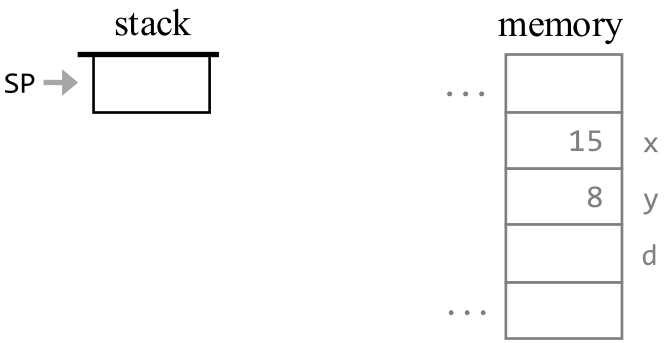
typically, generated by a compiler



Logical operations

VM pseudocode

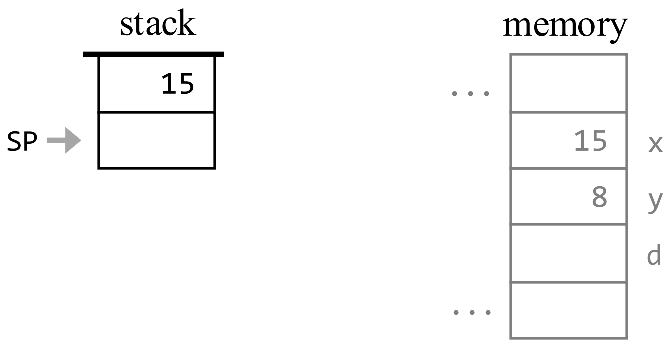
```
// (x < 7) or (y == 8)
push x
push 7
lt
push y
push 8
eq
or
```



Logical operations

VM pseudocode

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



stack

15

SP

memory

. . .

15 x

8 y

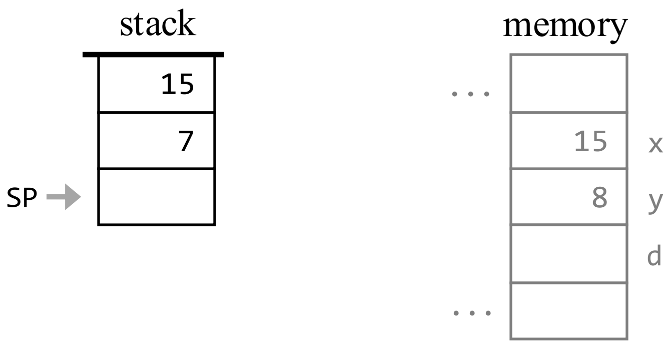
d

. . .

Logical operations

VM pseudocode

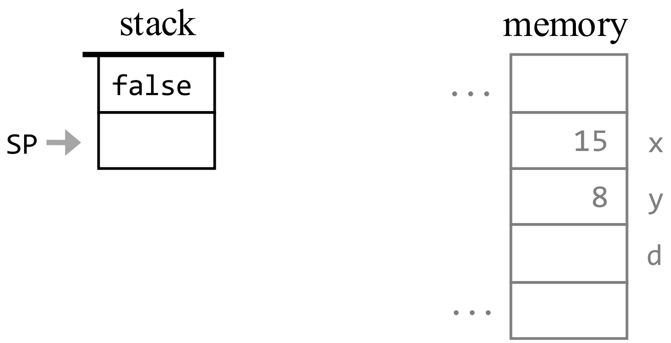
```
// (x < 7) or (y == 8)
push x
push 7
lt
push y
push 8
eq
or
```



Logical operations

VM pseudocode

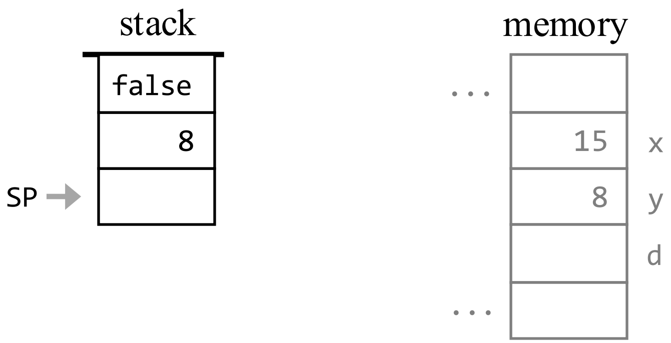
```
// (x < 7) or (y == 8)
push x
push 7
lt
push y
push 8
eq
or
```



Logical operations

VM pseudocode

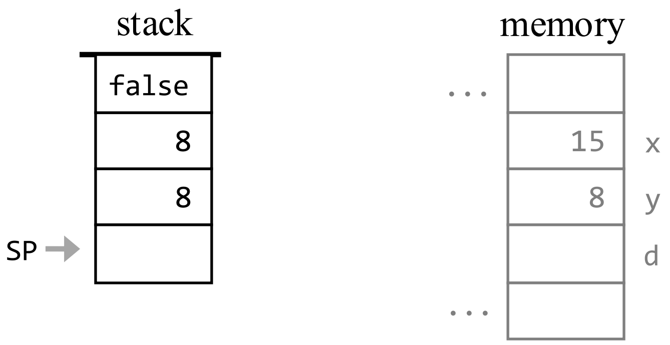
```
// (x < 7) or (y == 8)
push x
push 7
lt
push y
push 8
eq
or
```



Logical operations

VM pseudocode

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



stack

false
8
8
SP

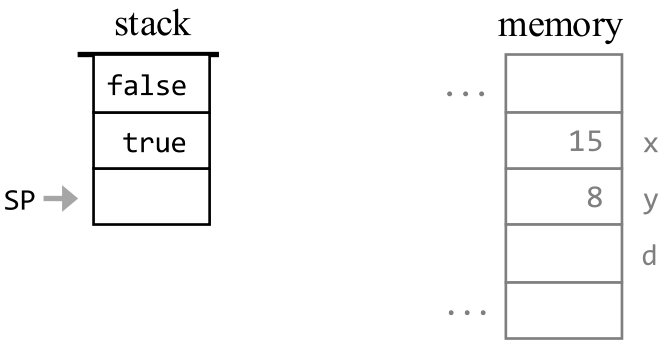
memory

...
15 x
8 y
d
...

Logical operations

VM pseudocode

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



stack

false

true

SP

memory

...

15 x

8 y

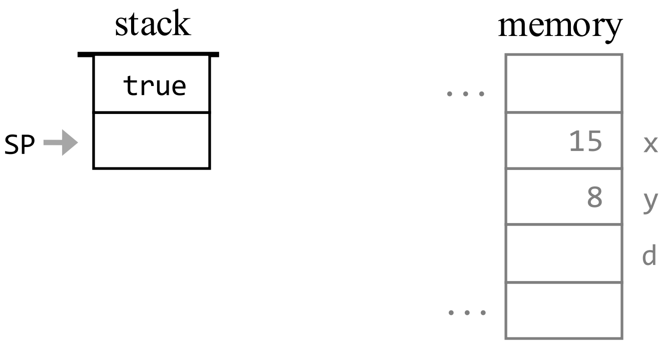
d

...

Logical operations

VM pseudocode

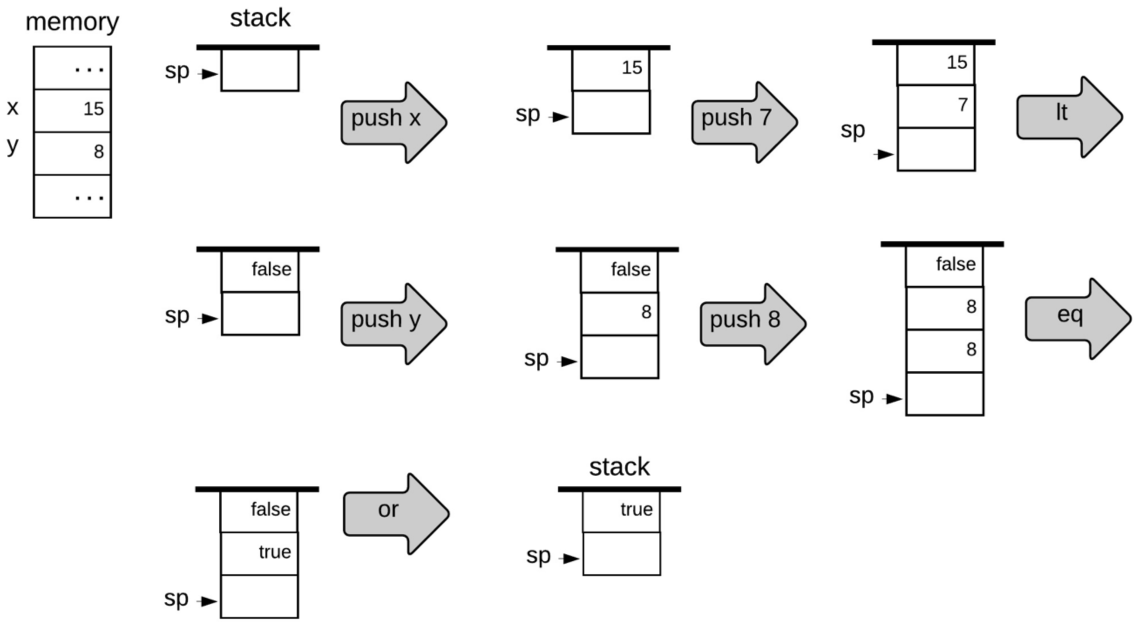
```
// (x < 7) or (y == 8)
push x
push 7
lt
push y
push 8
eq
or
```



Logical operations (example recap)

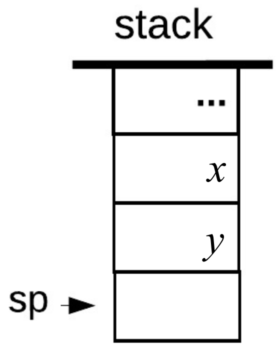
VM pseudocode (example 2)

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



Arithmetic / Logical commands: Recap

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.

The big picture: Compilation / expressiveness

Every high-level arithmetic ot logical expression can be translated into a sequence of VM commands, operating in a stack.

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*

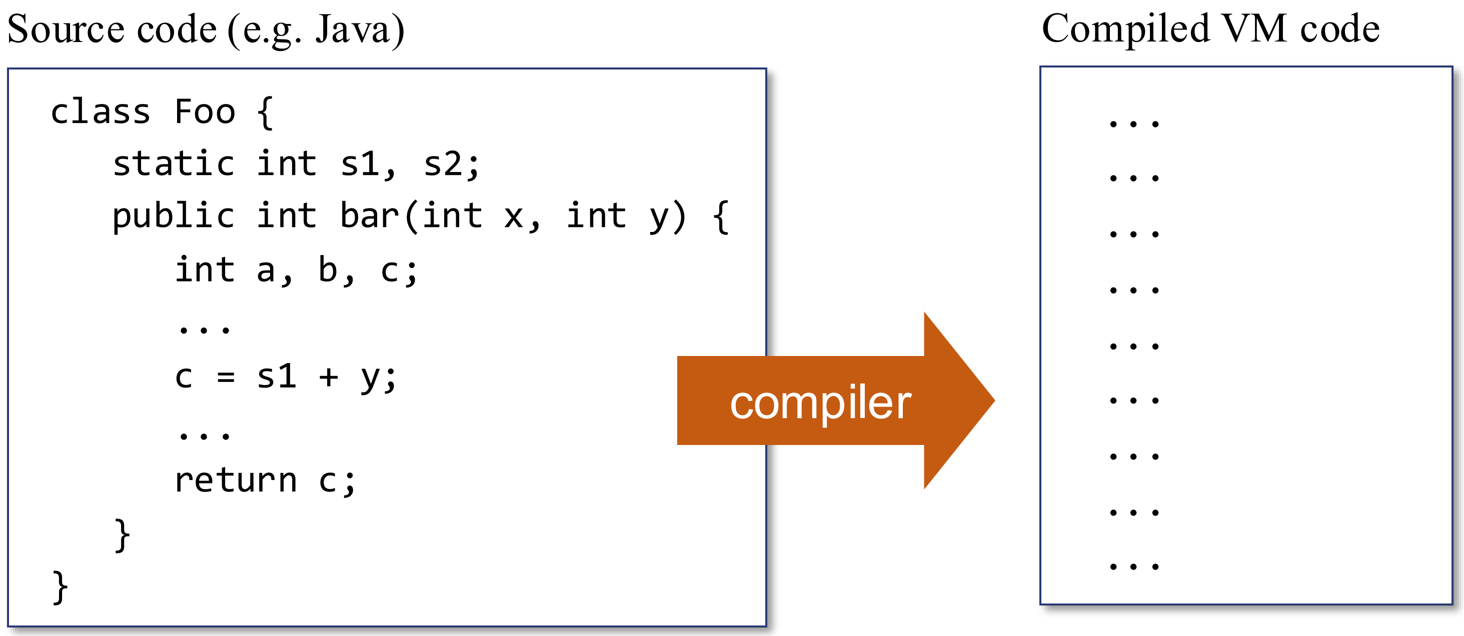
Function commands

`Function` *functionName nVars*

`Call` *functionName nArgs*

`return`

The Big Picture



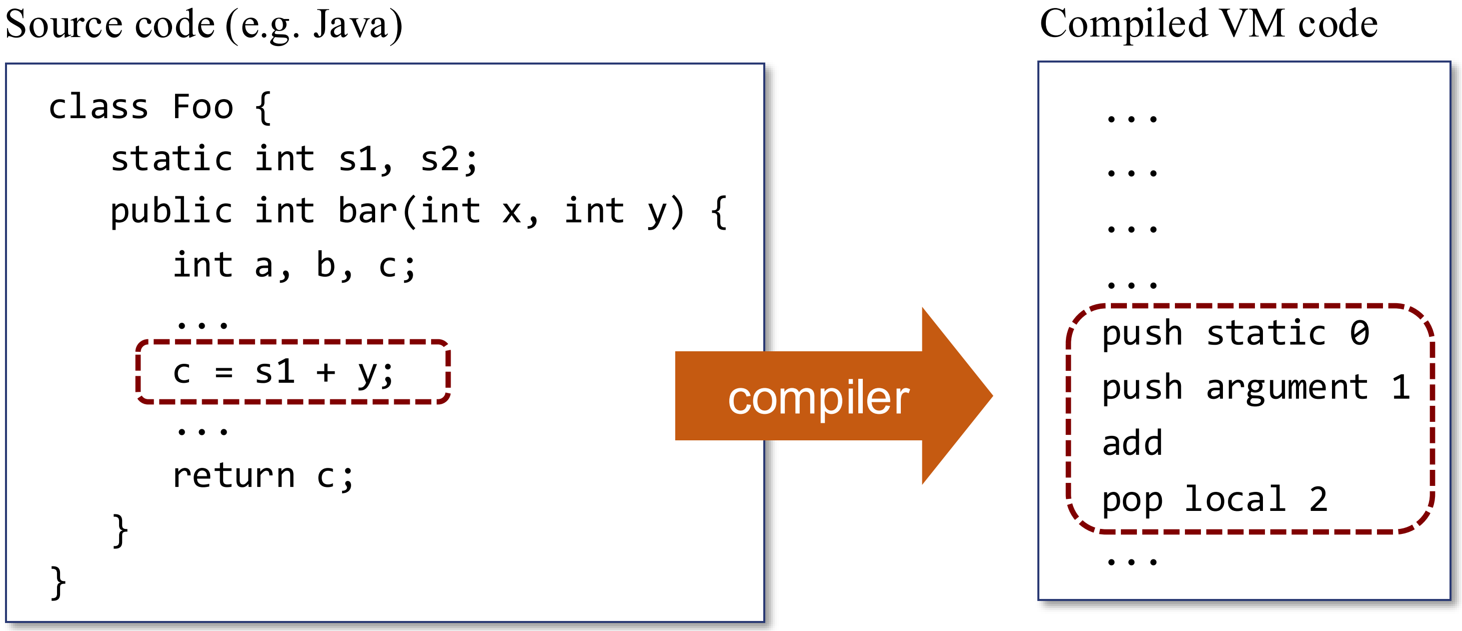
Source code (e.g. Java)

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

Compiled VM code

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

The Big Picture

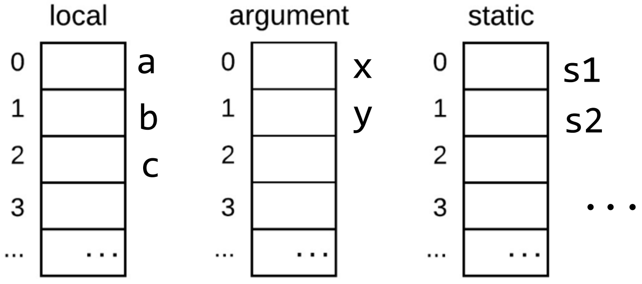


Source code (e.g. Java)

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

Compiled VM code

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



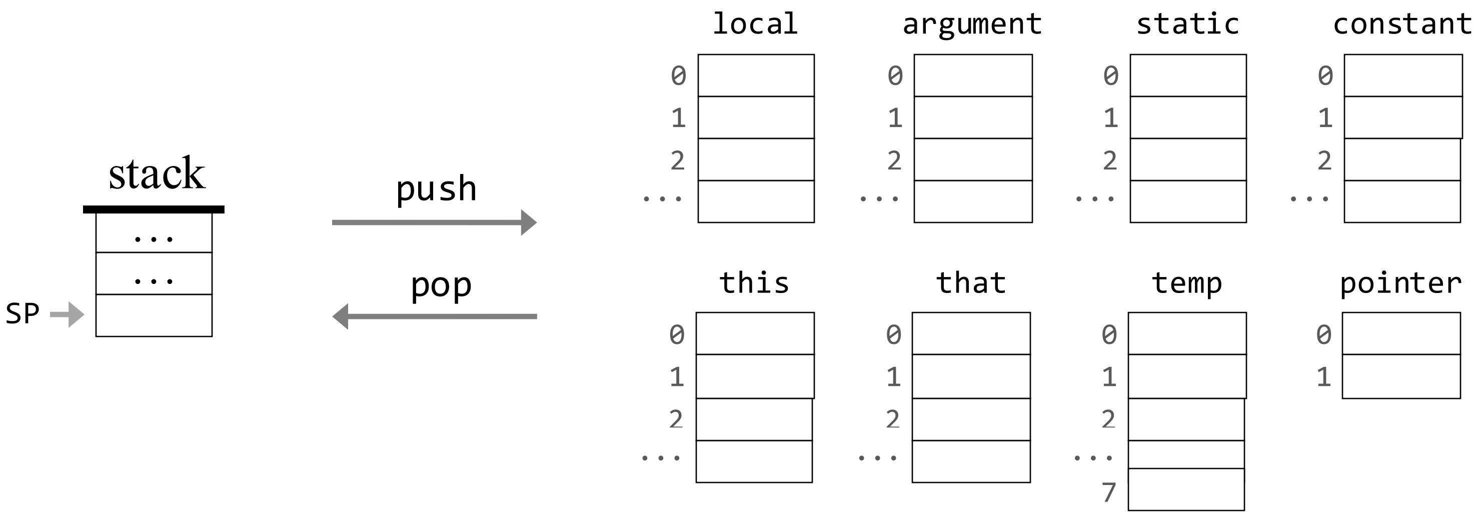
virtual memory segments

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.

