Nand to Tetris, Part II

These slides support
the introductory section "Part II" of the book

The Elements of Computing Systems

By Noam Nisan and Shimon Schocken

MIT Press

Hello, World Below

Jack program

```
// First example in Programming 101
class Main {
    function void main() {
        do Output.printString("Hello World!");
        do Output.println(); // New line
        return;
    }
}
```

Issues:

- How does the program execute?
- How does it handle `do`, `while`, ... etc.?
- How does it write on the screen?
- How are classes and methods realized?
- How is function-call-and-return handled?
- How is memory allocated to objects and arrays?

Add your questions to the list…

Q: How can high-level programmers ignore all these issues?

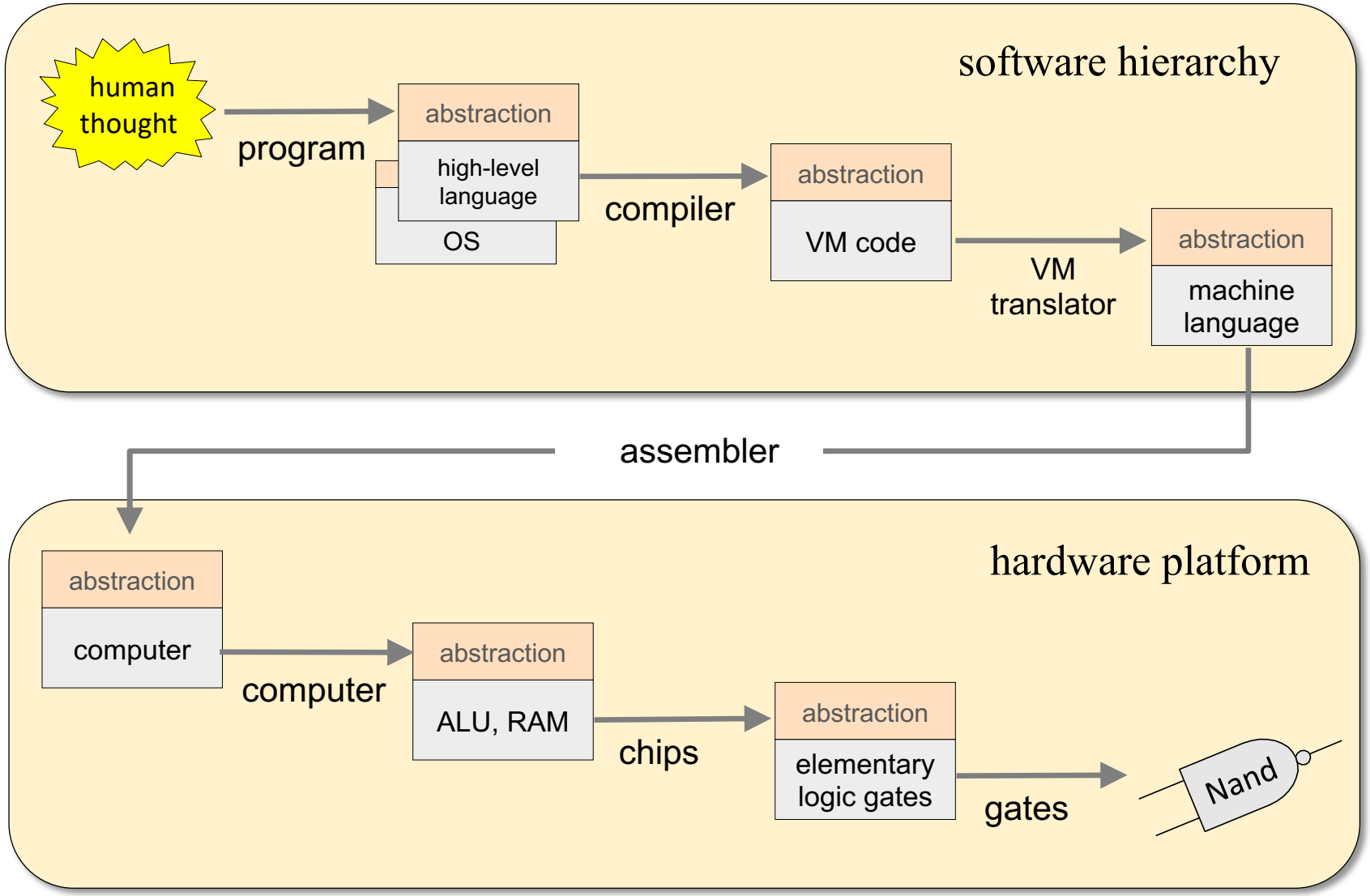
A: They treat the high-level program as an ***abstraction***

What makes the abstraction work?

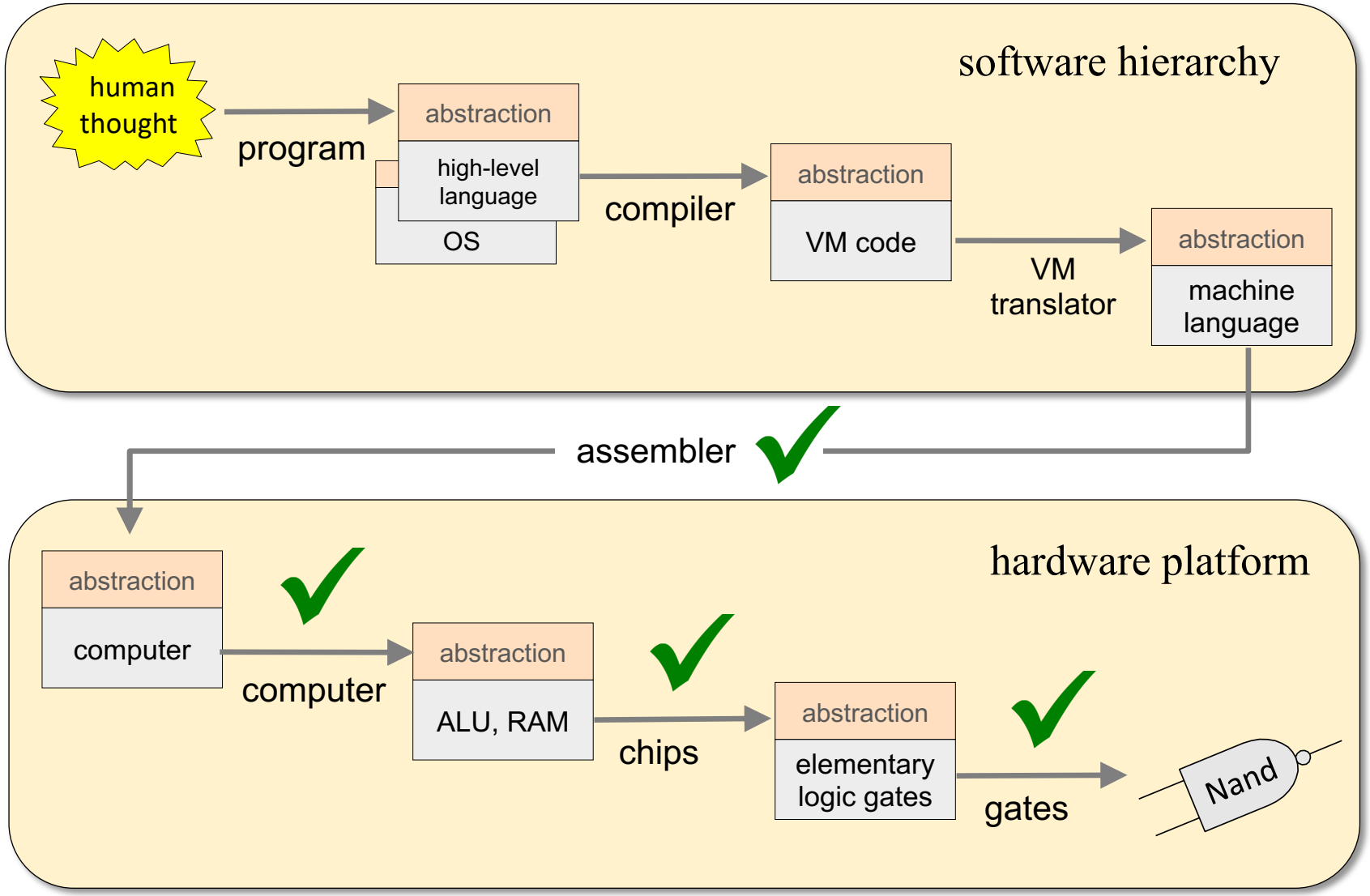
- Assembler
- Virtual machine
- Compiler
- Operating system