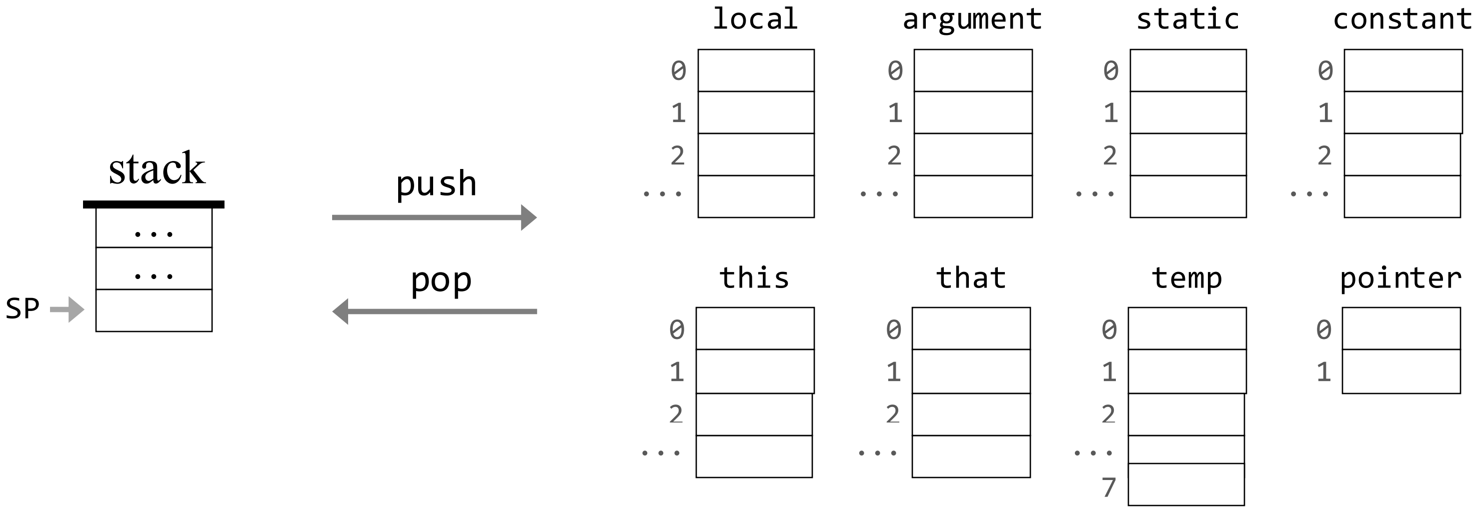
Virtual memory segments



Our VM architecture features 8 *virtual memory segments*

(their roles will become clear when we'll develop the compiler).

Virtual memory segments



Our VM architecture features 8 *virtual memory segments*
(their roles will become clear when we'll develop the compiler).

VM abstraction: All segments look and behave exactly the same:

push / **pop** ***segment i***

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

VM commands

Push / pop

`push` *segment i*

`pop` *segment i*

Arithmetic / Logical

`add`, `sub` , `neg`

`eq` , `gt` , `lt`

`and`, `or` , `not`

Example

```
// local 2 ← local 2 + argument 0
push local 2
push argument 0
add
pop local 2
```

Implementation options

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; 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 memory segments as dedicated RAM segments.

VM commands

Push / pop

`push` *segment i*

`pop` *segment i*

Arithmetic / Logical

`add`, `sub` , `neg`

`eq` , `gt` , `lt`

`and`, `or` , `not`

Implementation options

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;
Implement the VM commands as methods that operate on these ADTs;

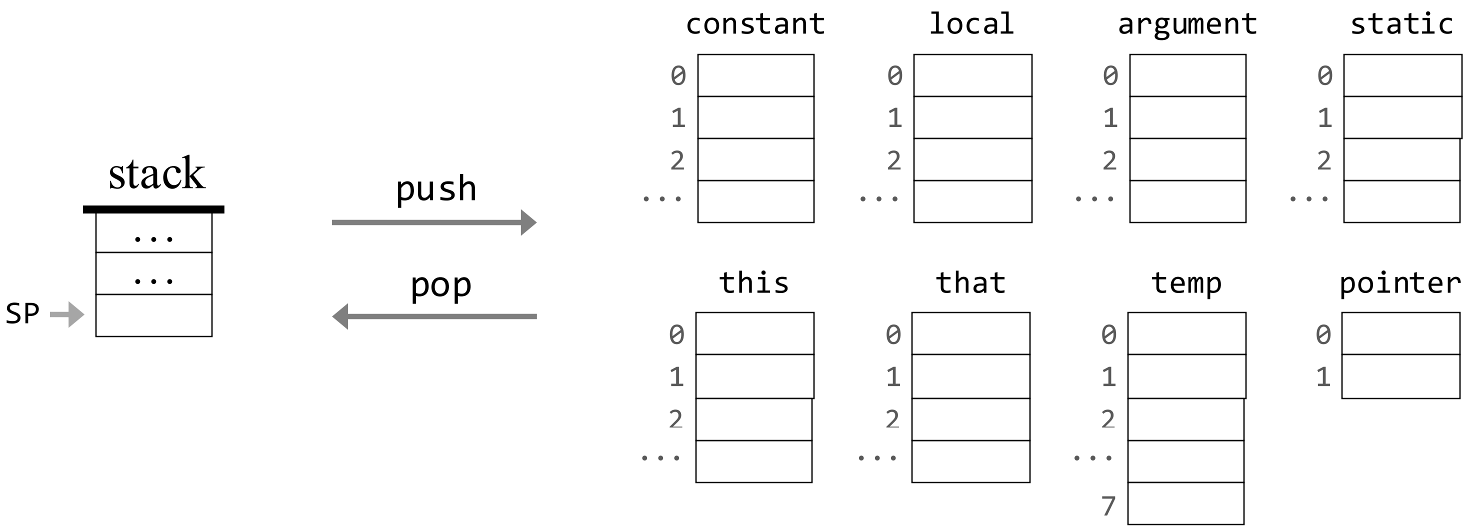
Translation: Translate each VM command into machine language instructions that operate on a host RAM;
Use an addressing contract that realizes the stack and the memory segments as dedicated RAM segments.

The approach taken by:

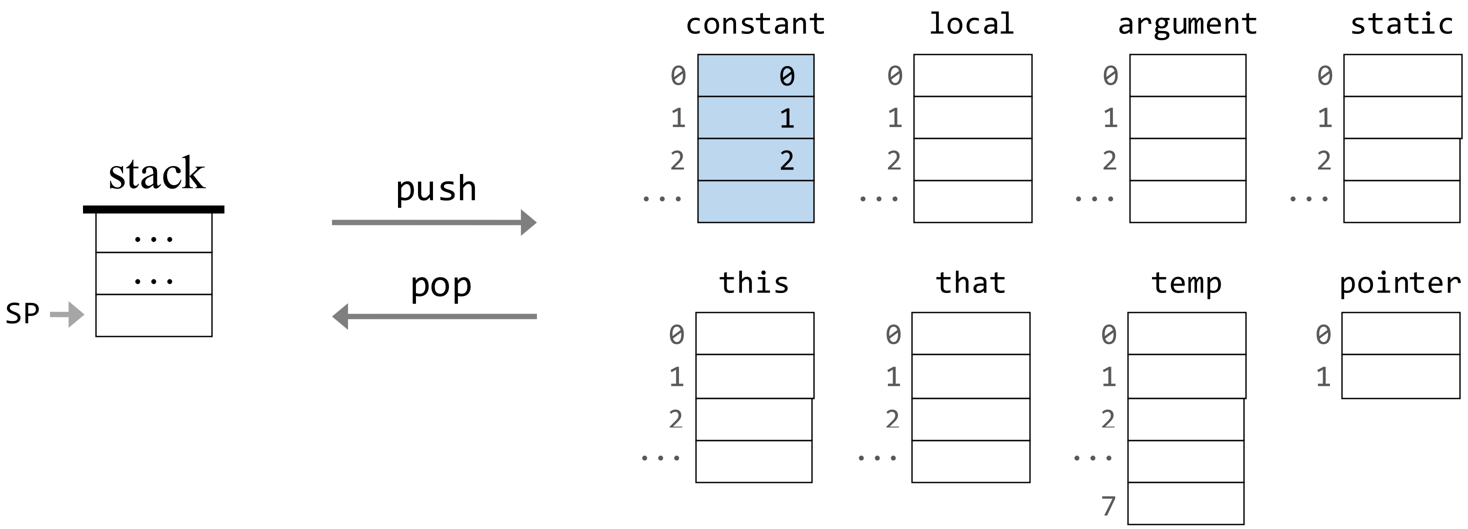
- Java, C#, Python, Ruby, Scala, ...
- Jack (designed in Nand to Tetris)

We'll start with implementing
the push / pop commands.

Push / pop commands



Implementing `push constant` *i*

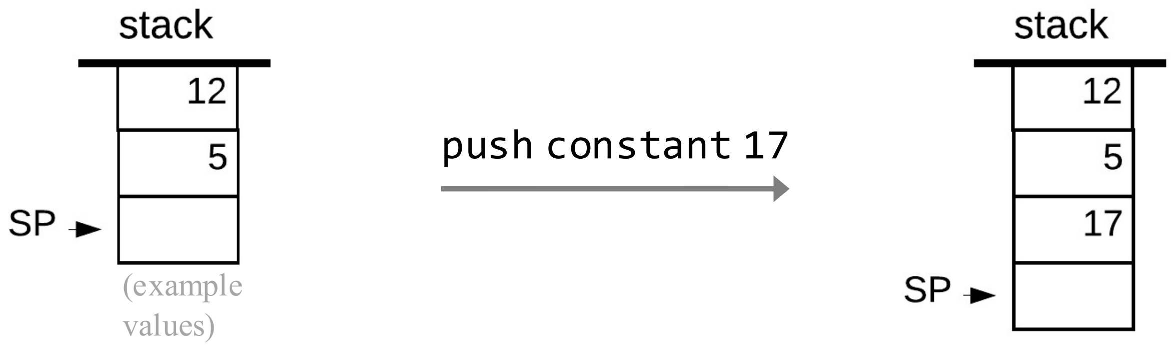


<u>The `constant` segment</u> represents the integers, `0`, `1`, `2`, `3`, ...

<u>Abstraction:</u> `constant` *i* supplies the integer *i*

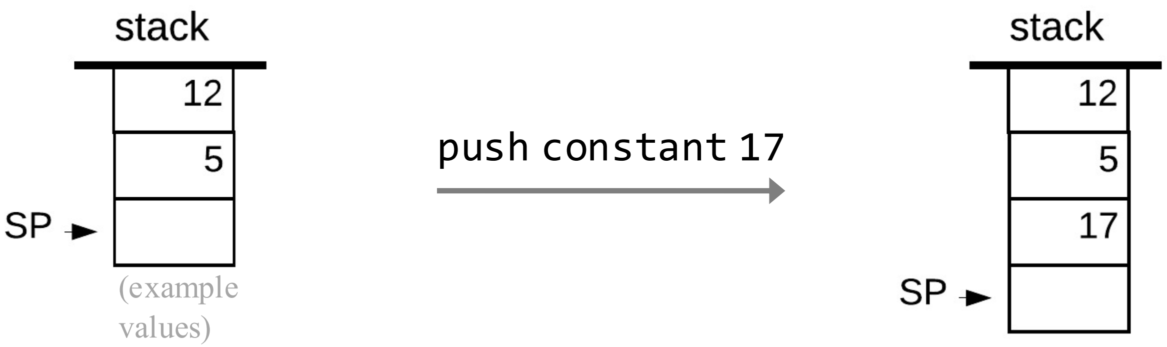
Implementing `push constant` *i*

Abstraction



Implementing `push constant` *i*

Abstraction

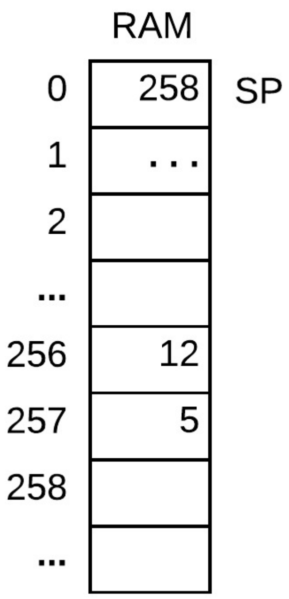


(example values)

Implementation

Stack:
stored in the RAM,
base address = 256

Stack Pointer:
stored in `RAM[0]`

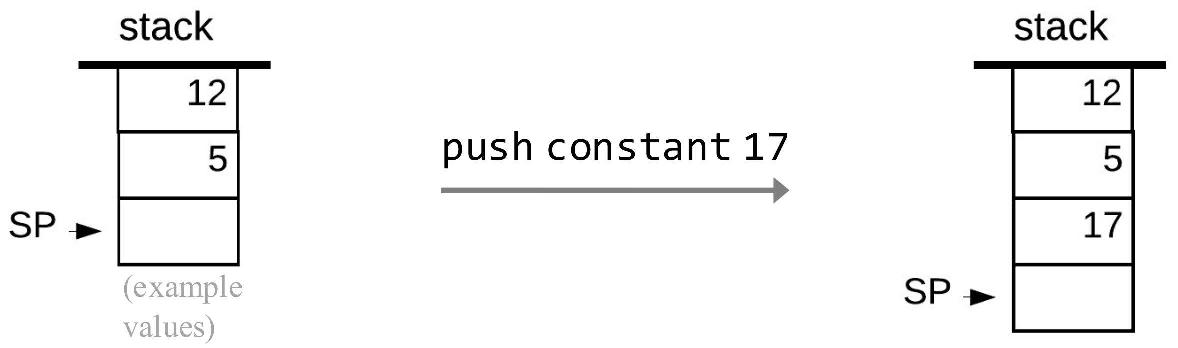


RAM		
0	258	SP
1	...	
2		
...		
256	12	
257	5	
258		
...		

(before)

Implementing `push constant` *i*

Abstraction



(example values)

Implementation

Stack:
stored in the RAM,
base address = 256

Stack Pointer:
stored in `RAM[0]`

RAM

	RAM	
0	258	SP
1	...	
2		
...		
256	12	
257	5	
258		
...		

(before)

pseudocode

```
// push constant 17
RAM[SP] = 17
SP++
```

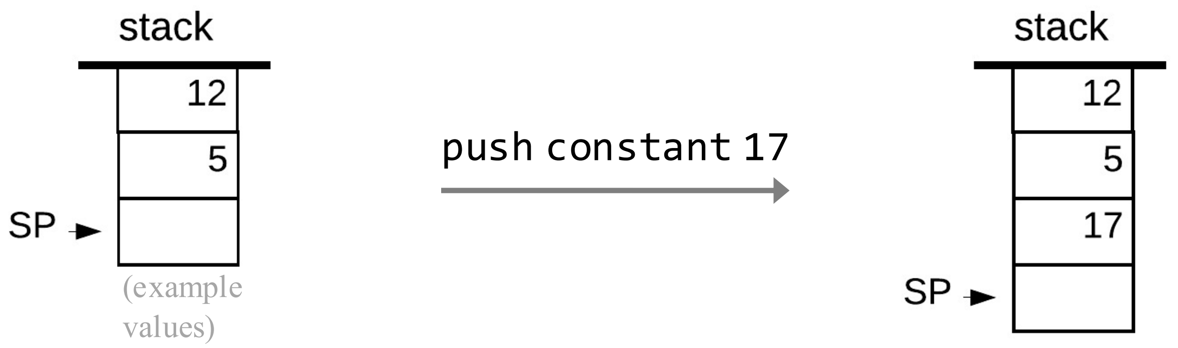
push constant 17 →

	RAM	
0	259	SP
1	...	
2		
...		
256	12	
257	5	
258	17	
...		

(after)

Implementing `push constant` *i*

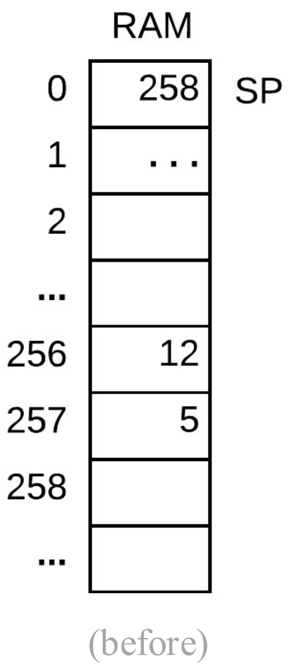
Abstraction



Implementation

Stack:
stored in the RAM,
base address = 256

Stack Pointer:
stored in RAM[0]

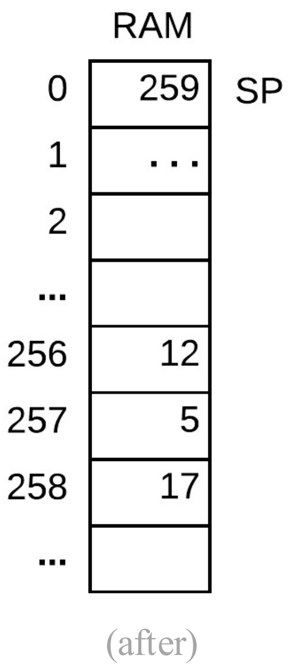


pseudocode

```
// push constant 17
RAM[SP] = 17
SP++
```

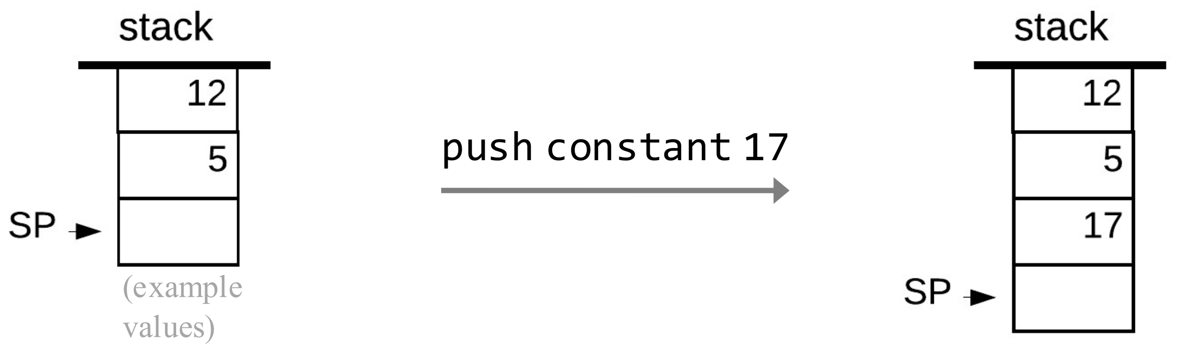
Hack assembly

```
// D = 17
@17
D=A
// RAM[SP] = D
@SP
A=M
M=D
// SP++
@SP
M=M+1
```



Implementing `push constant` *i*

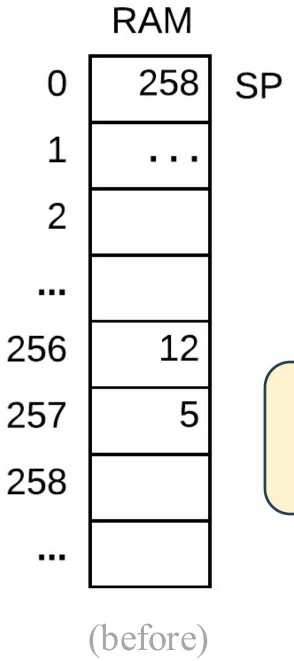
Abstraction



Implementation

Stack:
stored in the RAM,
base address = 256

Stack Pointer:
stored in `RAM[0]`



pseudocode

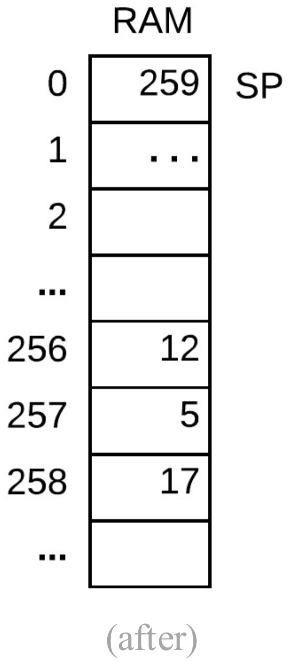
```
// push constant 17
RAM[SP] = 17
SP++
```

Hack assembly

```
// D = 17
@17
```

Let's generalize

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



Implementing `push constant` *i*

Abstraction

VM code

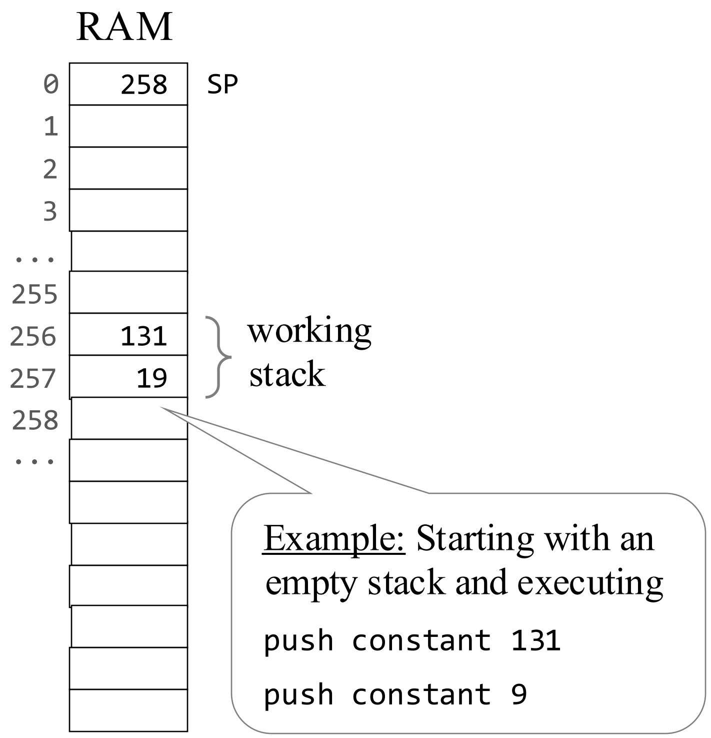
```
push constant i
```

VM translator

Implementation

Assembly code

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



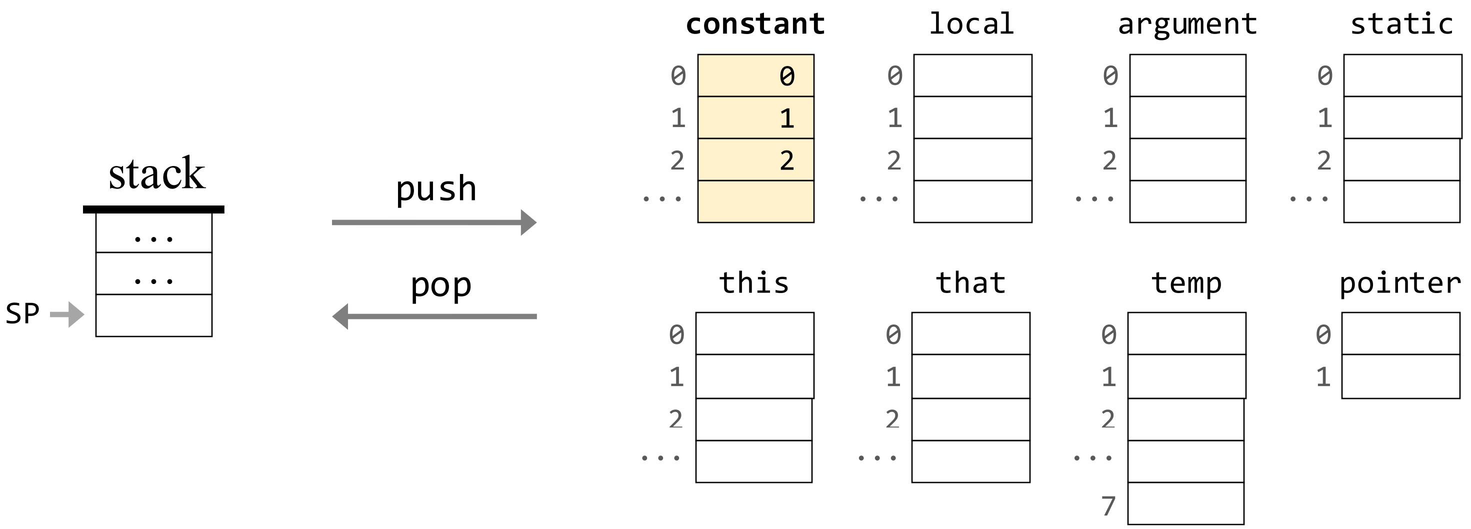
RAM

0	258	SP
1		
2		
3		
...		
255		
256	131	working stack
257	19	
258		
...		

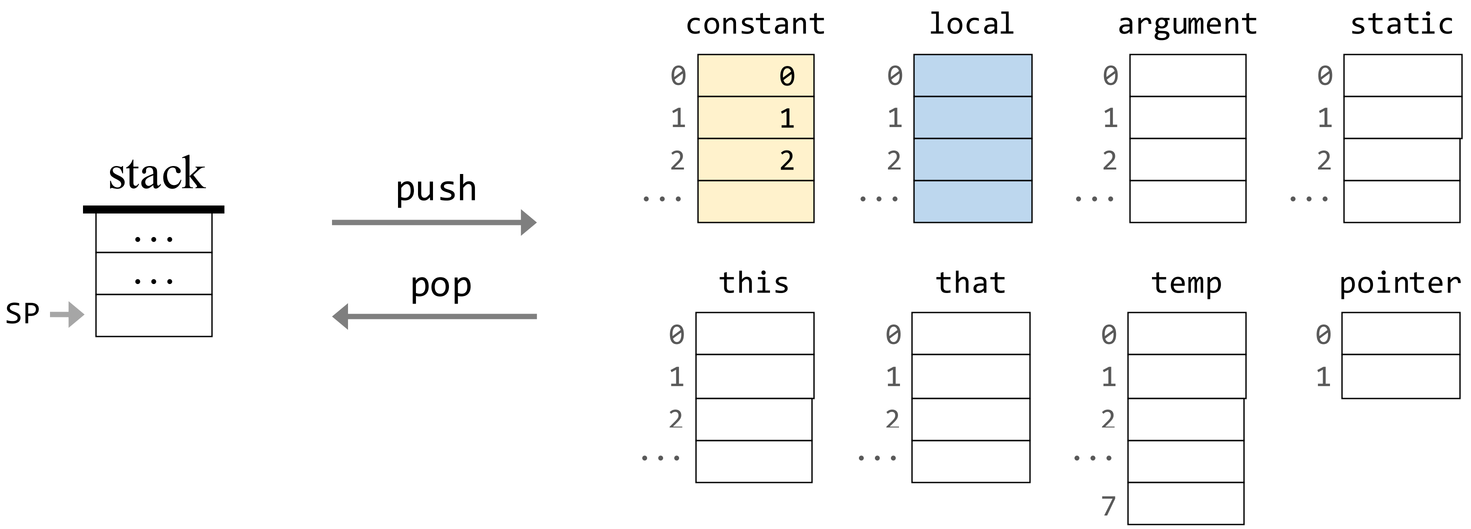
Example: Starting with an empty stack and executing

```
push constant 131
push constant 9
```

Implementing `push constant` *i*

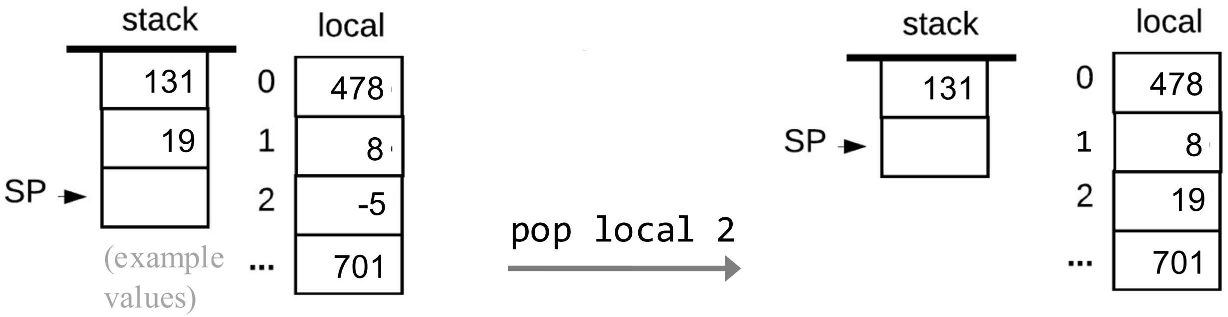


Implementing `push / pop local` *i*



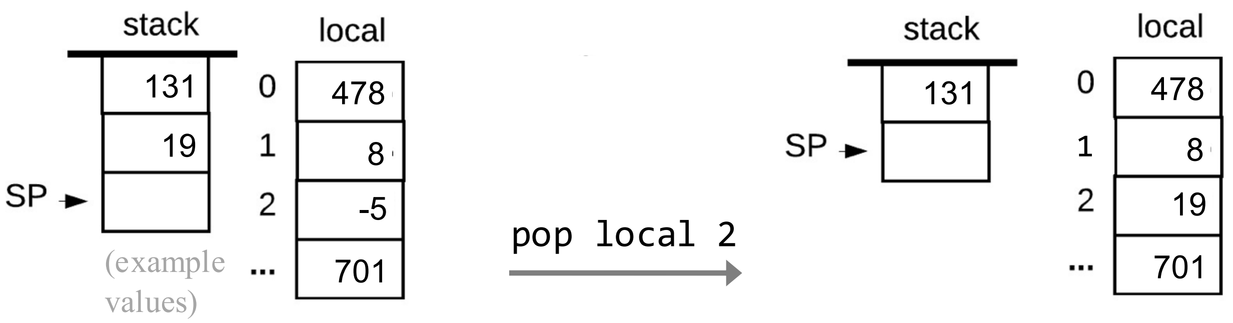
Implementing `push` / `pop local` *i*

Abstraction



Implementing `push / pop local i`

Abstraction



stack

131
19
SP ►

(example values)

local

0	478
1	8
2	-5
...	701

`pop local 2` →

stack

131
SP ►

local

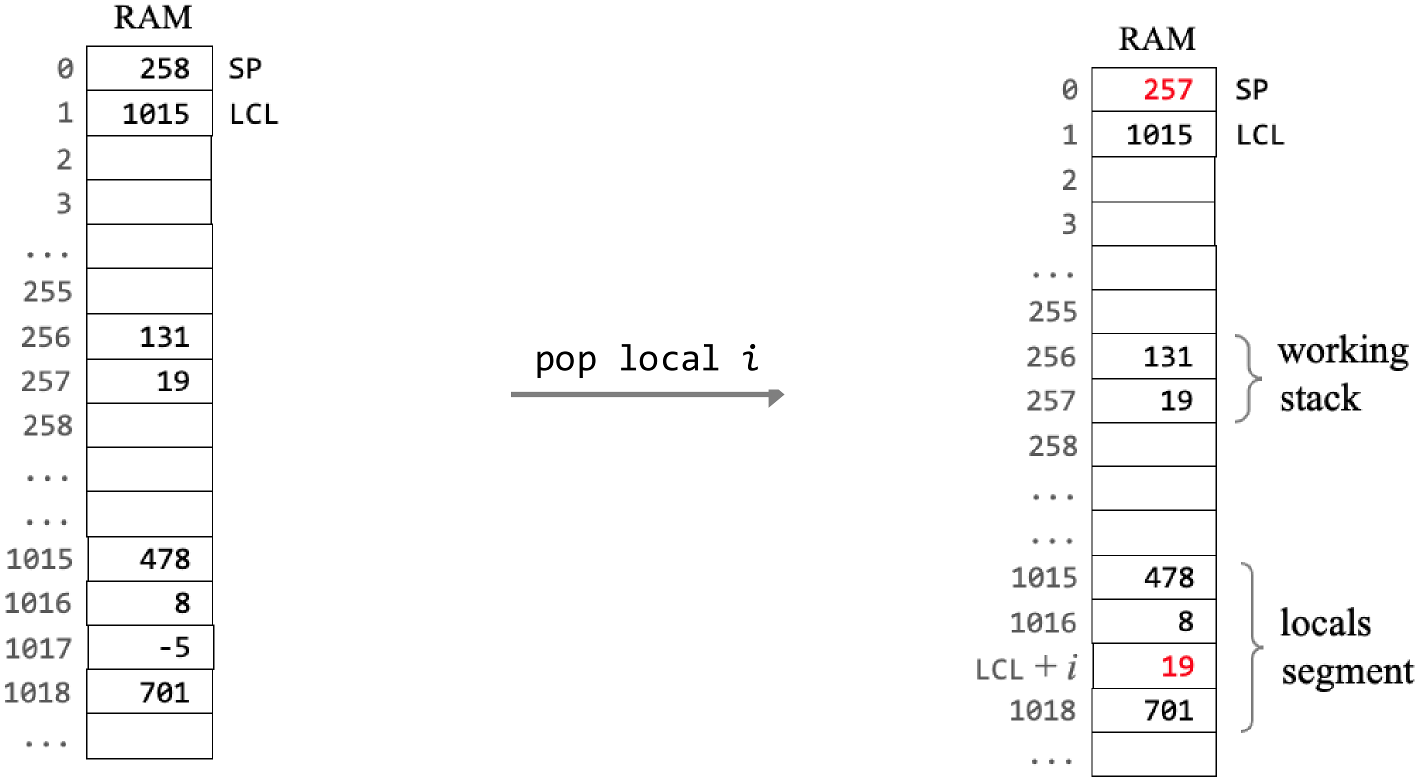
0	478
1	8
2	19
...	701

Implementation

locals segment: stored somewhere in the RAM;

`LCL` = base address

(1015 is an example)



RAM

0	258	SP
1	1015	LCL
2		
3		
...		
255		
256	131	
257	19	
258		
...		
...		
1015	478	
1016	8	
1017	-5	
1018	701	
...		

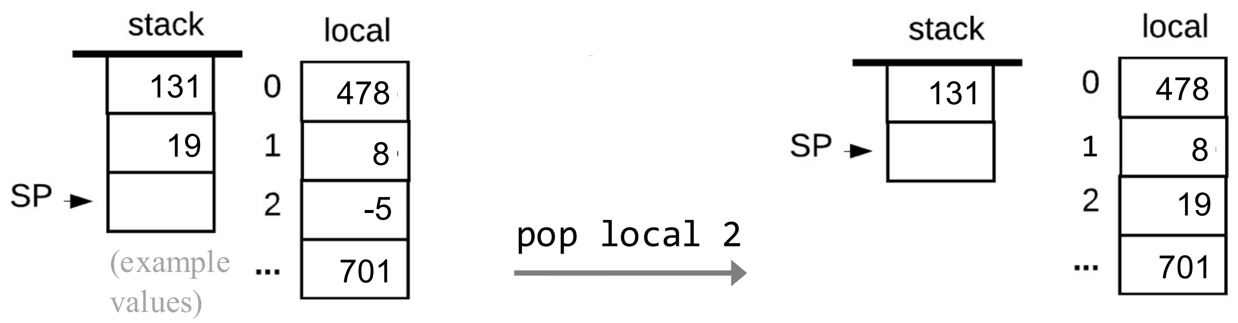
`pop local i` →

RAM

0	257	SP
1	1015	LCL
2		
3		
...		
255		
256	131	working stack
257	19	
258		
...		
...		
1015	478	locals segment
1016	8	
LCL + *i*	19	
1018	701	
...		

Implementing `push` / `pop local` *i*

Abstraction



stack

	131
	19
SP ►	

(example values)

local

0	478
1	8
2	-5
...	701

`pop local 2`

stack

	131
SP ►	

local

0	478
1	8
2	19
...	701

Implementation

locals segment: stored somewhere in the RAM;

`LCL` = base address

(1015 is an example)

RAM

0	258	SP
1	1015	LCL
2		
3		
...		
255		
256	131	
257	19	
258		
...		
...		
1015	478	
1016	8	
1017	-5	
1018	701	
...		