} Projects 6-12

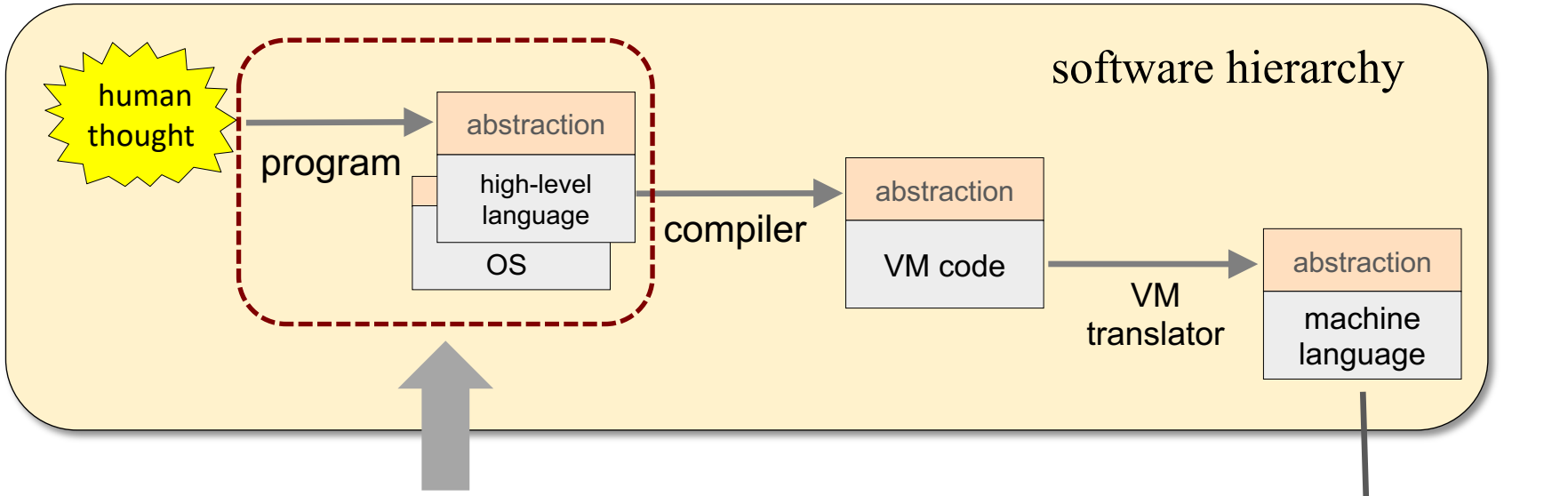
Nand to Tetris Roadmap



Nand to Tetris Roadmap: Part I



Nand to Tetris Roadmap: Part II

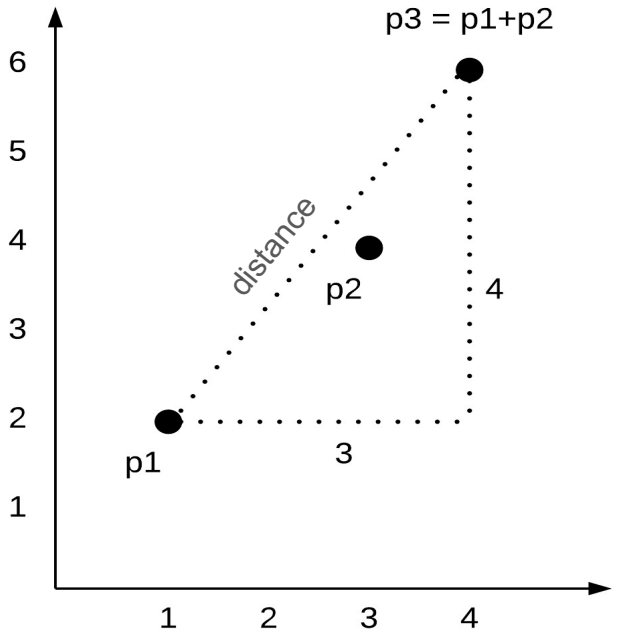


Part II: Building a software infrastructure that enables the compilation and execution of programs written in high-level languages

Our choice: *Jack*

A simple, Java-like, object-based language.

High-level language



Example: write a program that:

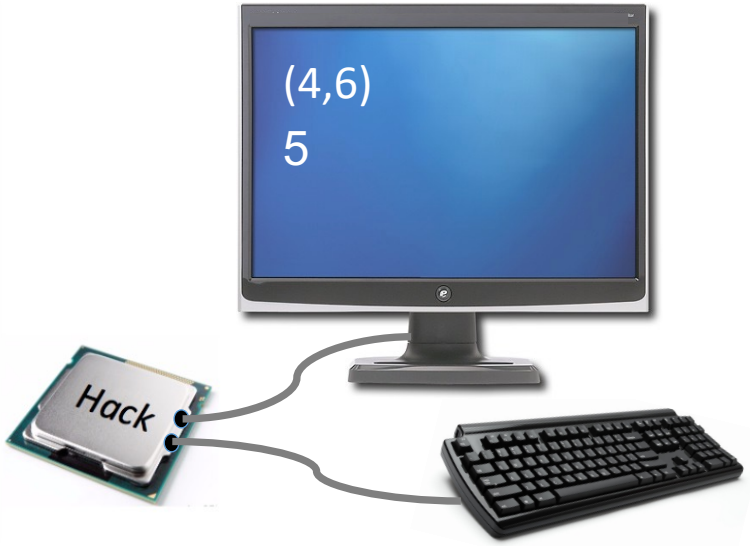
- Constructs 2D points, like `p1` and `p2`
- Computes and prints `p3 = p1 + p2`
- Computes and prints `distance(p1,p3)`

  (Using a `Point` class/ type)

A `Point` class:

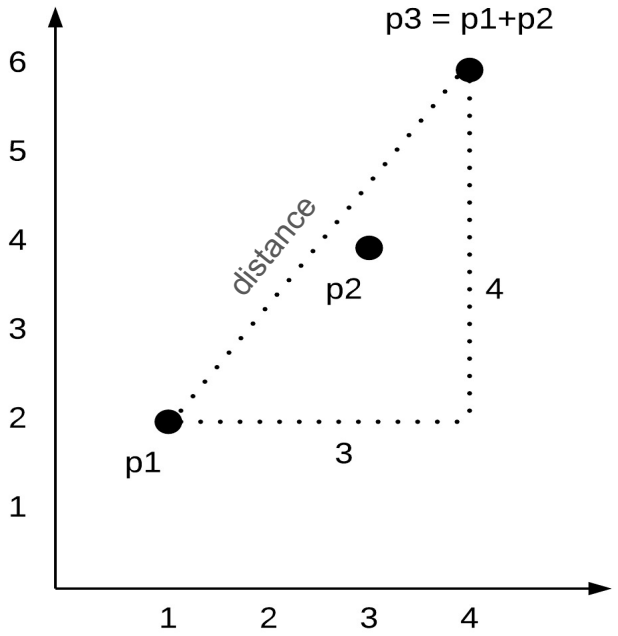
Written in the Jack language

```
/** Represents a 2D point (API) */
class Point {
    /** Constructs a new point with the given coordinates */
    constructor Point new(int ax, int ay)
    /** Returns the point which is this point plus the other point */
    method Point plus(Point other)
    /** Cartesian distance between this and the other point */
    method int distance(Point other)
    /** Prints this point as "(x,y)" */
    method void print()
    ...
}
```



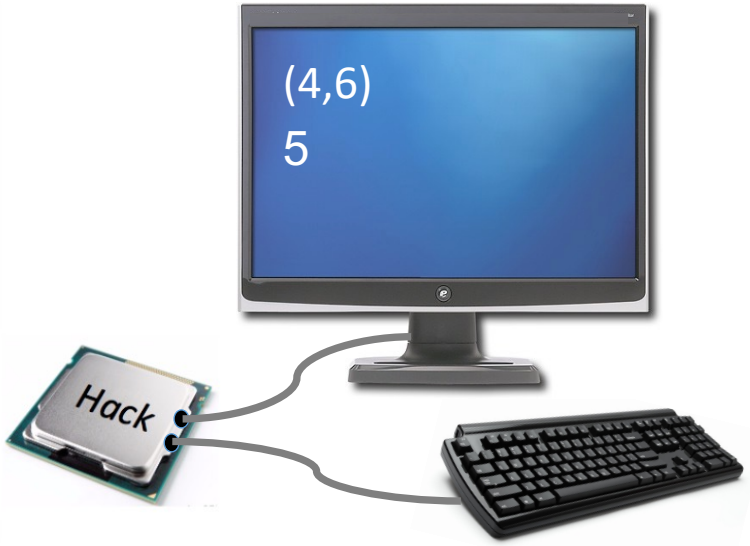
High-level language

```
/** Constructs and manipulates Point objects */
class Main {
  function void main() {
     var Point p1, p2, p3;
     let p1 = Point.new(1,2);
     let p2 = Point.new(3,4);
     let p3 = p1.plus(p2);
     do p3.print();
     do Output.println();
     do Output.printInt(p1.distance(p3));
     return;
  }
}
```