Pseudocode

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

`pop local i`

Hack assembly

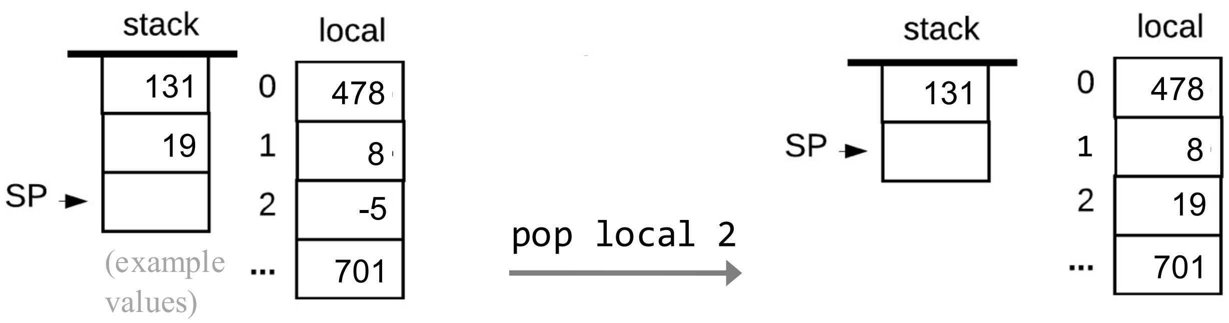
You do it!

RAM

0	257	SP	
1	1015	LCL	
2			
3			
...			
255			
256	131		working stack
257	19		
258			
...			
...			
1015	478		locals segment
1016	8		
LCL + *i*	19		
1018	701		
...			

Implementing `push / pop local` *i*

Abstraction



Implementation

locals segment: stored somewhere in the RAM;

`LCL` = base address

(1015 is an example)

RAM

0	258	SP
1	1015	LCL
2		
3		
...		
255		
256	131	
257	19	
258		
...		
...		
1015	478	
1016	8	
1017	-5	
1018	701	
...		

Pseudocode

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

`pop local i`

Hack assembly

Let's generalize

RAM

0	257	SP	
1	1015	LCL	
2			
3			
...			
255			
256	131		working stack
257	19		
258			
...			
...			
1015	478		locals segment
1016	8		
LCL + *i*	19		
1018	701		
...			

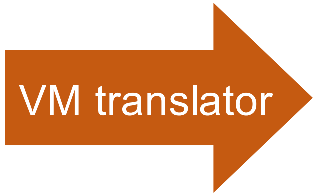
Implementing `push` / `pop local` *i*

Abstraction

VM code

```
pop local i
```

```
push local i
```



Implementation

Assembly pseudo code

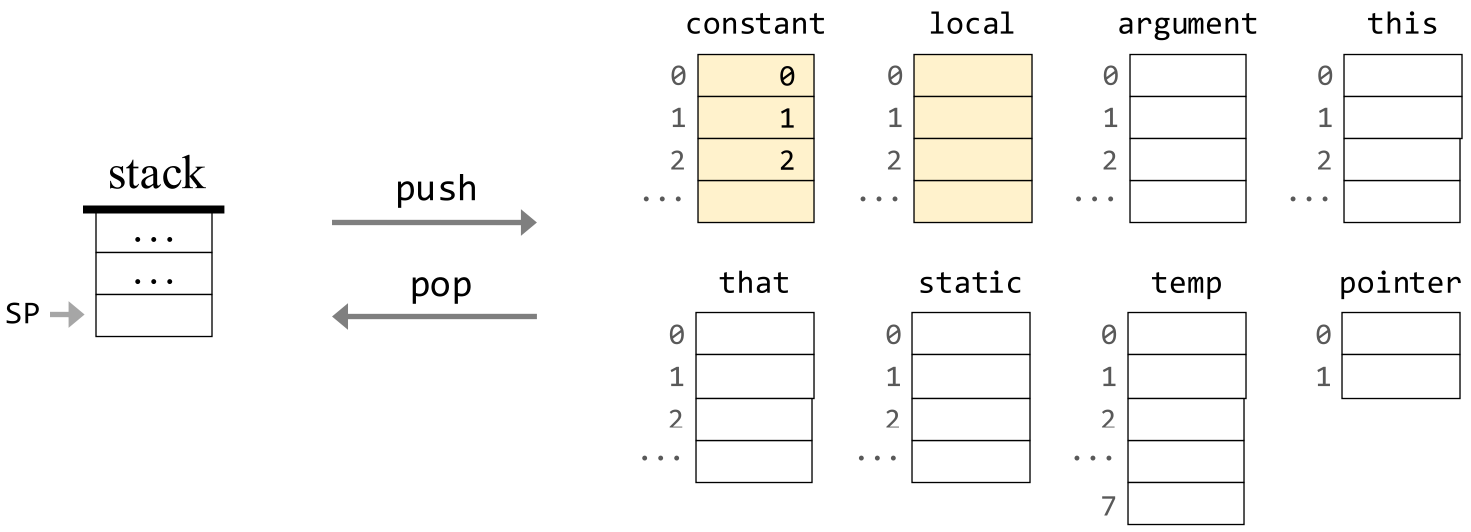
```
// pop local i
addr ← LCL + i
SP--
RAM[addr] ← RAM[SP]
```

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

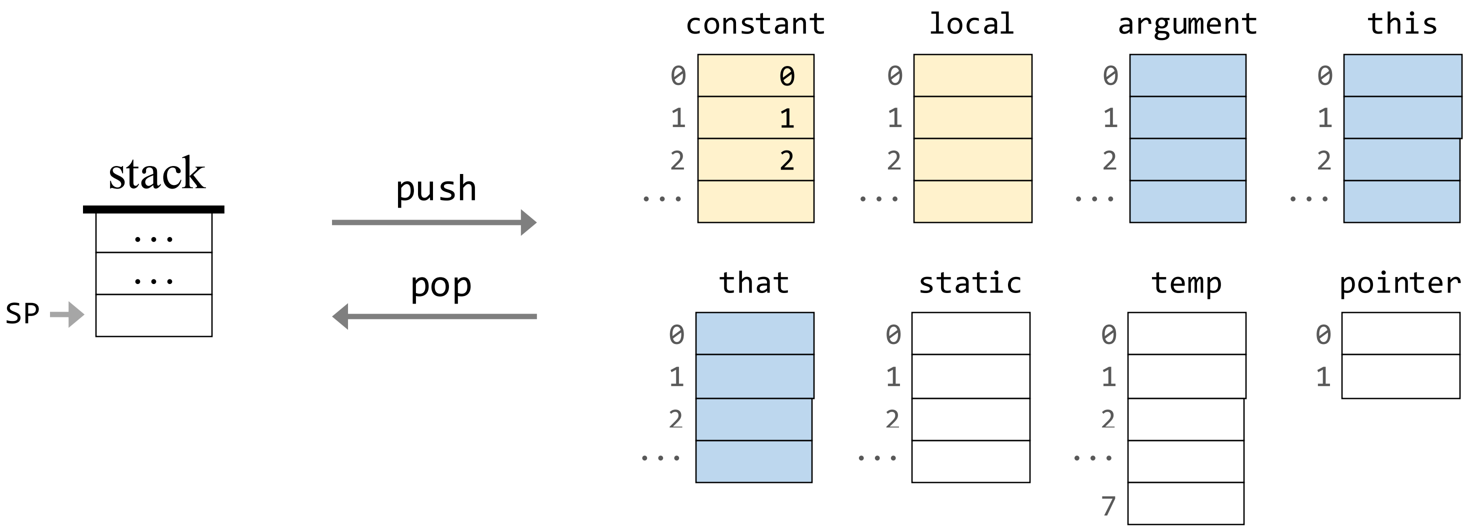
RAM

0	258	SP
1	1015	LCL
2		
3		
...		
255		
256	131	working stack
257	19	
258		
...		
...		
1015	478	locals segment
1016	8	
1017	-5	
1018	701	
...		

Implementing `push` / `pop local` *i*



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



The segments `argument`, `this`, and `that`:

Implemented exactly the same way as `local`

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

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++
```

RAM

0	258	SP
1	1015	LCL
2		
3		
...		
255		
256	131	working stack
257	19	
258		
...		
...		
1015	478	locals segment
1016	8	
1017	-5	
1018	701	
...		

Implementation of `local` (reminder)

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

Abstraction

VM code

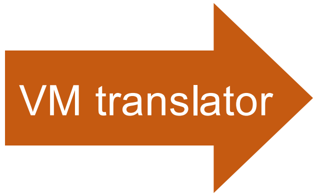
```
pop segment i
```

```
push segment i
```

where *segment* is

`local, argument, this, that`

and *i* is a non-negative integer



Implementation

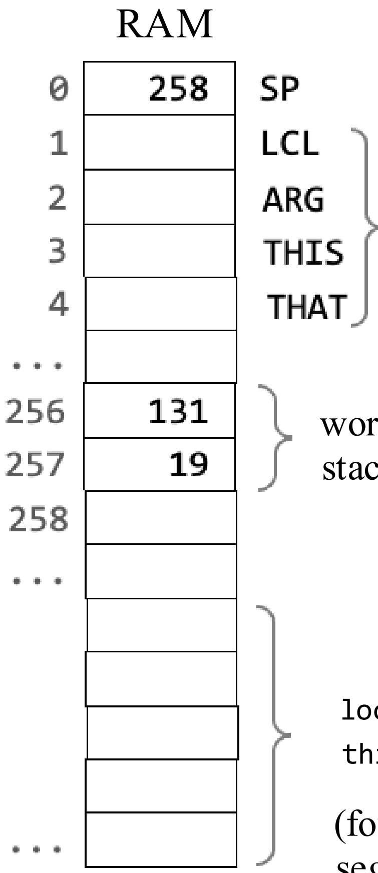
Assembly pseudo code

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

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

where *segmentPointer* is

`LCL, ARG, THIS, THAT`



base addresses of `local, argument, this, that`

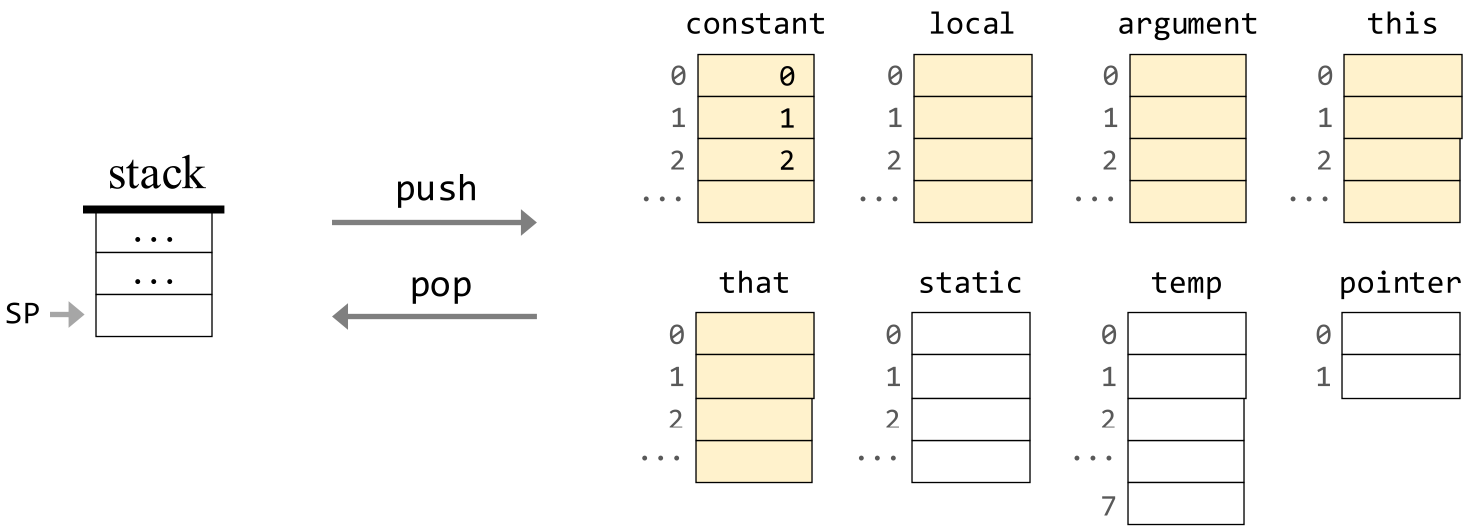
working stack

`local, argument, this, that`

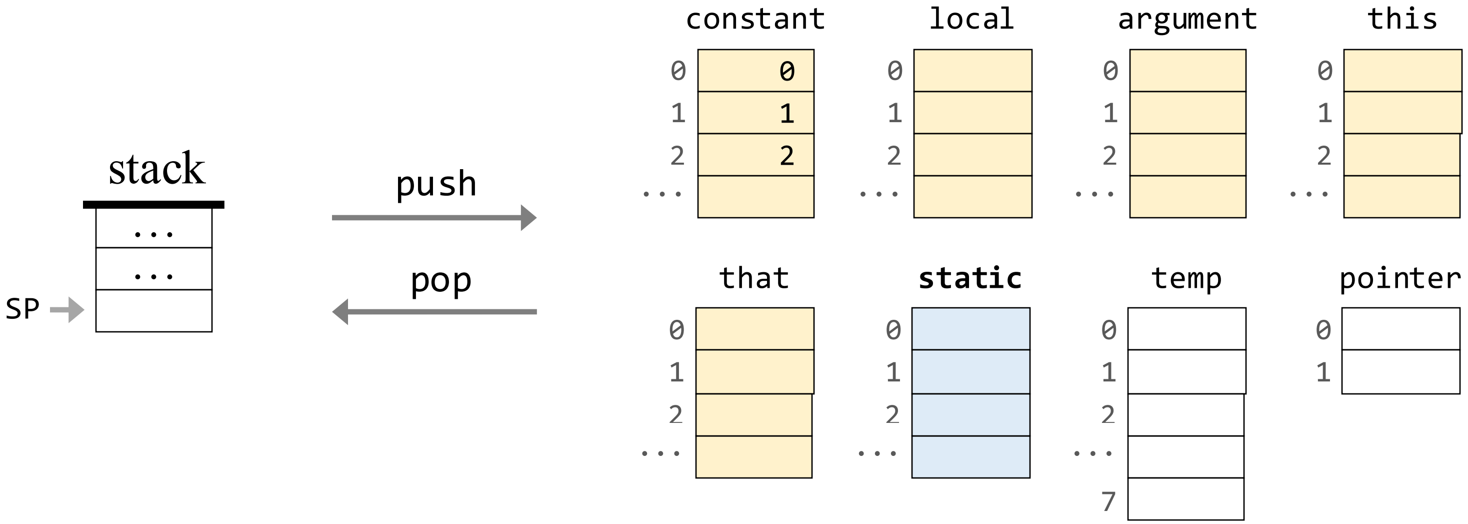
(four memory segments, each stored somewhere in the RAM)

Implementation of `local, argument, this, that`

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



Implementing `push / pop static` *i*



The Big Picture

When the compiler compiles classes, it maps all their *static variables* onto one VM segment, named `static`.

Implementing `push` / `pop static` *i*

Standard mapping (contract)

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

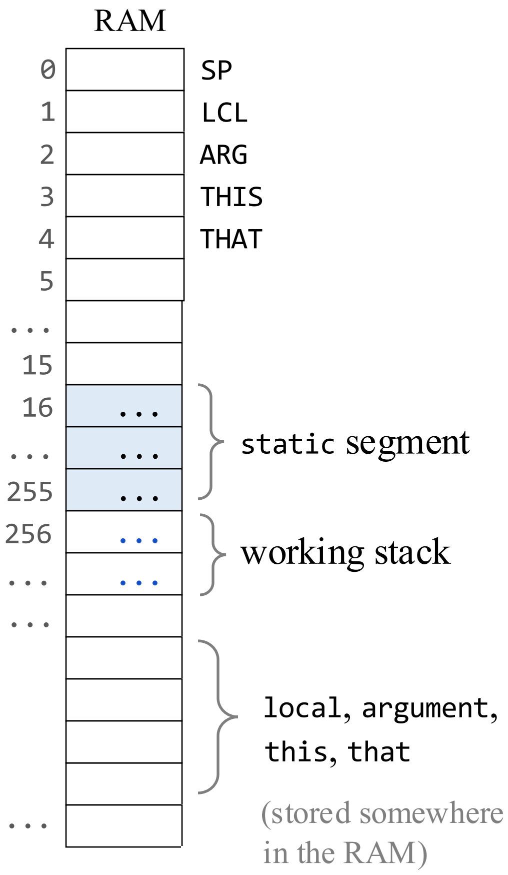
To translate `push`/`pop static` *i*

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

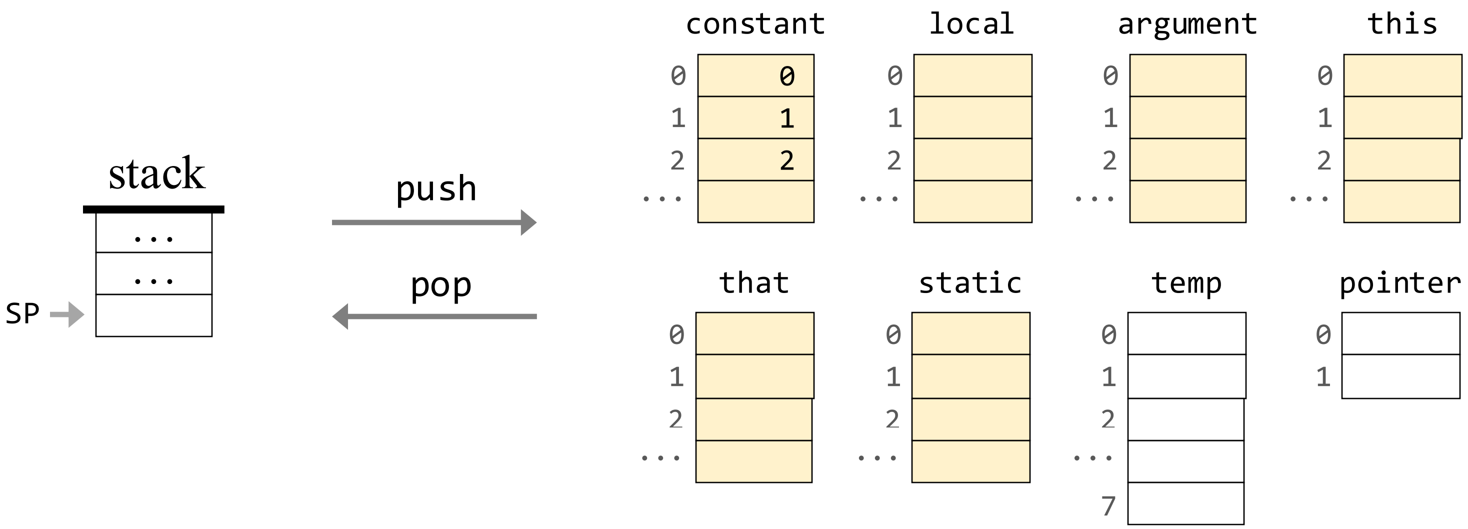
Generate assembly code that realizes:

`push`/`pop Xxx.`*i*

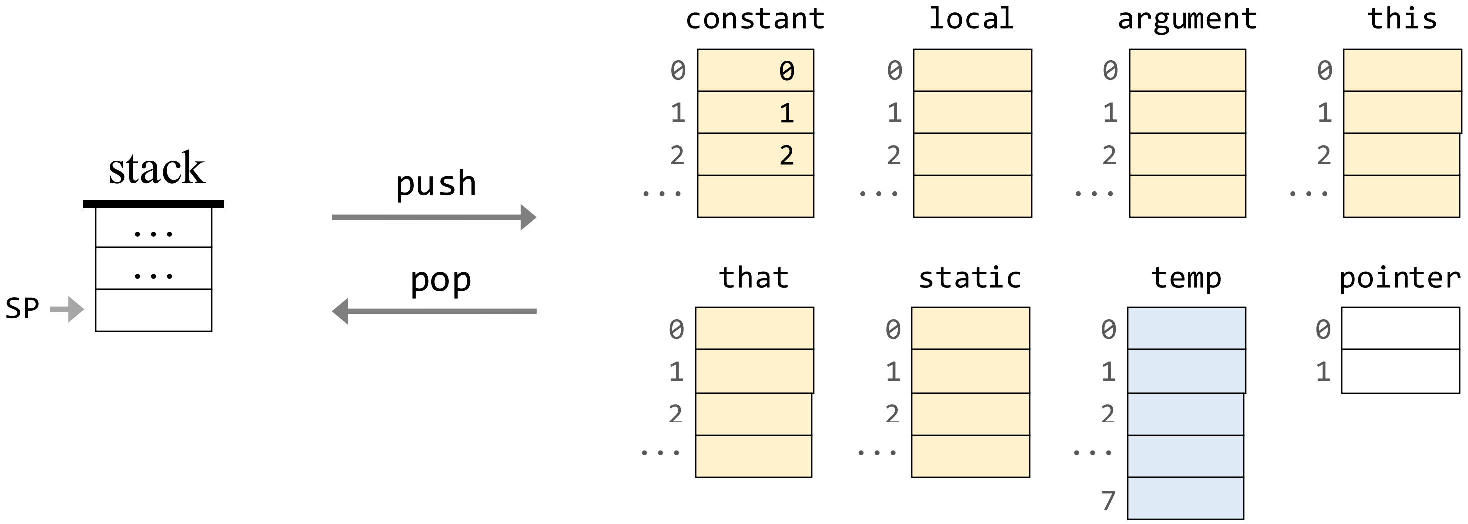
(Explanation: When this assembly code will be further translated to executable code, the Assembler will map these variables on RAM addresses 16, 17, 18, ..., exactly what we want).



Implementing `push / pop static` *i*



Implementing `push / pop temp` *i*



The Big Picture

When translating high-level code, compilers sometimes generate VM code that uses temporary variables (variables that don't come from the source code)

The `temp` segment: A fixed, 8-entry segment: `temp 0`, `temp 1`, …, `temp 7`

Implementing `push / pop temp` *i*

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]`

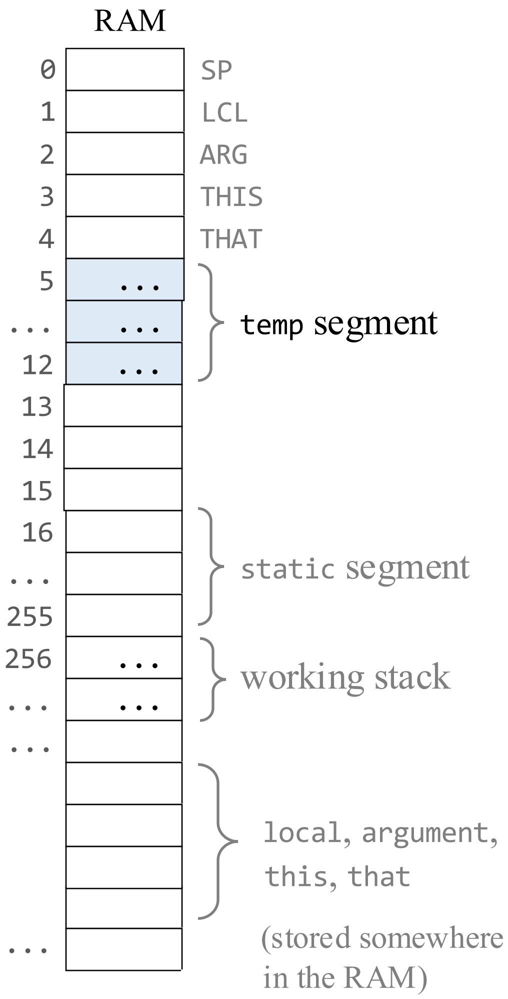
`...`

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

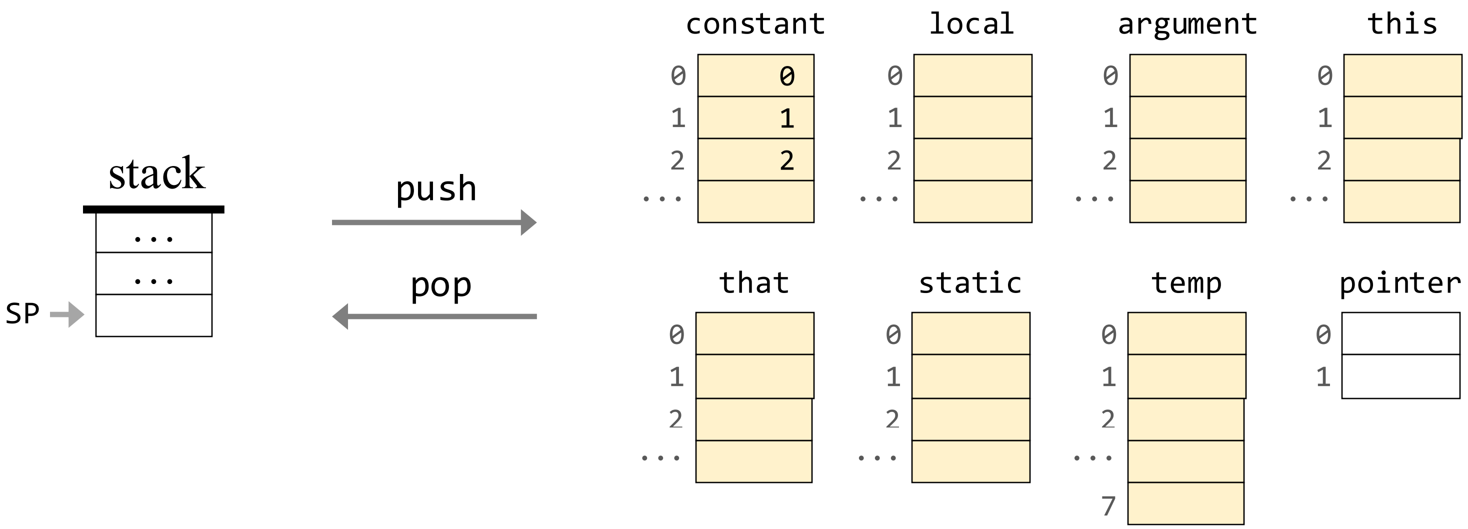
Implementing `push/pop temp` *i*

Generate assembly code that realizes:

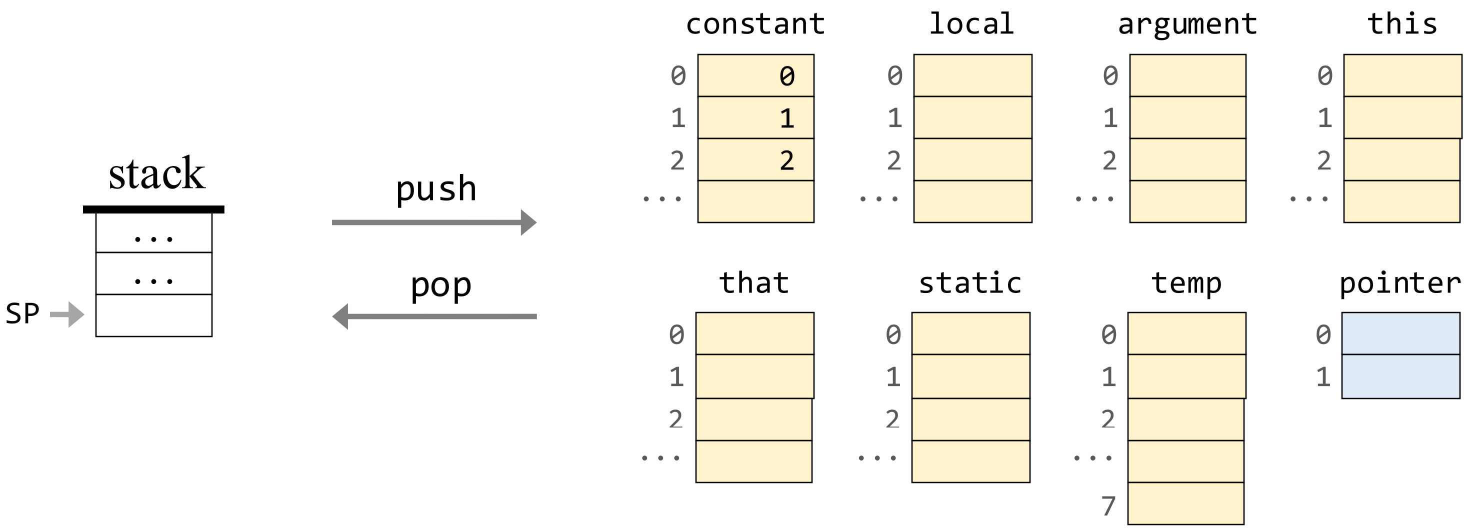
`push/pop RAM[5 +` *i*`]`



Implementing `push` / `pop temp` *i*



Implementing `push / pop pointer` *i*



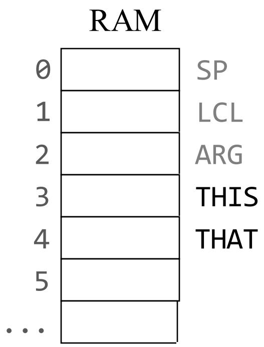
The Big Picture

The `pointer` segment comes to play when the compiler generates code that deals with *objects* and *arrays*;

More about this, when we learn how to write a compiler.

Abstraction: A fixed, 2-entry segment: `pointer 0`, `pointer 1`

Implementing `push / pop pointer` *i*



Abstraction

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

Implementation

(a truly virtual segment, not stored anywhere)

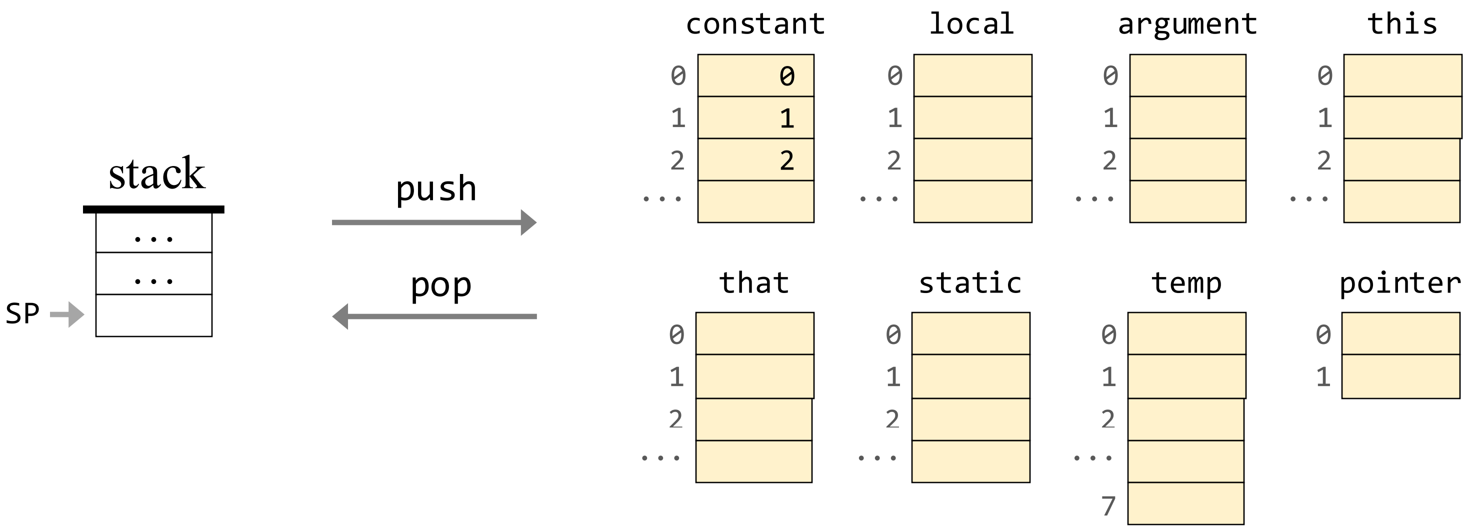
To translate: `push/pop pointer 0`

generate assembly code that realizes `push/pop THIS`

To translate: `push/pop pointer 1`

generate assembly code that realizes `push/pop THAT`

Push / pop commands



Recap

We described how to generate assembly code snippets that realize the VM operations

`push` / `pop` {`constant`, `local`, `argument`, `this`, `that`, `static`, `temp`, `pointer`} *i*

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`

Implementing the VM arithmetic-logical commands

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

Abstraction

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

Implementation

Popping implementation in assembly: Discussed

Pushing implementation is assembly: Discussed

`+`, `-`, `==`, `>`, `<`, `And`, `Or`, `Not` computations in assembly: Simple.

Conclusion

Translating the arithmetic-logical VM commands to assembly: Easy.

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
```

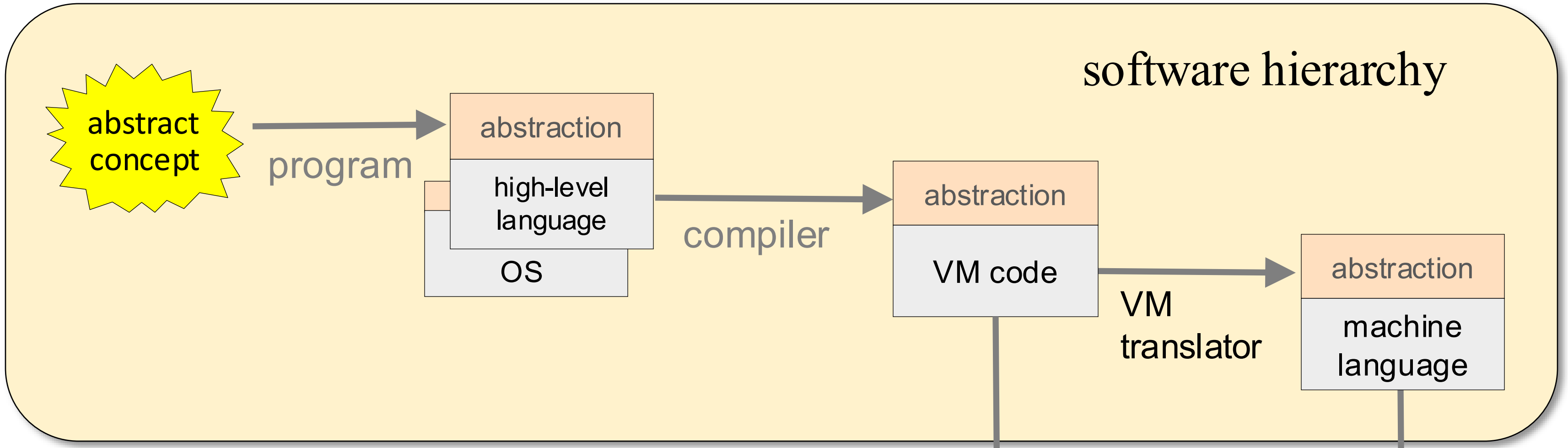
⇧

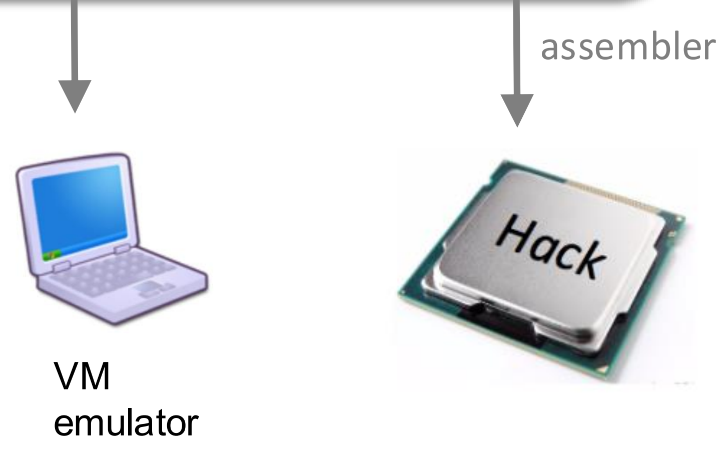
Next lecture

Lecture plan

- ✓ Overview
- ✓ The VM Language
- ➡ VM Emulator
- Standard Mapping
- VM Translator
- Project 7

VM implementations



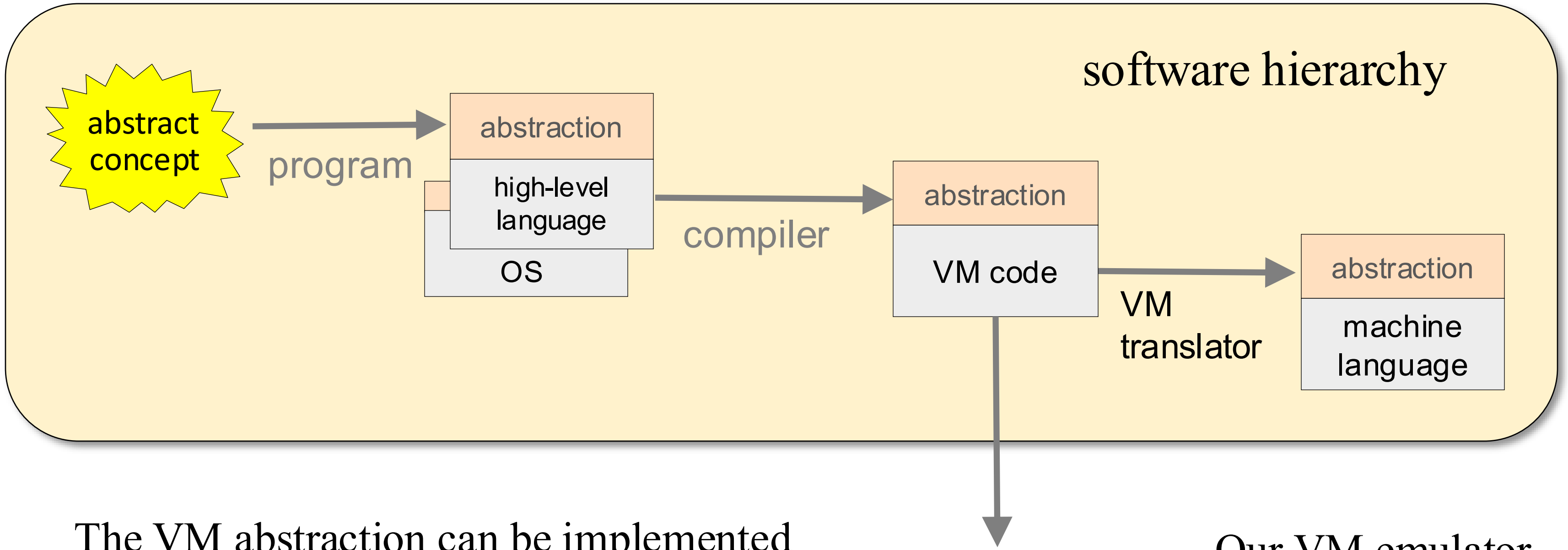


The VM abstraction can be implemented in several ways, including:

VM emulator: Uses a high-level language to execute VM commands (supplied)

VM translator: Translates VM commands into the machine language of a target platform (projects 7, 8).