Written in the Jack language

```
/** Represents a 2D point (API) */
class Point {
    /** Constructs a new point with the given coordinates */
    constructor Point new(int ax, int ay)
    /** Returns the point which is this point plus the other point */
    method Point plus(Point other)
    /** Cartesian distance between this and the other point */
    method int distance(Point other)
    /** Prints this point as "(x,y)" */
    method void print()
    ...
}
```



High-level language

```
/** Represents a 2D point.
    File name: Point.jack. */
class Point {
    // The coordinates of this point
    field int x, y
    // The number of Point objects constructed so far:
    static int pointCount;
    /** Constructs a two-dimensional point and
        initializes it with the given coordinates. */
   constructor Point new(int ax, int ay) {
        let x = ax;
        let y = ay;
        let pointCount = pointCount + 1;
        return this;
    }
    /** Returns the x coordinate of this point. */
    method int getx() { return x; }
    /** Returns the y coordinate of this point. */
    method int gety() { return y; }
    /** Returns the number of points constructed so far. */
    function int getPointCount() {
        return pointCount;
    }
    // Class declaration continues on top right.
```

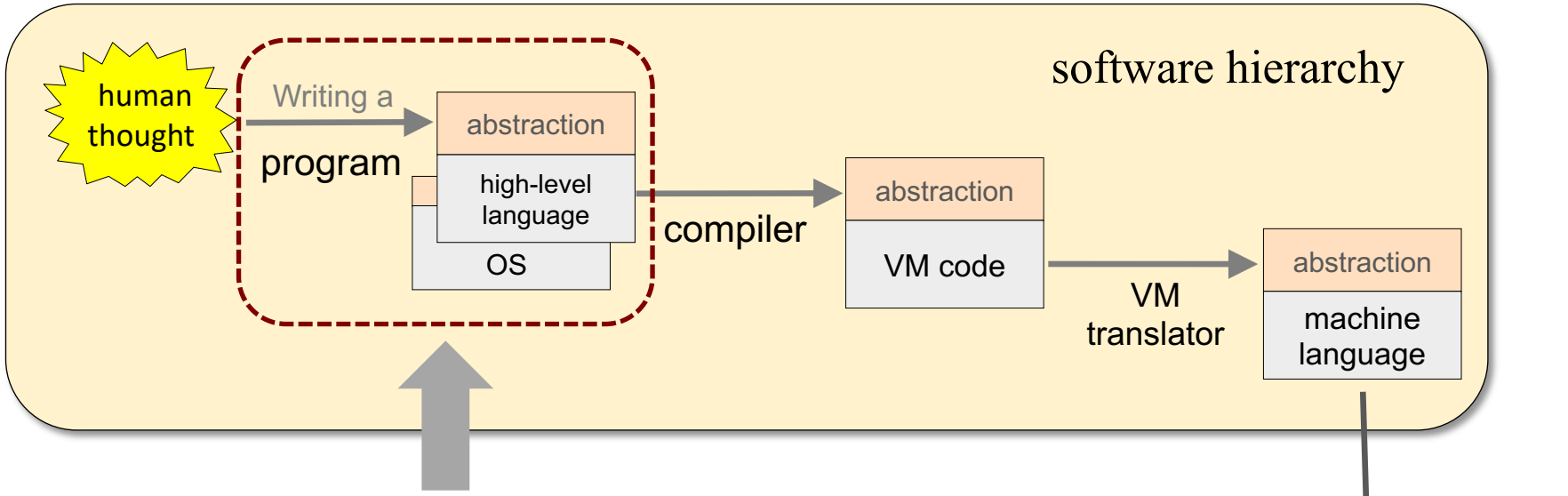
- For now, don't worry about syntax and details
- Just notice that Jack is a typical object-based, high-level language

High-level language

```
/** Represents a 2D point.
    File name: Point.jack. */
class Point {

   // The coordinates of this point
   field int x, y

   // The number of Point objects constructed so far:
   static int pointCount;

   /** Constructs a two-dimensional point and
       initializes it with the given coordinates. */
  constructor Point new(int ax, int ay) {
        let x = ax;
        let y = ay;
        let pointCount = pointCount + 1;
        return this;
   }

   /** Returns the x coordinate of this point. */
   method int getx() { return x; }

   /** Returns the y coordinate of this point. */
   method int gety() { return y; }

   /** Returns the number of points constructed so far. */
   function int getPointCount() {
       return pointCount;
   }

   // Class declaration continues on top right.
```

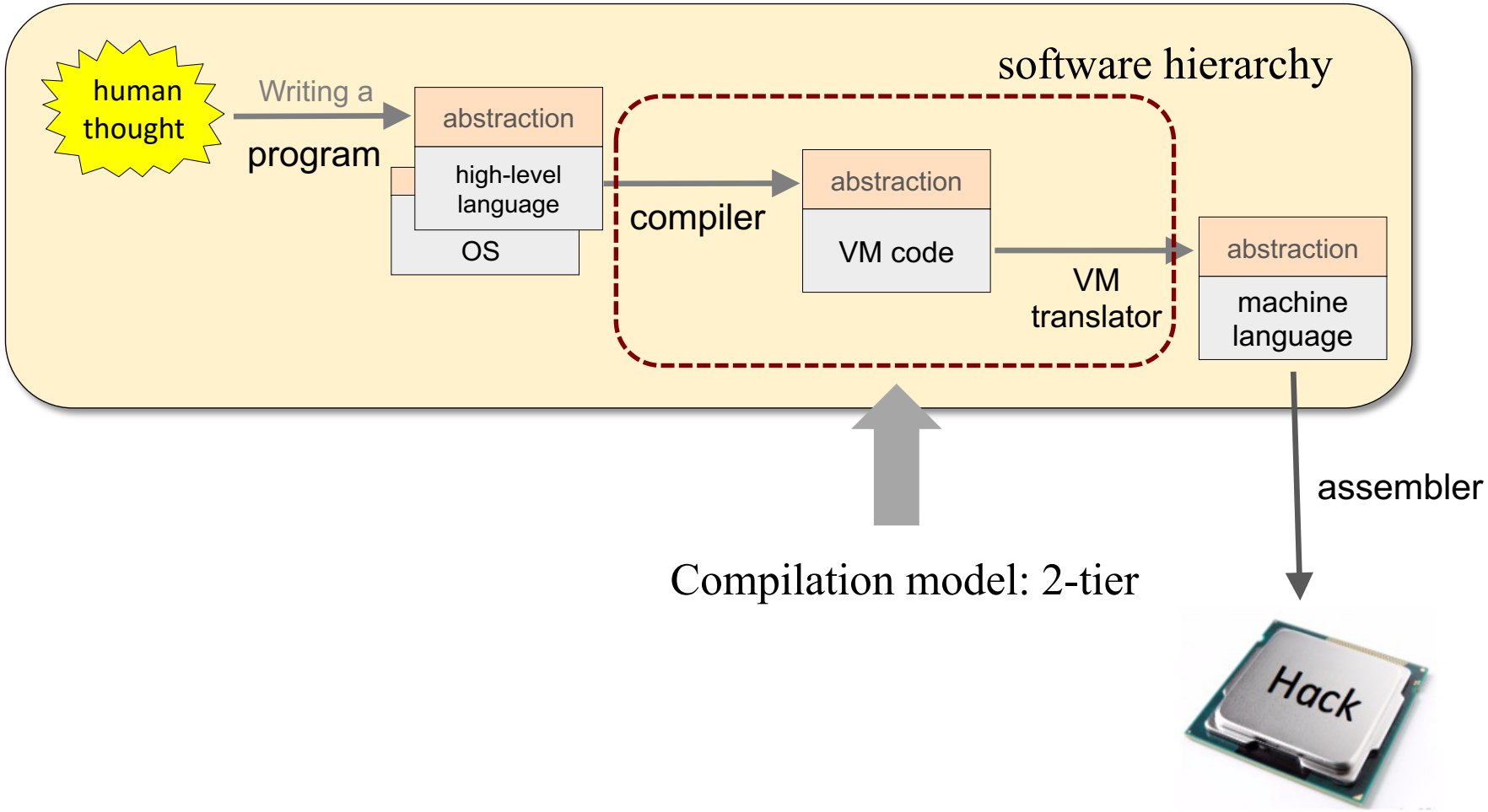
```
   /** Returns a point which is this
       point plus the other point. */
   method Point plus(Point other) {
       return Point.new(x + other.getx(),
                        y + other.gety());
   }

   /** Returns the Euclidean distance between
       this and the other point. */
   method int distance(Point other) {
      var int dx, dy;
      let dx = x – other.getx();
      let dy = y – other.gety();
      return Math.sqrt((dx*dx) + (dy*dy));
   }

   /** Prints this point as "(x,y)" */
   method void print() {
     do Output.printString("(");
     do Output.printInt(x);
     do Output.printString(",");
     do Output.printInt(y);
     do Output.printString(")");
     return;
   }

} // End of Point class declaration.
```

Nand to Tetris Roadmap: Part II



What does it take to implement such a language?

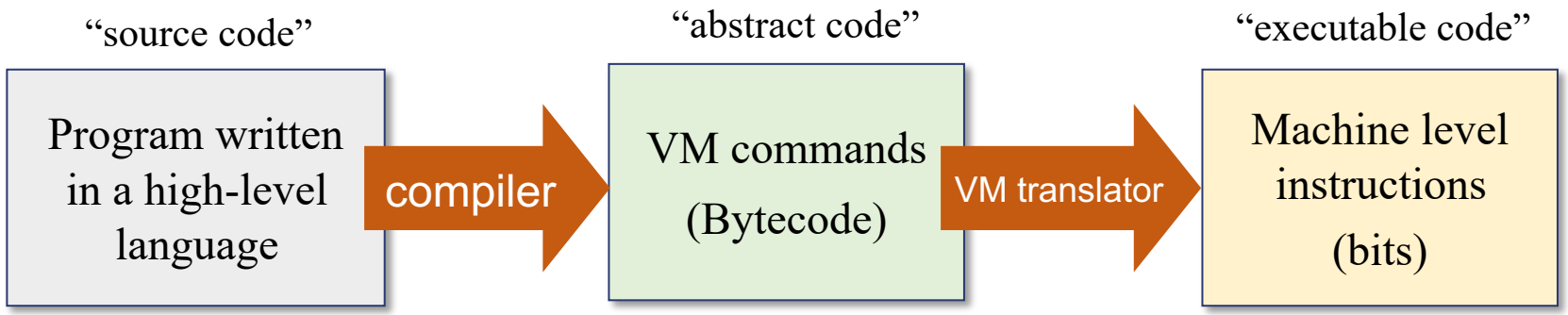
Nand to Tetris Roadmap: Part II



Compilation

One tier:

Two tier:



Compilation

One tier:

Pros:

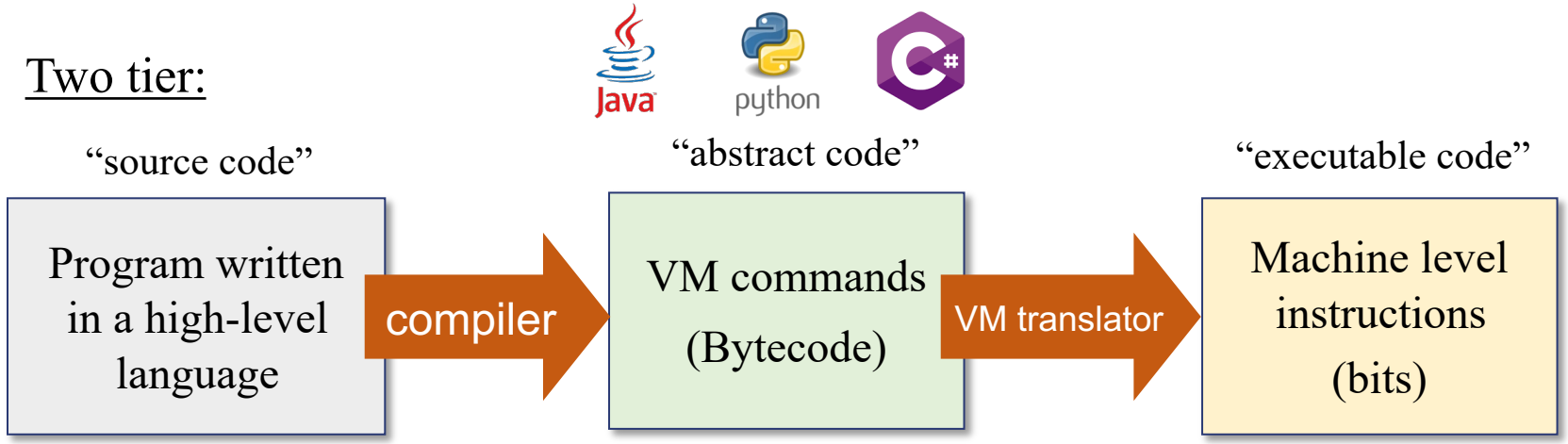
1-tier compilers generate efficient, target-specific code

Cons:

Requires multiple compilers / compilations, one for each target hardware platform.

Compilation

Two tier:



Pros:

Compile once, run everywhere

Cons:

- Requires multiple VM translators, one for each target hardware platform
  (But: VM translators are much simpler than compilers)
- Generates less efficient machine language code
  (But: For numerous apps, the inefficiency delta is insignificant).

Compilation

Program folder
(named points)

Main.jack

```
class Main
  function void main {
     var Point p1;
     ...
  }
}
```

Point.jack

```
class Point
  field int x,y;
  ...
}
```

compiler

Program folder
(named points)

Main.vm

```
...
push local 1
push const 3
add
...
```

Point.vm

```
...
push arg 2
call Math.sqrt
...
```

VM translator

points.asm

```
...
M=D
@i
M=0
(LOOP)
@i
D=M
@n
D=D-M
@END
D;JGT
@addr
...
```

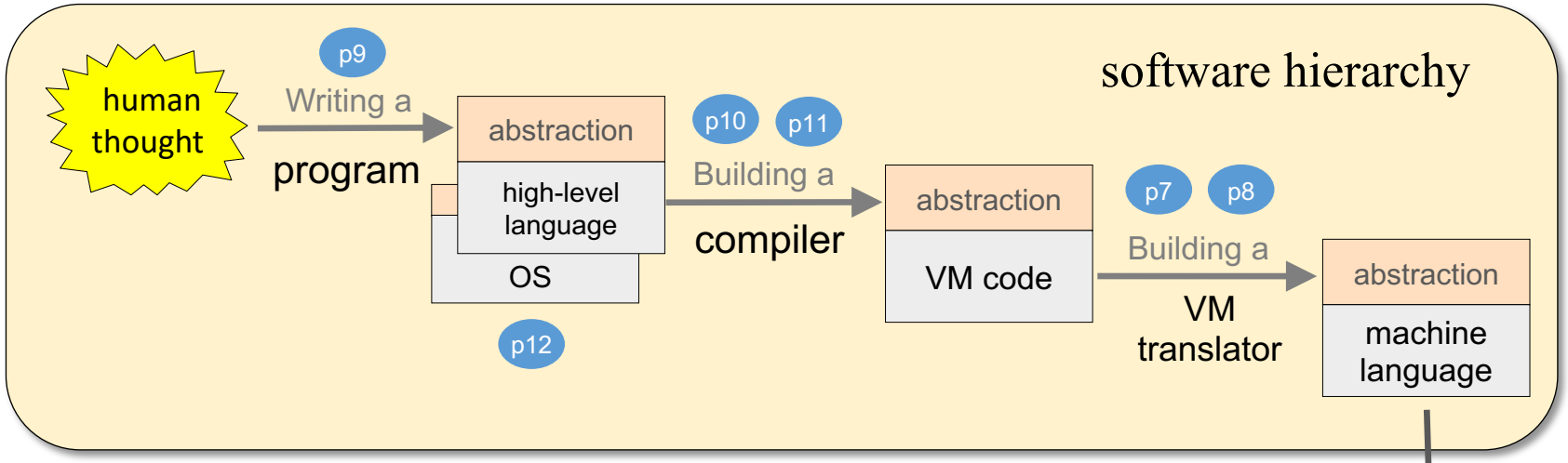
assembler

points.hack

```
...
1101010010101011
1100001101010101
0101010011101100
1100101111111111
0101110101010011
1100111111010101
0101110110101011
1100100000001011
0101110111100010
1111000111010100
0010101000101011
0111000111010111
1010101000101011
...
```

load

Nand to Tetris Roadmap: Part II



Part II chapters / projects:

7, 8: building the compiler's backend (virtual machine)

9: writing a computer game

10, 11: developing the compiler's frontend

12: developing an operating system.