VM implementations



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

<u>VM emulator</u>: Uses a high-level language to execute VM commands (supplied).

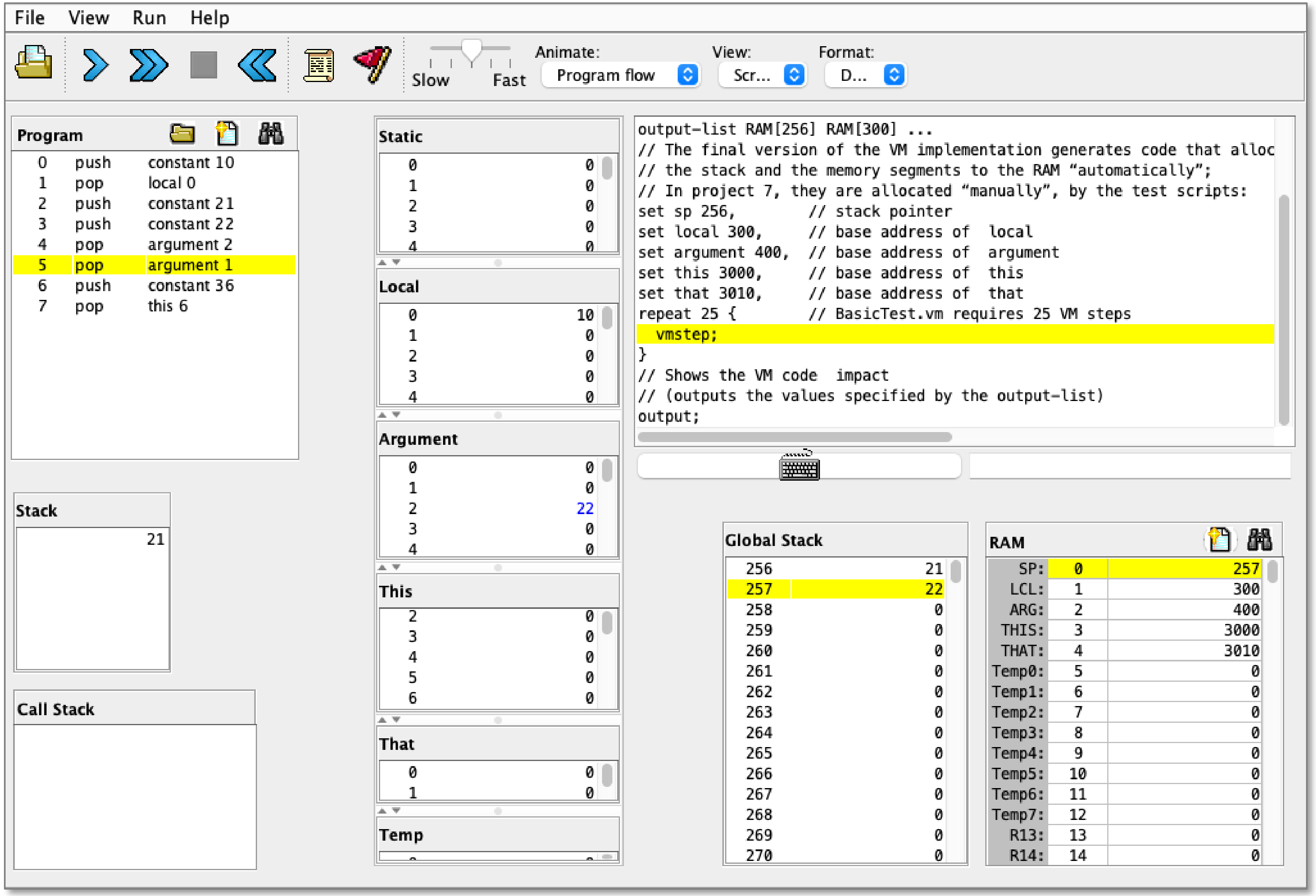
<u>Our VM emulator</u>

2. Executes VM programs

2. Visualizes...

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

Emulating a VM program



Animate: Program flow
View: Scr...
Format: D...

Program

```
0   push   constant 10
1   pop    local 0
2   push   constant 21
3   push   constant 22
4   pop    argument 2
5   pop    argument 1
6   push   constant 36
7   pop    this 6
```

```
output-list RAM[256] RAM[300] ...
// The final version of the VM implementation generates code that alloc
// 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;
```

Stack

```
21
```

Call Stack

Static

0	0
1	0
2	0
3	0
4	0

Local

0	10
1	0
2	0
3	0
4	0

Argument

0	0
1	0
2	22
3	0
4	0

This

2	0
3	0
4	0
5	0
6	0

That

0	0
1	0

Temp

Global Stack

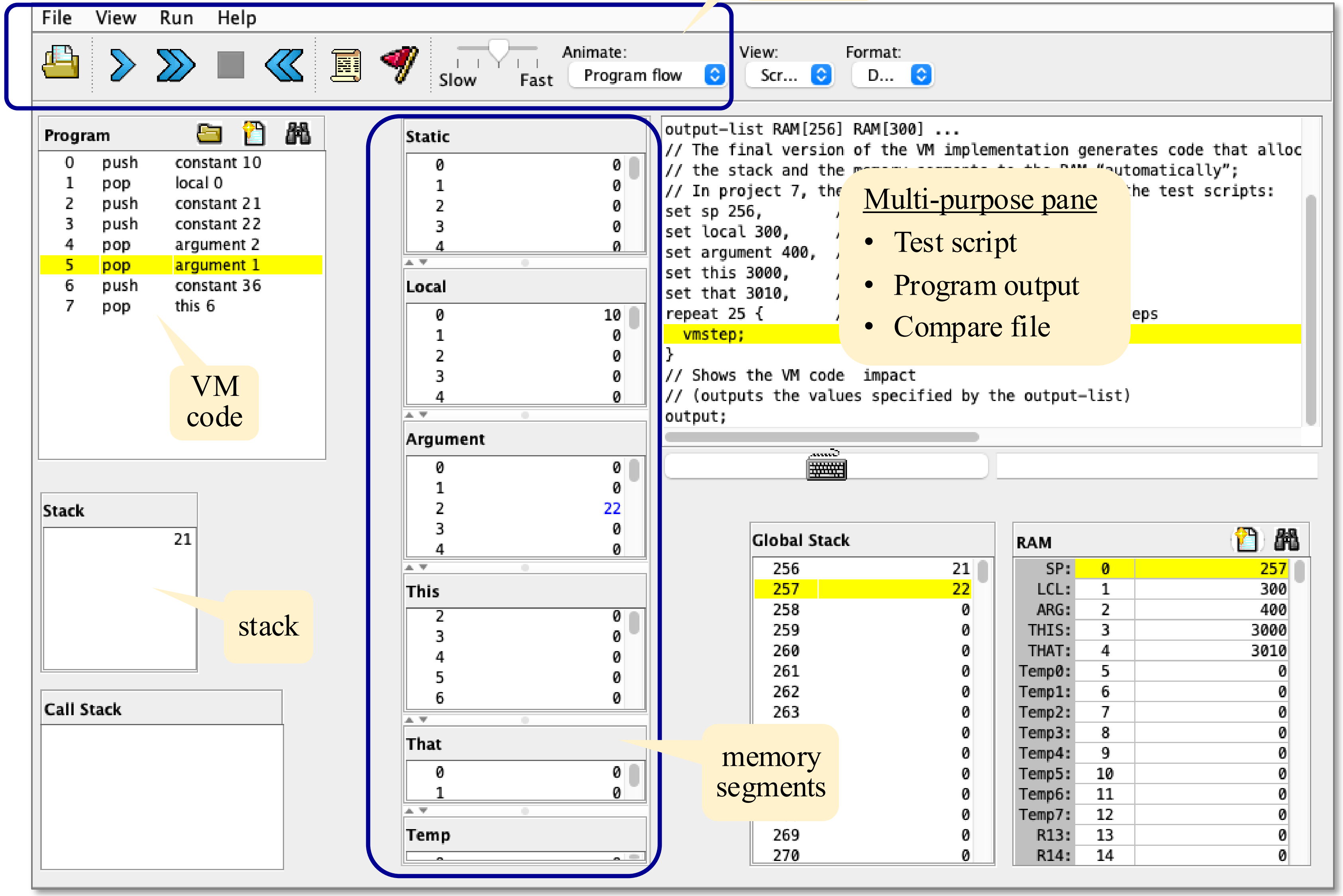
256	21
257	22
258	0
259	0
260	0
261	0
262	0
263	0
264	0
265	0
266	0
267	0
268	0
269	0
270	0

RAM

SP:	0	257
LCL:	1	300
ARG:	2	400
THIS:	3	3000
THAT:	4	3010
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

Emulating a VM program

execution controls



Program

```
0   push   constant 10
1   pop    local 0
2   push   constant 21
3   push   constant 22
4   pop    argument 2
5   pop    argument 1
6   push   constant 36
7   pop    this 6
```

VM code

Stack

21

stack

Static

0	0
1	0
2	0
3	0
4	0

Local

0	10
1	0
2	0
3	0
4	0

Argument

0	0
1	0
2	22
3	0
4	0

This

2	0
3	0
4	0
5	0
6	0

That

0	0
1	0

Temp

memory segments

```
output-list RAM[256] RAM[300] ...
// The final version of the VM implementation generates code that alloc
// the stack and the memory segments to the RAM "automatically";
// In project 7, the                                    the test scripts:
set sp 256,        /
set local 300,     /
set argument 400,  /
set this 3000,     /
set that 3010,     /
repeat 25 {        /                                    eps
  vmstep;
}
// Shows the VM code  impact
// (outputs the values specified by the output-list)
output;
```

Multi-purpose pane

- Test script
- Program output
- Compare file

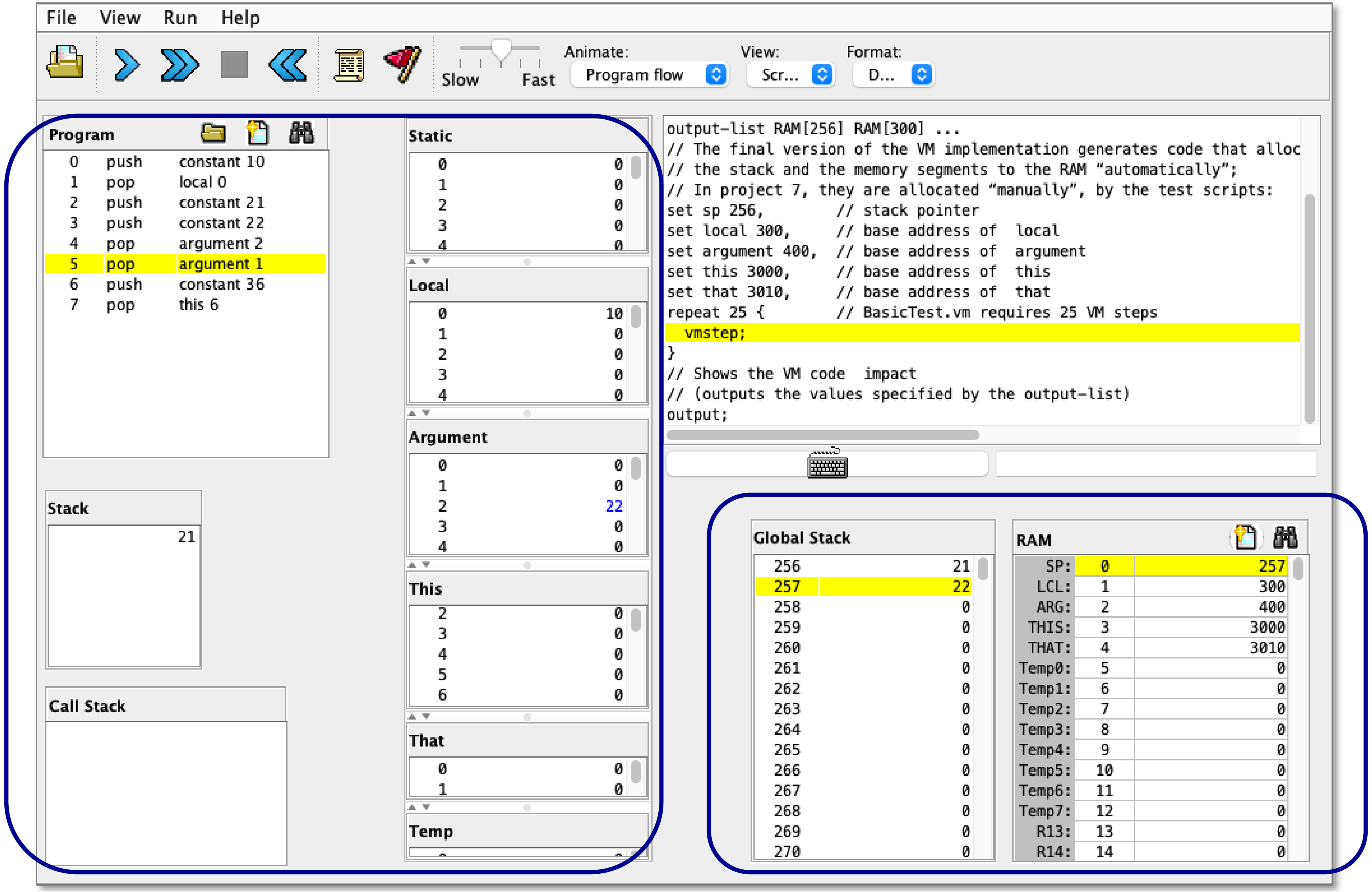
Global Stack

256	21
257	22
258	0
259	0
260	0
261	0
262	0
263	0
	0
	0
	0
	0
	0
269	0
270	0

RAM

SP:	0	257
LCL:	1	300
ARG:	2	400
THIS:	3	3000
THAT:	4	3010
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

Emulating a VM program



Animate: Program flow

View: Scr...

Format: D...

Program

```
0   push   constant 10
1   pop    local 0
2   push   constant 21
3   push   constant 22
4   pop    argument 2
5   pop    argument 1
6   push   constant 36
7   pop    this 6
```

```
output-list RAM[256] RAM[300] ...
// The final version of the VM implementation generates code that alloc
// 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;
```

Stack

21

Call Stack

Static

0	0
1	0
2	0
3	0
4	0

Local

0	10
1	0
2	0
3	0
4	0

Argument

0	0
1	0
2	22
3	0
4	0

This

2	0
3	0
4	0
5	0
6	0

That

0	0
1	0

Temp

Global Stack

256	21
257	22
258	0
259	0
260	0
261	0
262	0
263	0
264	0
265	0
266	0
267	0
268	0
269	0
270	0

RAM

SP:	0	257
LCL:	1	300
ARG:	2	400
THIS:	3	3000
THAT:	4	3010
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

Abstraction

How the abstraction is realized

Emulating a VM program: Testing

`BasicTest.vm`

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

(example test program from project 7)

`BasicTestVME.tst`

```
load BasicTest.vm,
output-file BasicTest.out,
compare-to BasicTest.cmp,

// In project 7 we allocate the stack and the virtual segments to the RAM
// “manually”, using test script commands (in project 8 we will develop
// the ability to do these allocations “automatically”):

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 impact of the executed VM code on selected RAM addresses
// (contents of selected pointers, virtual segments, etc.):
output-list RAM[256] RAM[300] RAM[401] RAM[402]...
output;
```

- The script runs the VM program on the VM emulator;
- Enables experimenting with the VM commands before implementing them in assembly.

Demo

Lecture plan

- Overview
- The VM Language
- VM Emulator
- ➡ Standard Mapping
- VM Translator
- Project 7

Standard VM mapping

Background

We've introduced a virtual machine (VM) model;

The VM can be implemented on numerous target platforms,
in numerous different ways.

Standard mapping (on some target platform)

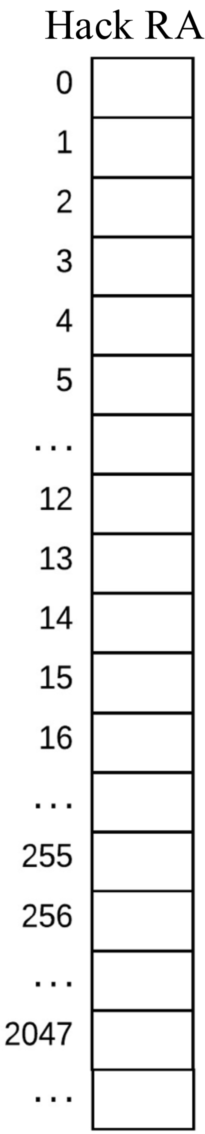
Recommends how to realize the VM on a specific target platform
(where to store the stack, the segments pointers, the segments, etc.)