Chapter 7

Virtual Machine

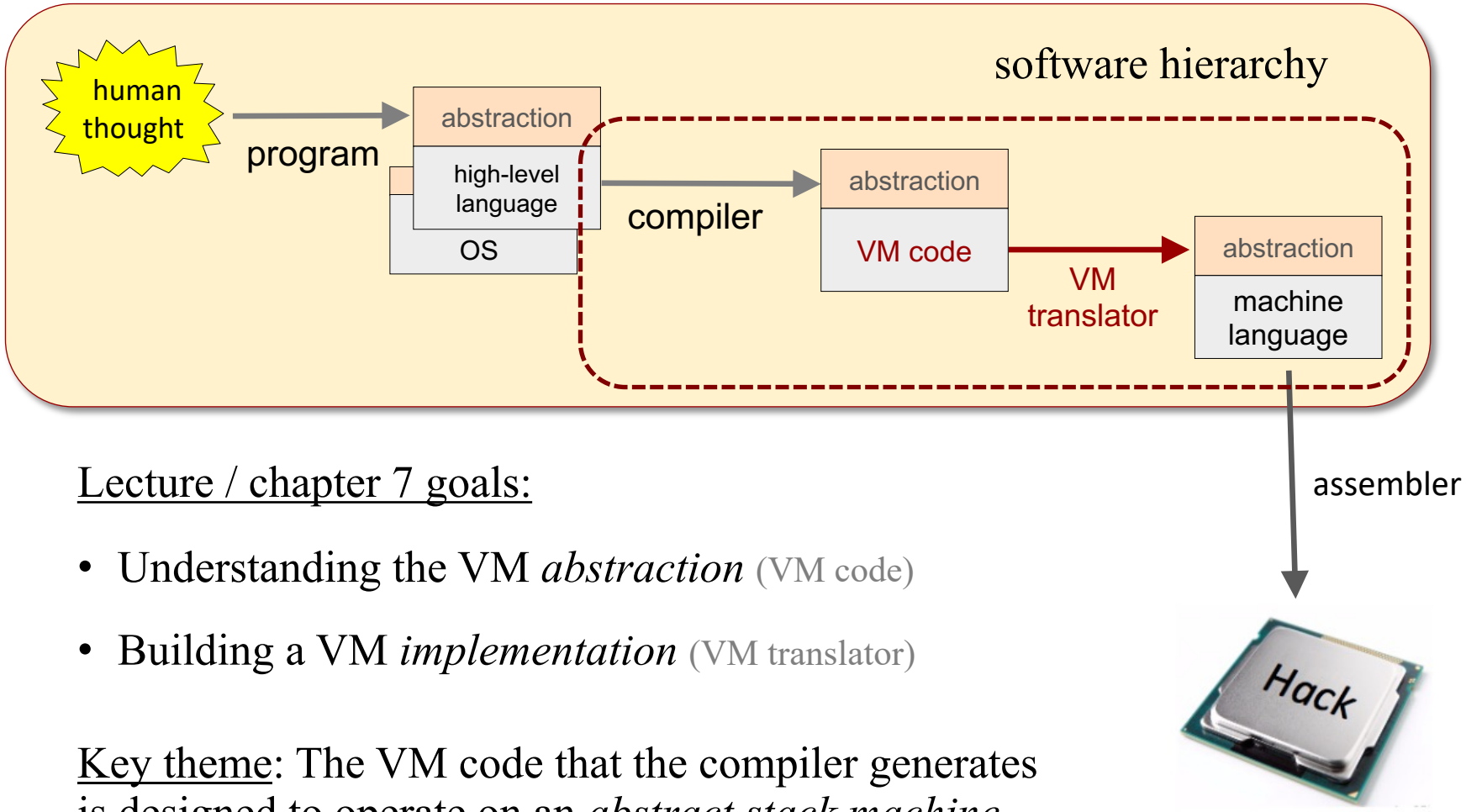
(Part I)

These slides support chapter 7 of the book
The Elements of Computing Systems
By Noam Nisan and Shimon Schocken
MIT Press

Virtual Machine I: Lecture plan

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

Nand to Tetris Roadmap: Part II



Lecture / chapter 7 goals:

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

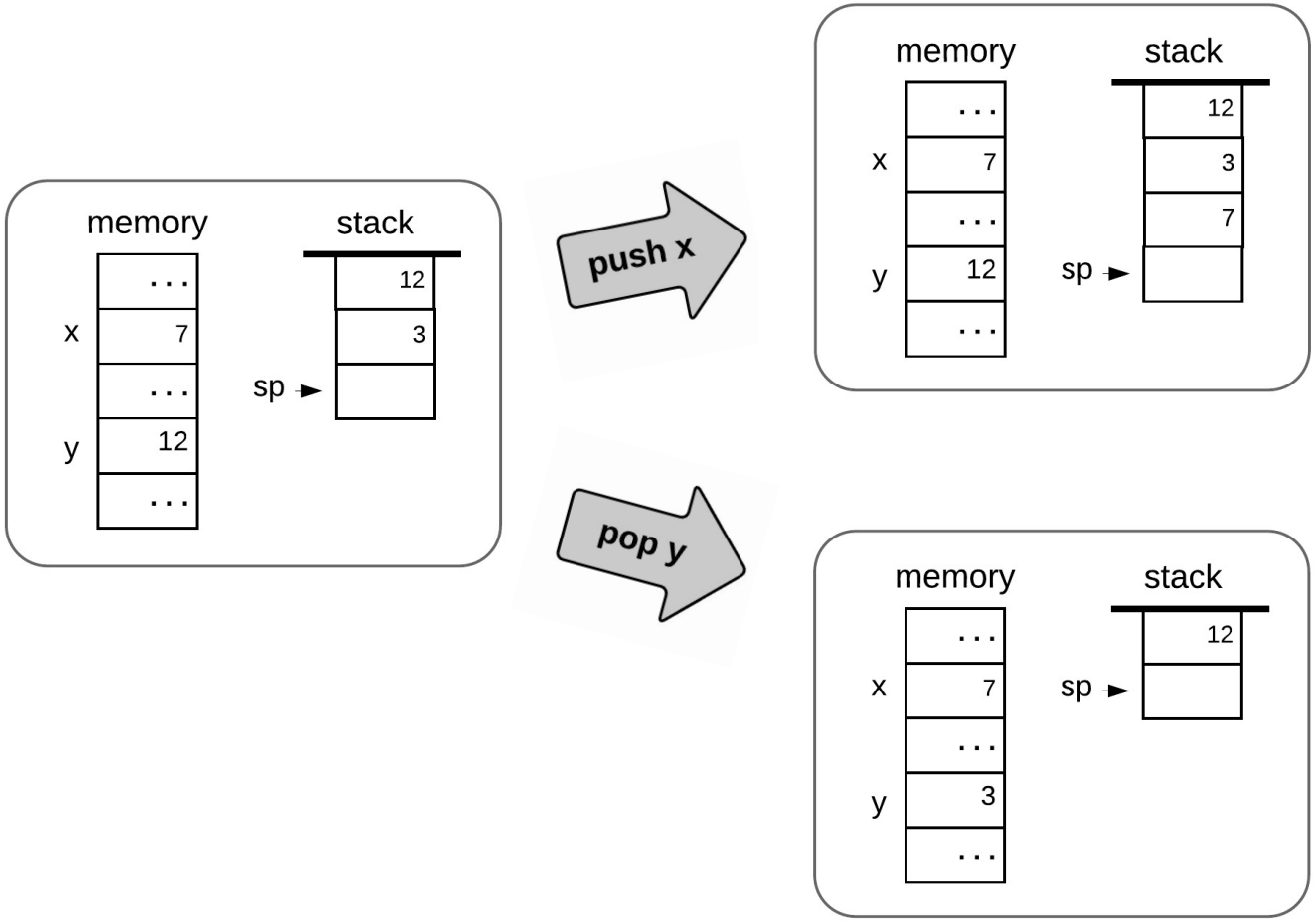
Key theme: The VM code that the compiler generates is designed to operate on an *abstract stack machine.*