Benefits

Promotes compatibility with other tools / libraries that conform to this standard:

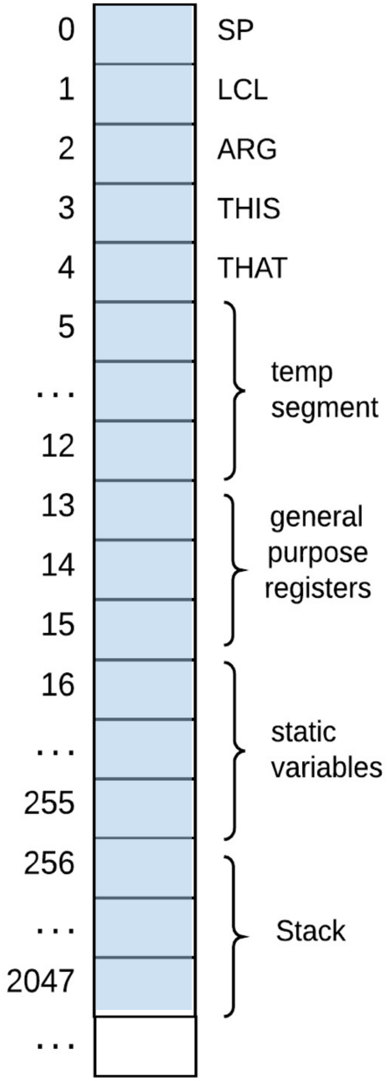
- VM emulators, OS routines
- Testing systems / test scripts
- Etc.

Standard VM mapping on the Hack platform

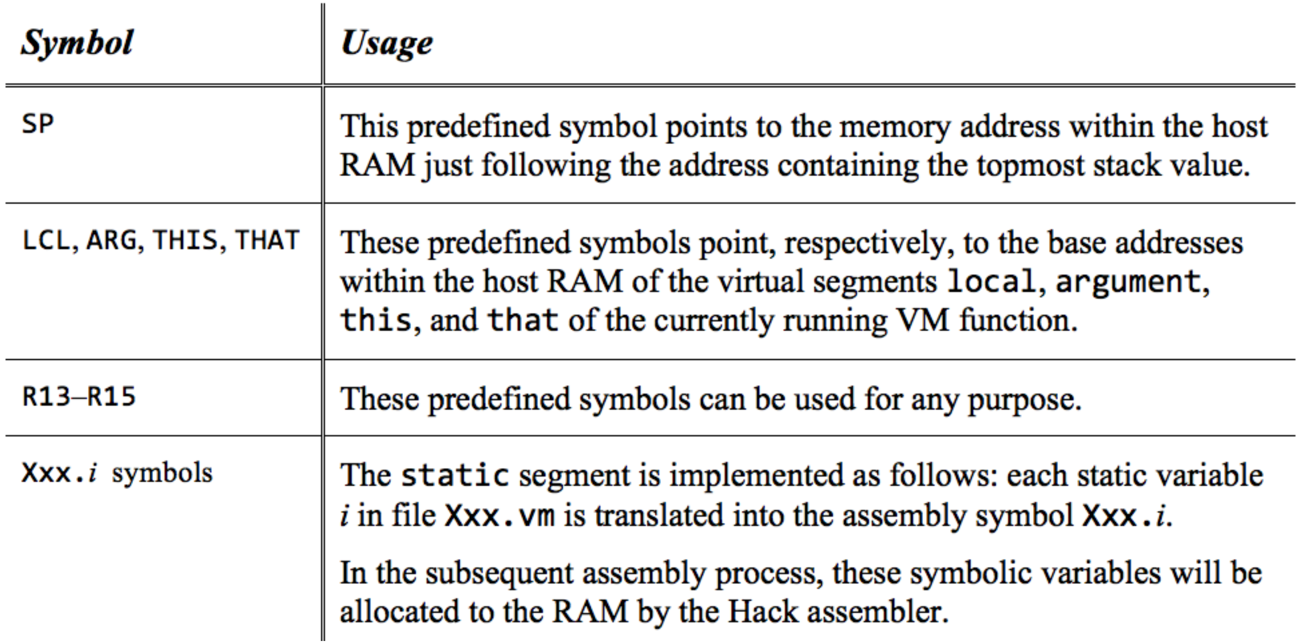


Standard VM mapping on the Hack platform

Hack RAM



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:

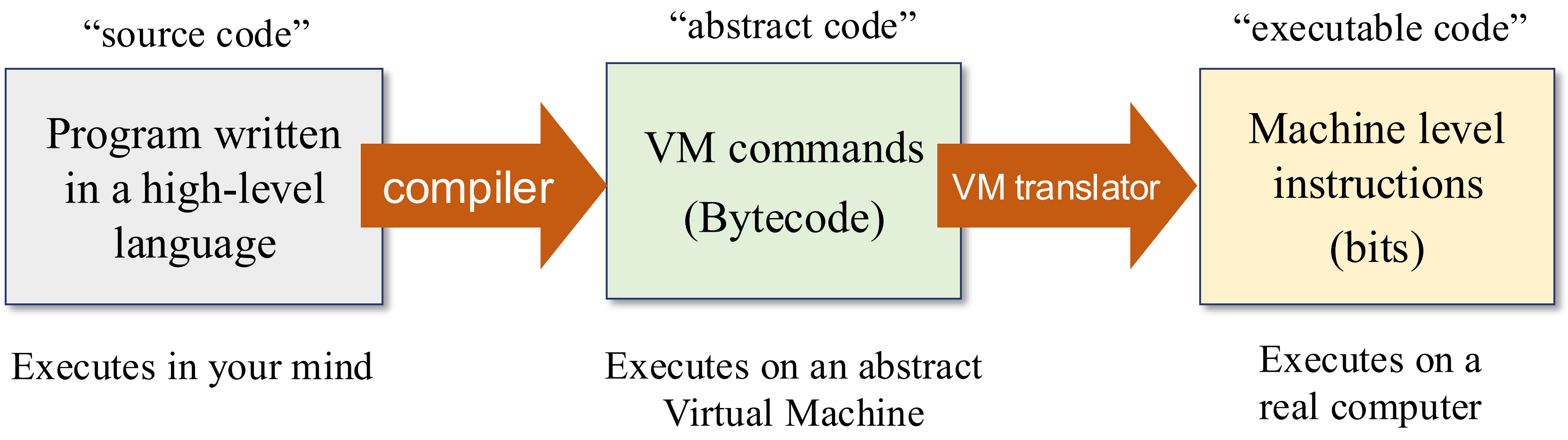


Symbol	***Usage***
`SP`	This predefined symbol points to the memory address within the host RAM just following the address containing the topmost stack value.
`LCL`, `ARG`, `THIS`, `THAT`	These predefined symbols point, respectively, to the base addresses within the host RAM of the virtual segments `local`, `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
- The VM Language
- VM Emulator
- Standard Mapping
- ➡ VM Translator
- Project 7

The Big Picture: Program compilation



“source code”

Program written in a high-level language

Executes in your mind

compiler

“abstract code”

VM commands

(Bytecode)

Executes on an abstract Virtual Machine

VM translator

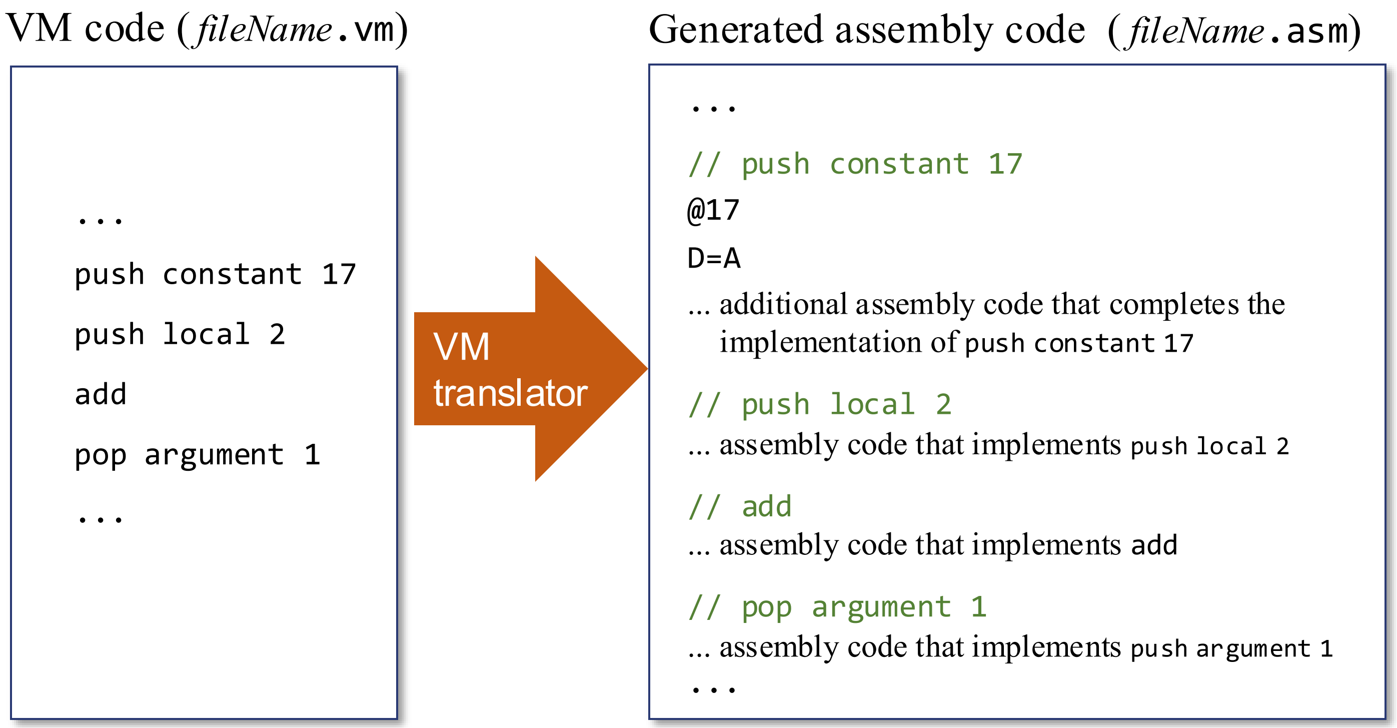
“executable code”

Machine level instructions

(bits)

Executes on a real computer

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, generates assembly code according to the standard mapping, and emits the generated code into the output file.

The VM translator

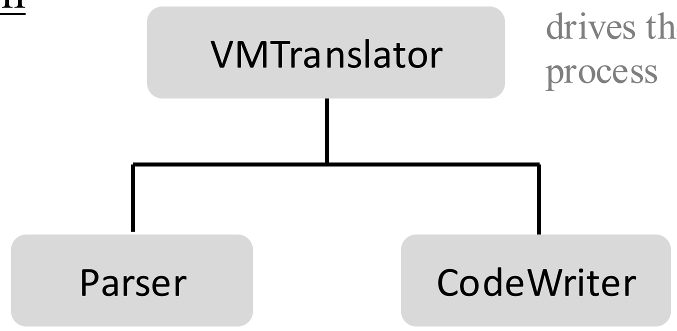
<u>Usage</u>: (if the translator is implemented in Java; Other languages will have a similar command line)

```
$ 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`

<u>Proposed implementation</u>



drives the process

Reads and parses a VM command

Generates the assembly code that realizes the parsed command

The VM translator

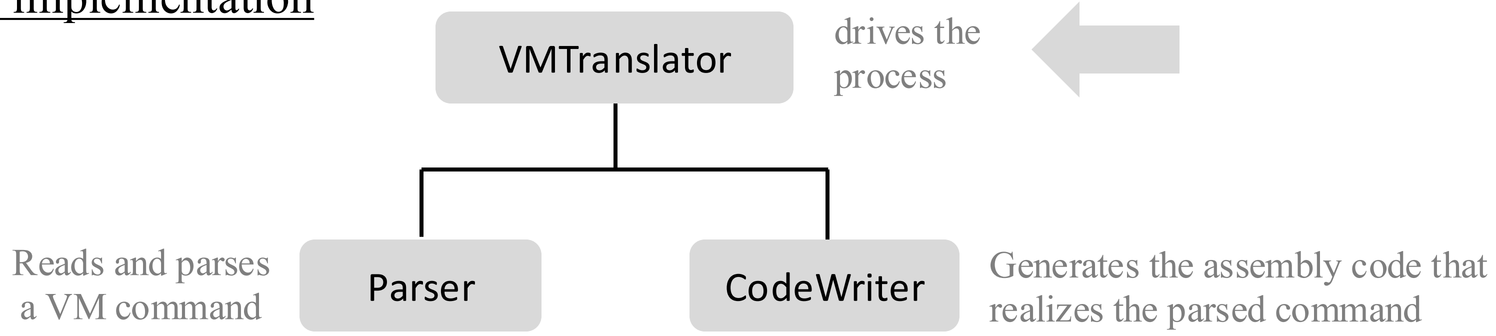
Usage: (if the translator is implemented in Java)

```
$ 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`

Proposed implementation


VMTranslator
drives the process
Parser
Reads and parses a VM command
CodeWriter
Generates the assembly code that realizes the parsed command

VMTranslator

- 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 assembly code from it, using the services of the `Parser` and a `CodeWriter`.