Stack

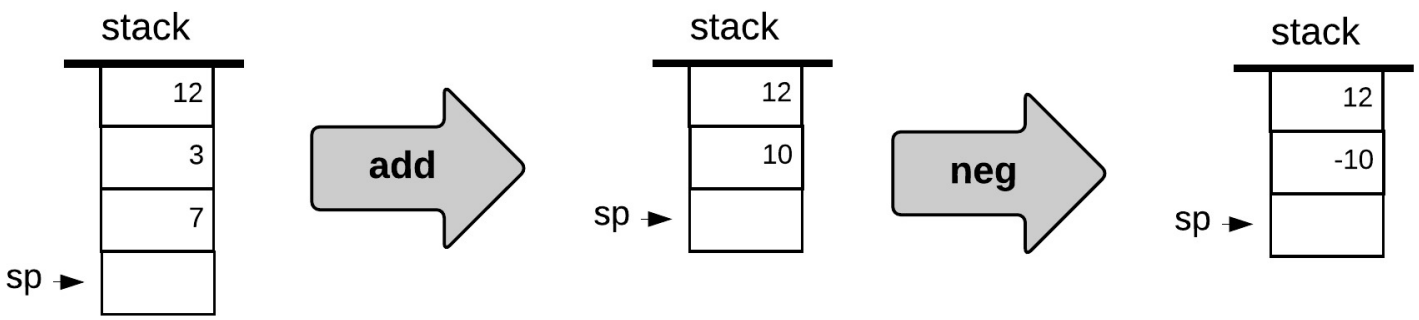
Basic operations:

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

Stack


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

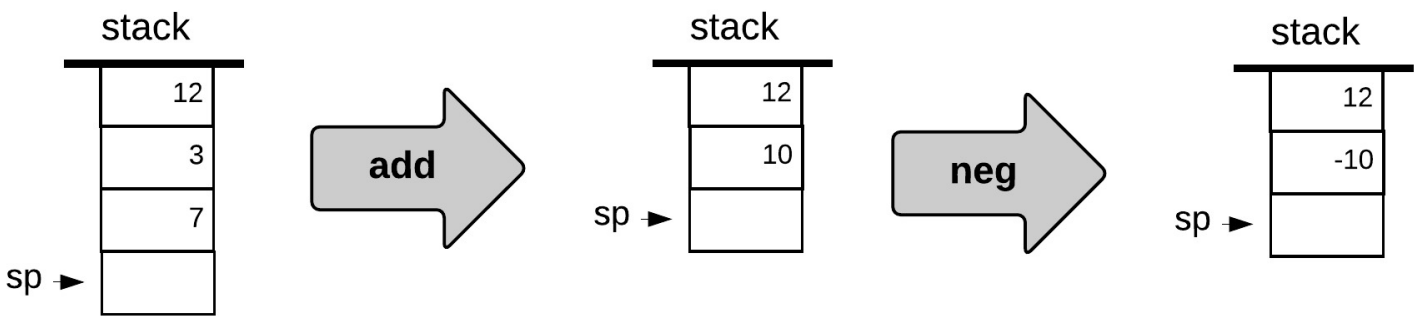
Stack arithmetic



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

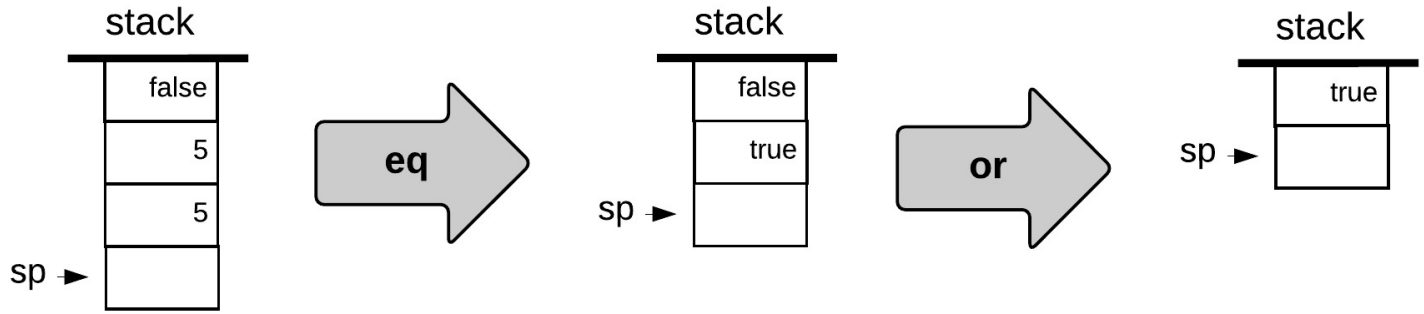
- pops *n* values (arguments) from the stack,
- Computes *f* on the values,
- Pushes the resulting value onto the stack.

Stack arithmetic



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

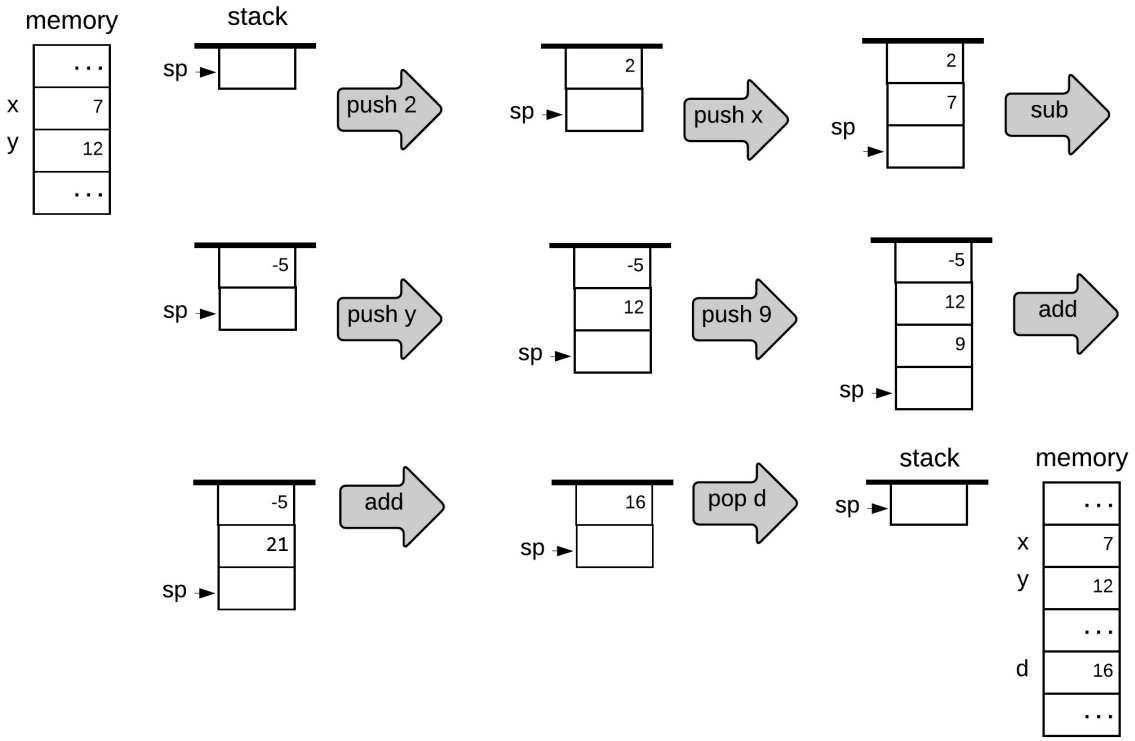
- pops *n* values (arguments) from the stack,
- Computes *f* on the values,
- Pushes the resulting value onto the stack.



Arithmetic commands

VM code (example)

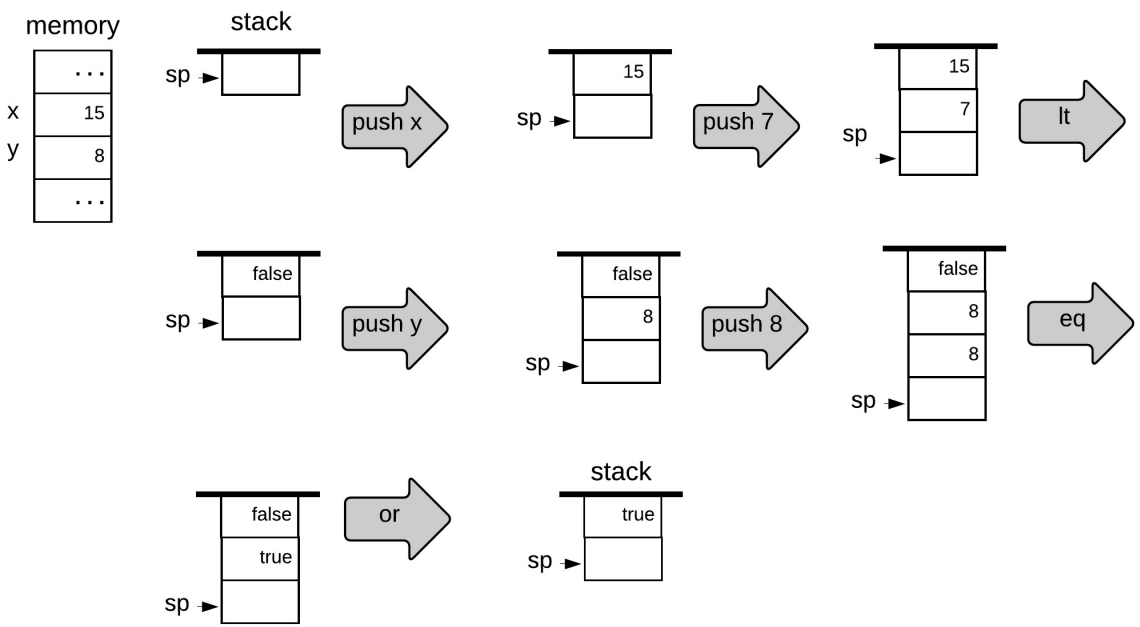
```
// d = (2 - x) + (y + 9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Logical commands

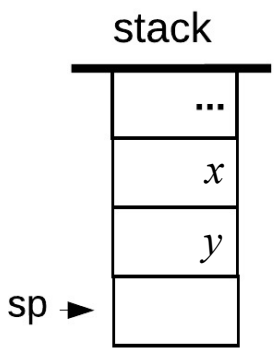
VM code (example)

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



Arithmetic / Logical VM 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



<u>The big picture: Compilation</u>

- 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*.

Virtual Machine I: Lecture plan

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

The VM language (*abstraction*)

<u>Push / pop commands</u>

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

<u>Arithmetic / Logical commands</u>

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

This lecture

<u>Branching commands</u>

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

<u>Function commands</u>

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

Next lecture

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 s1
push y
add
pop c
...
```

- Two-tier compilation: The compiler translates each high-level statement into a sequence of VM commands
- Arithmetic-logical operators like `+`, `-`, `<`, `==`, ... are translated into the VM arithmetic-logical commands `add`, `sub`, `lt`, `eq`, ...
- References to high-level variables are translated into `push` / `pop` commands.

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;
      ...
   }
}
```

Each variable has a *kind*:

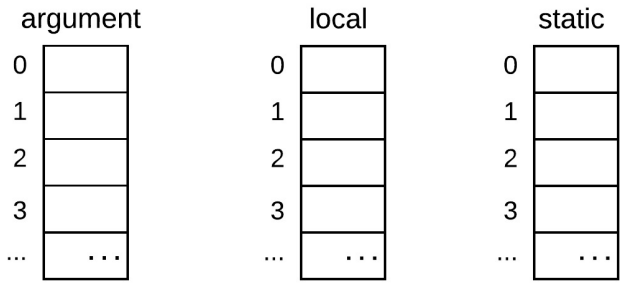
`local`, `static`, `argument`, `field`

compiler

Compiled VM code

```
...
...
...
...
push s1
push y
add
pop c
...
```

Abstract memory segments



- The variable kinds must be represented also at the VM level
- This is done using VM-level *abstract memory segments*

The big picture: Compilation

Source code (Jack)

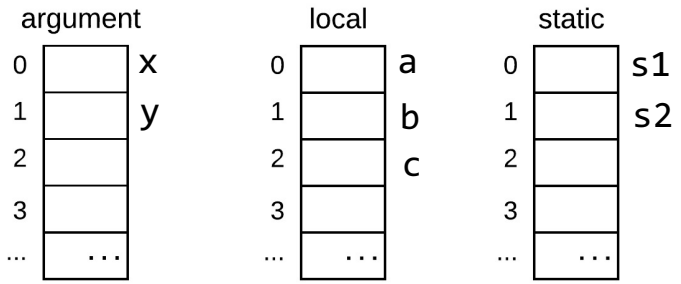
```
class Foo {
   static int s1, s2;
   function int bar (int x, int y) {
      var int a, b, c;
      ...
      let c = s1 + y;
      ...
   }
}
```

compiler

Compiled VM code

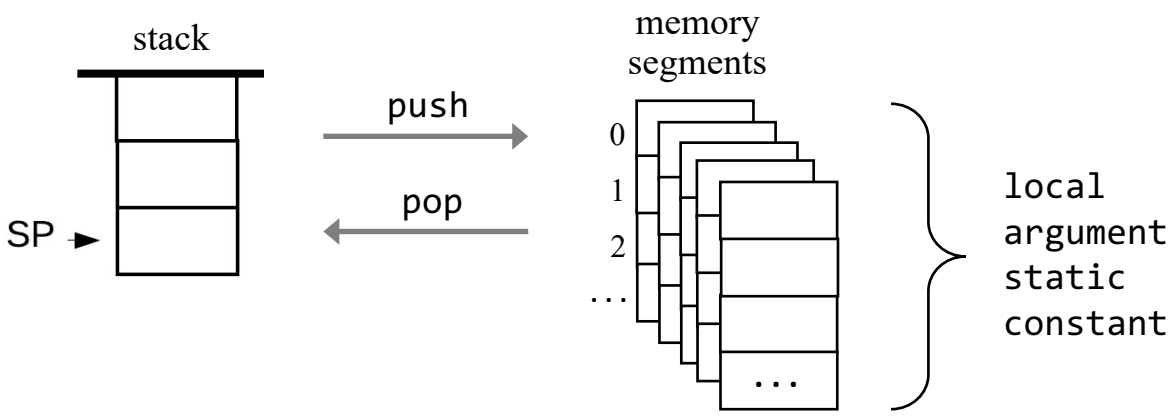
```
...
...
...
...
push static 0
push argument 1
add
pop local 2
...
```

Abstract memory segments



- The compiler maps high-level symbolic variable names on *abstract memory segment entries*
- The compiler translates each reference to a symbolic variable in the high-level code into a VM command of the form `push` / `pop` *segment index*

Memory segments

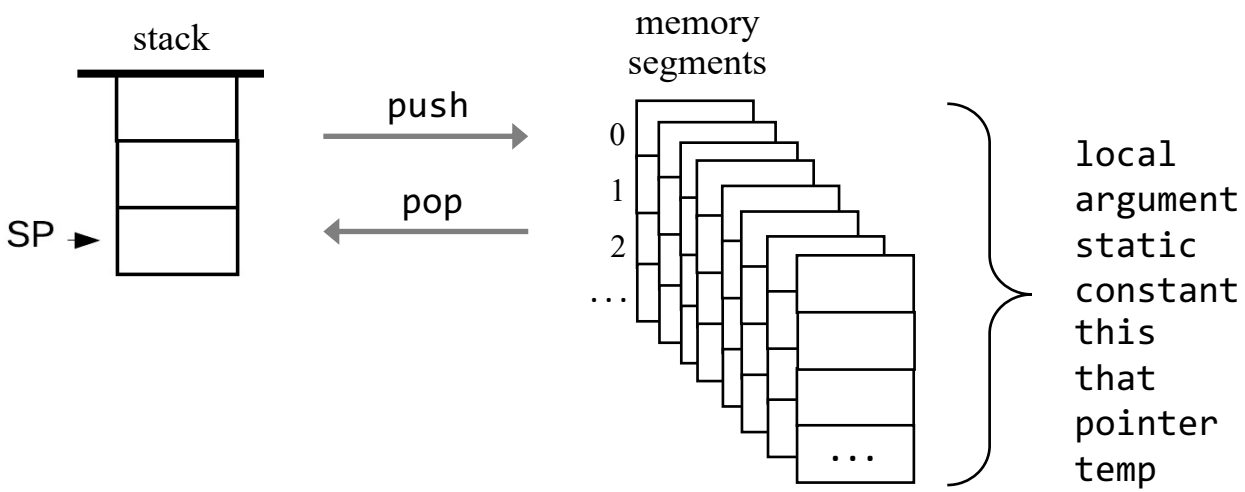


Example:

Suppose you want to set `static 2` to `argument 3 + 17`