The VM translator

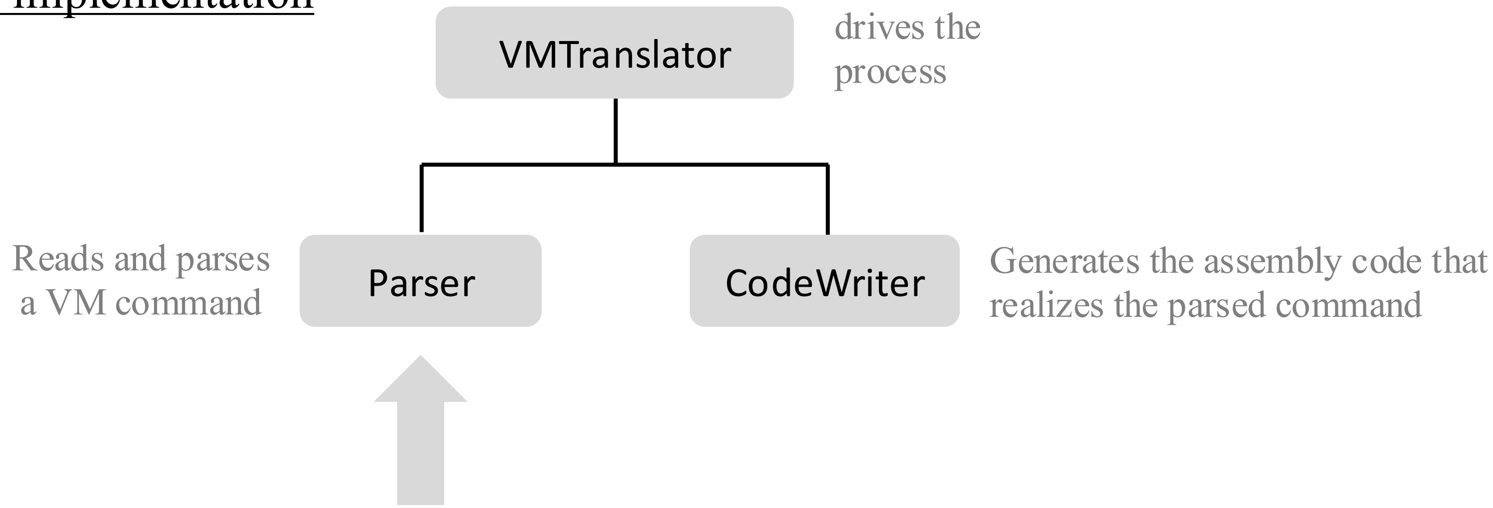
Usage: (if the translator is implemented in Java)

```
$ 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`

Proposed implementation


VMTranslator
drives the process
Parser
Reads and parses a VM command
CodeWriter
Generates the assembly code that realizes the parsed command

The VM commands syntax

Push / pop

`push` *symbol n*

`pop` *symbol n*

Arithmetic / logical

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

Branching

`label` *symbol*

`goto` *symbol*

`if-goto` *symbol*

Function

`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.

- Here we see the syntax of *all* the VM commands, including the *branching* and *function* commands that will be implemented in project 8;
- The basic parser (but not the code generator) that you write in project 7 should handle *all* the VM commands presented here;
- Note that parsing-wise, we don’t care what the commands do; We focus only on their syntax;

Parser API

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 command and makes it the *current instruction* (string)
- Parsing the *current instruction*:

  `commandType()`: Returns the type of the current command (a string constant):

  `C_ARITHMETIC` if the current command is an arithmetic-logical command;

  `C_PUSH`, `C_POP` if the current command is one 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, …`

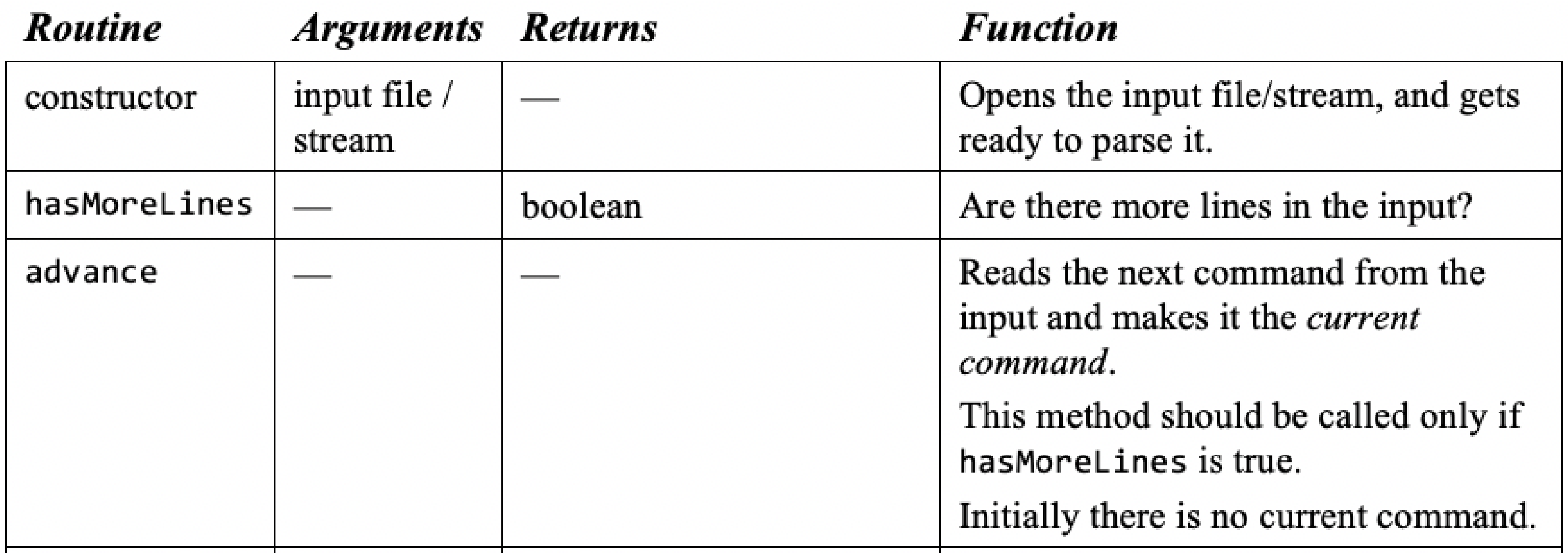
`commandType()` returns `C_ARITHMETIC`;
`arg1()` returns `"add"`, `"neg"`, `"eq"`,…

`push local 3`

`commandType()` returns `C_PUSH`;
`arg1()` returns `"local"`; `arg2()` returns `3`

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

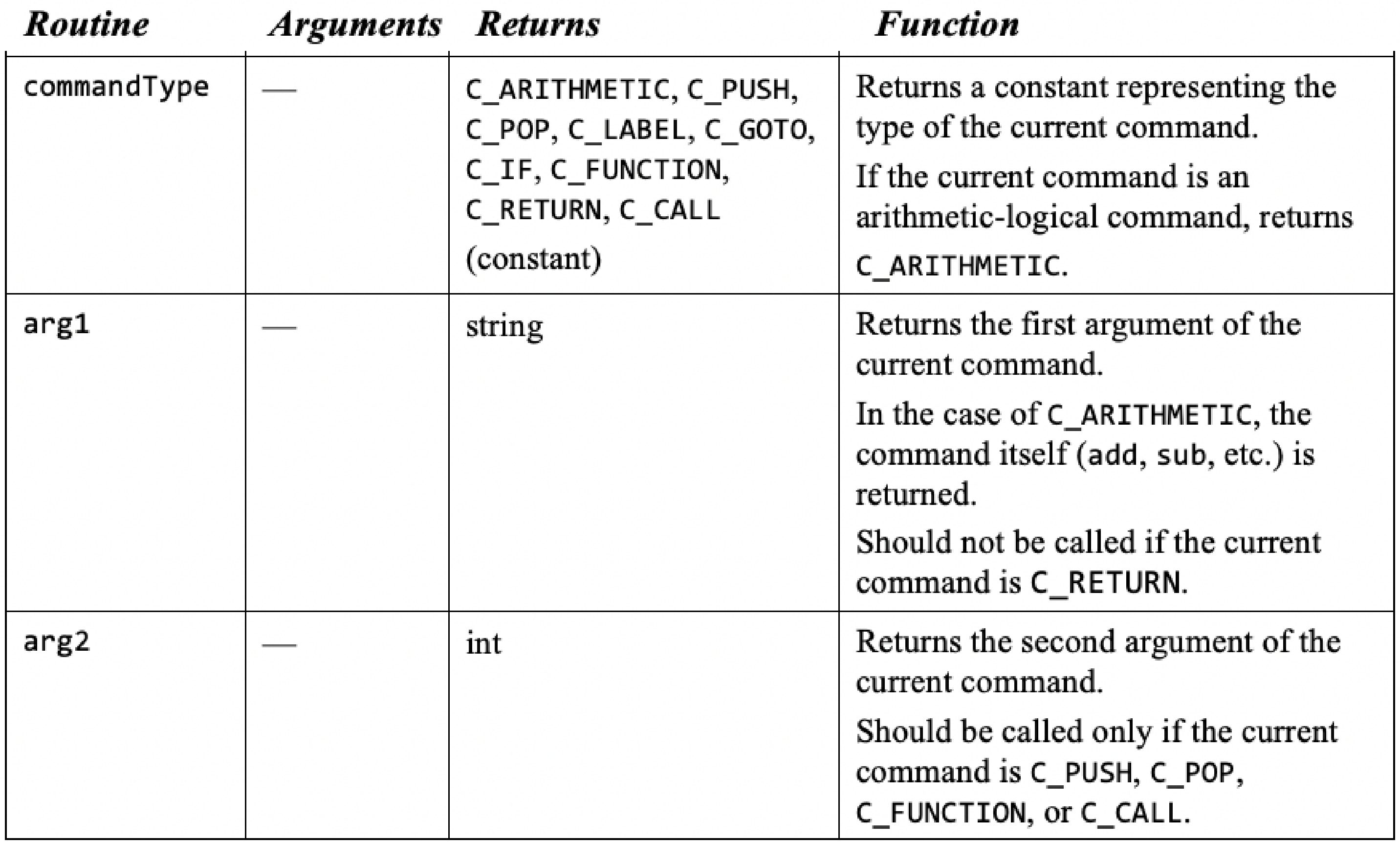


Routine	***Arguments***	***Returns***	***Function***
constructor	input file / stream	—	Opens the input file/stream, and gets ready to parse it.
`hasMoreLines`	—	boolean	Are there more lines in the input?
`advance`	—	—	Reads the next command from the input and makes it the *current command*. This method 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`.

The VM translator

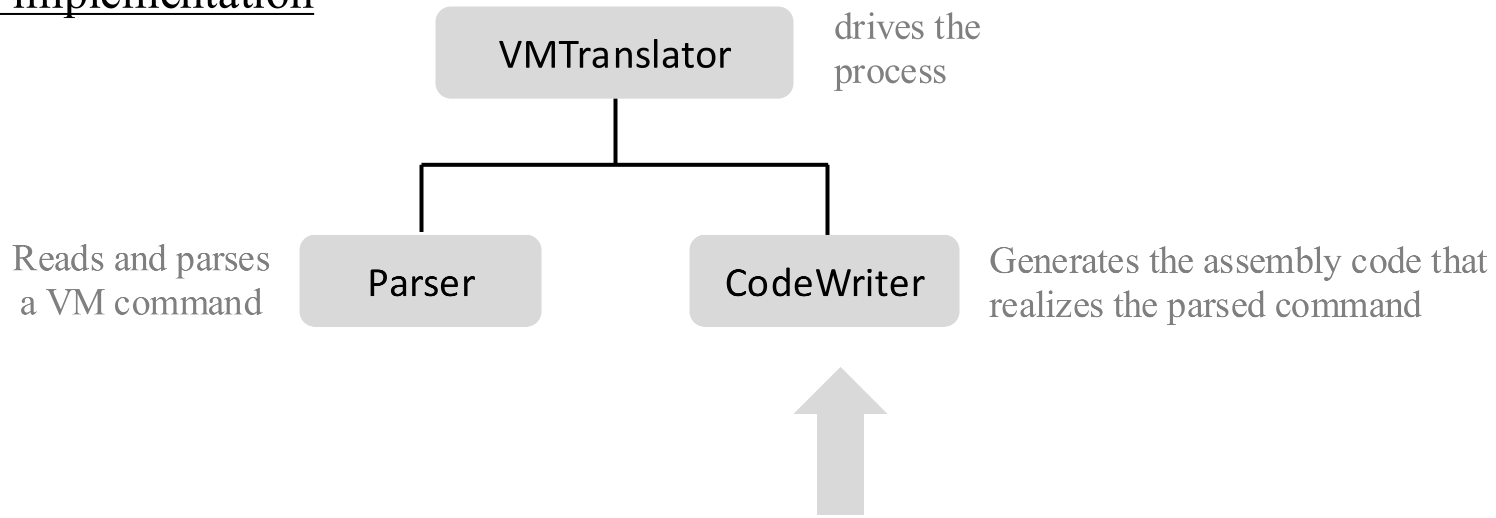
Usage: (if the translator is implemented in Java)

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

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

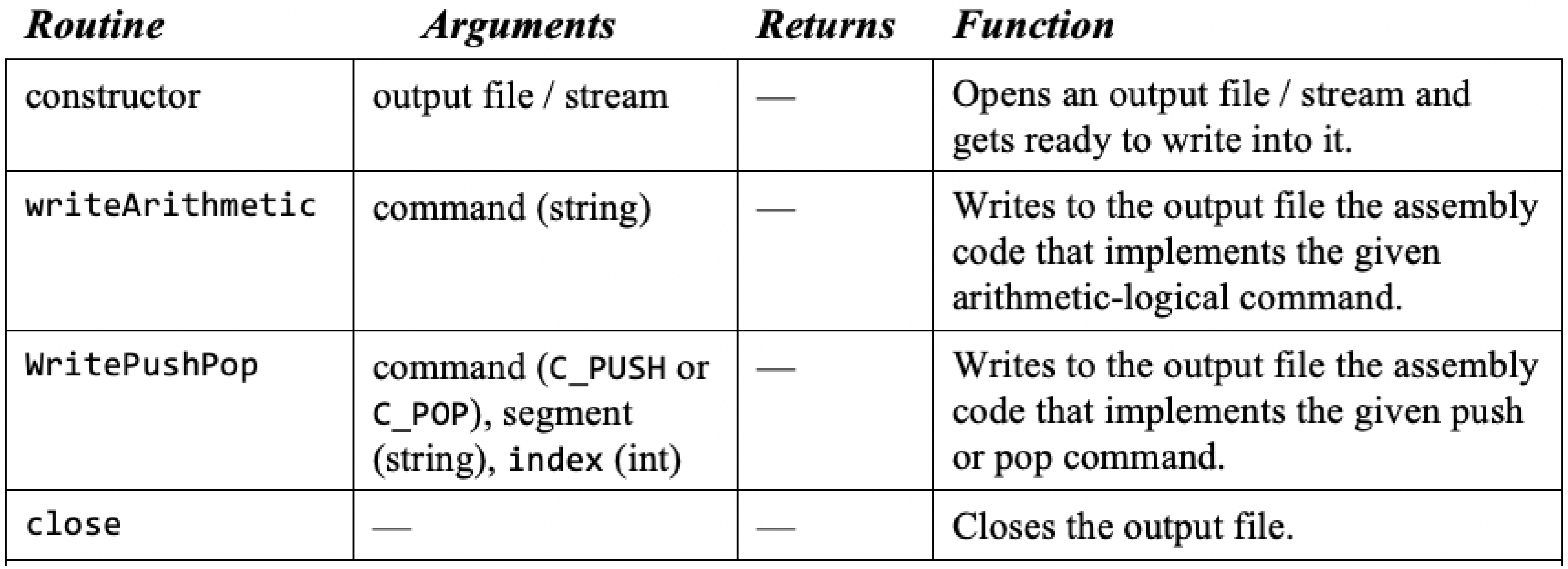
Output: An assembly file named *fileName*.asm

Proposed implementation


VMTranslator
drives the process
Parser
Reads and parses a VM command
CodeWriter
Generates the assembly code that realizes the parsed command

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.

Implementation notes

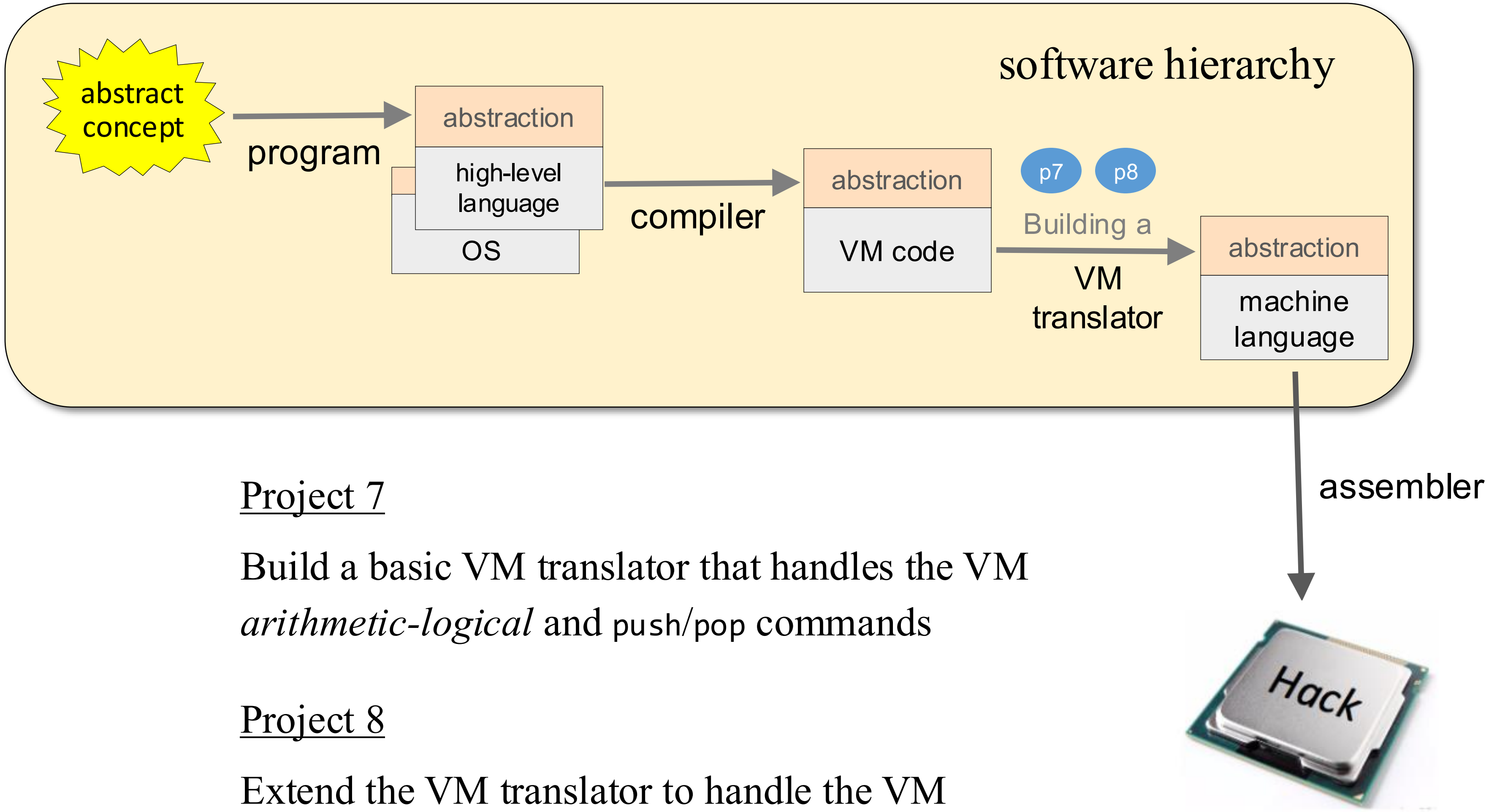
- The components/fields of each VM command are supplied by the Parser routines;
- Implement `true` as `-1` (minus 1) and `false` as `0`;
- Start by writing and debugging *on paper* the assembly code that each VM command must generate; Then have your CodeWriter routines write this code;
- More routines will be added to this module in Project 8, for handling all the commands of the VM language.

Lecture plan

- Overview
- The VM Language
- VM Emulator
- Standard Mapping
- VM Translator

➡ Project 7

Project 7



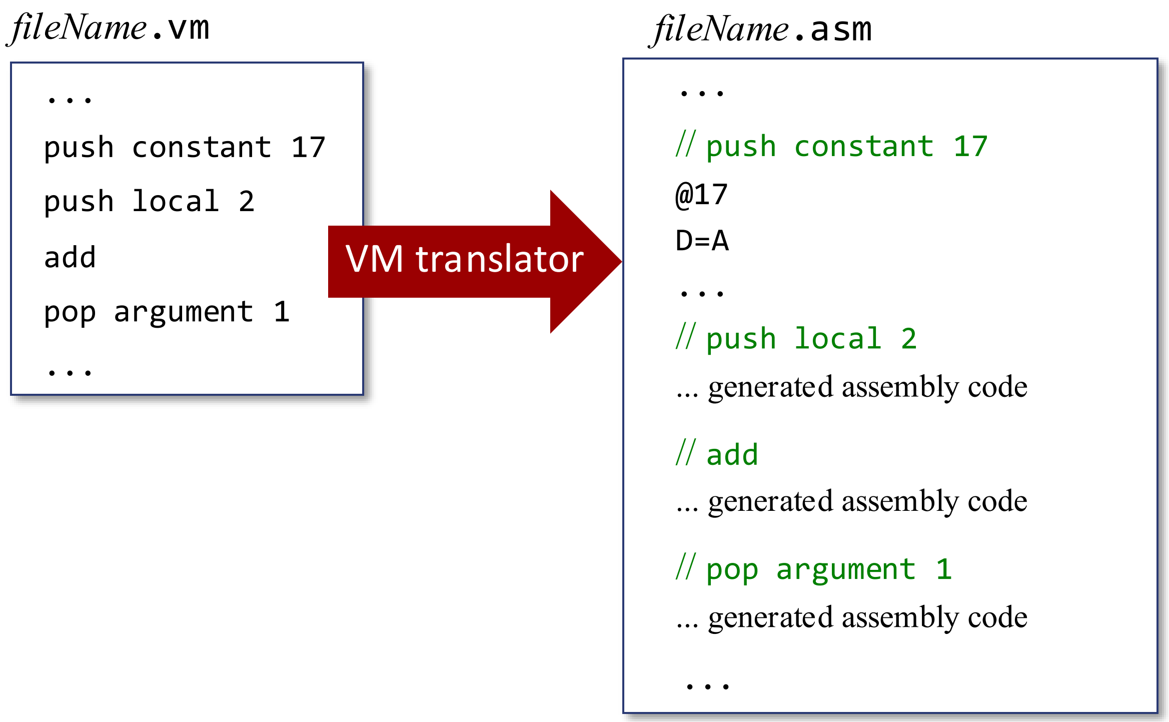
Project 7

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



fileName.vm

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

VM translator

fileName.asm

```
...
// push constant 17
@17
D=A
...
// push local 2
... generated assembly code
// add
... generated assembly code
// pop argument 1
... generated assembly code
...
```

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. (recommended) 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 how the program's operations realize the stack and the segments 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.