```
// static 2 ← argument 3 + 17
push argument 3
push constant 17
add
pop static 2
```

Memory segments



- Altogether, our VM model features 8 memory segments, whose roles will be described in the next lecture
- All segments are accessed using the same syntax:

`push` / `pop` *segment i*

where *segment* is `local`, `argument`, `static`, `constant`, `this`, `that`, `pointer`, or `temp`, and *i* is a non-negative integer.

The VM language: Abstraction

✓ 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

Virtual Machine I: Lecture plan

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

The VM language: Implementation

Push / pop commands

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

Branching commands

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

Arithmetic / Logical commands

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

Function commands

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

How can we make this language "work"?

One option: Develop a *VM translator*

VM translator: A program that translates VM commands into machine language instructions

Next task: Think how to implement each VM command in assembly

The VM language: Implementation

Push / pop commands

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

Branching commands

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

Arithmetic / Logical commands

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

Function commands

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

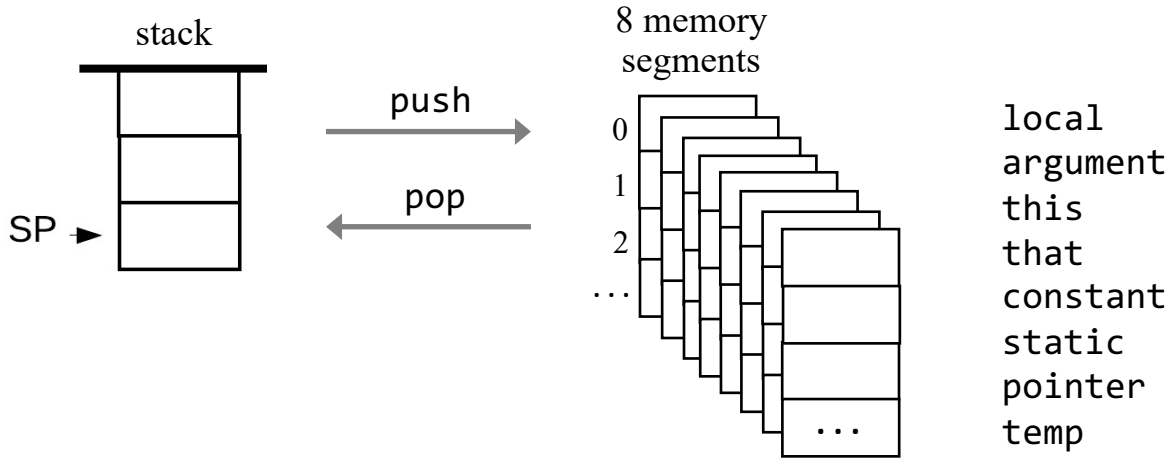
How can we make this language "work"?

One option: Develop a *VM translator*

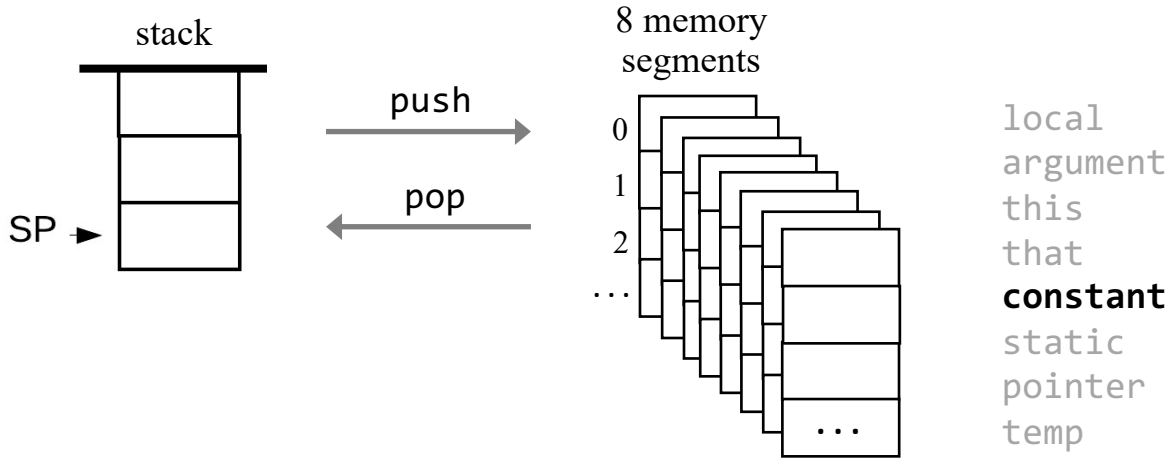
VM translator: A program that translates VM commands into machine language instructions

Next task: Think how to implement each VM command in assembly

Implementing `push` / `pop` commands



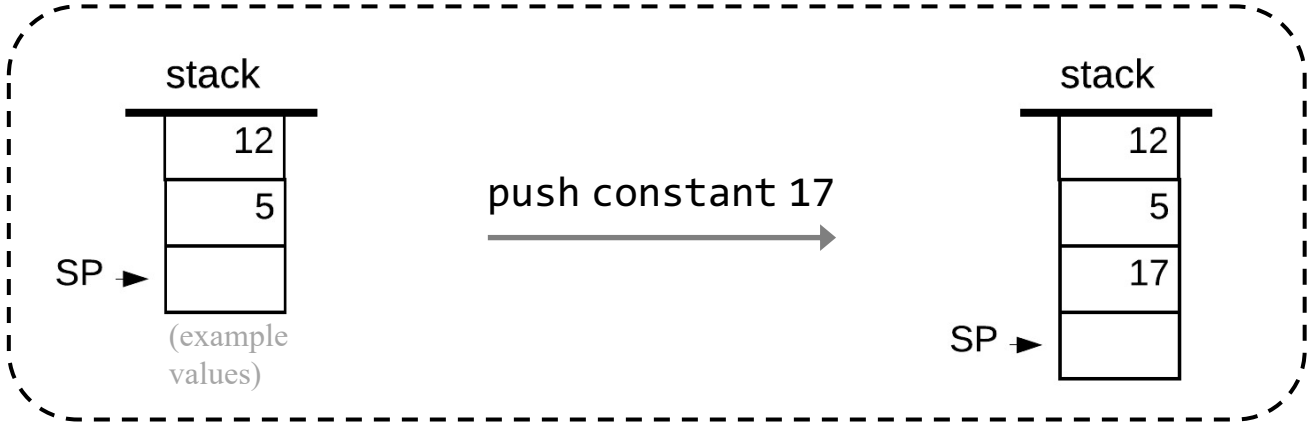
Implementing `push` / `pop` commands




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

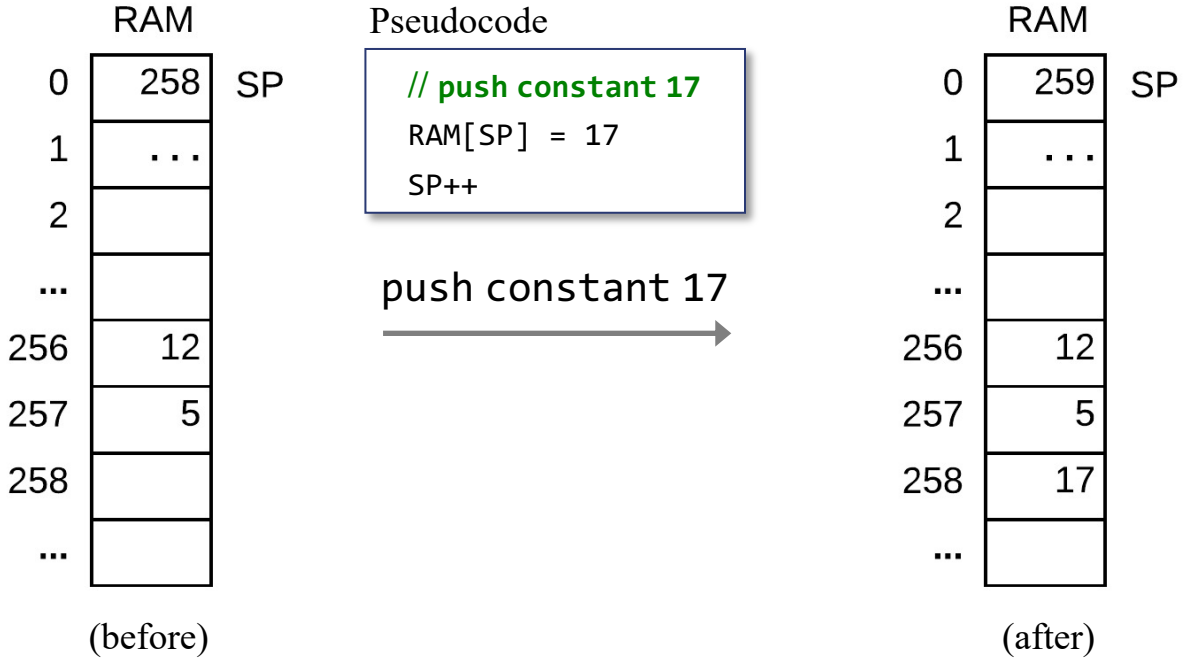
Implementing `push constant i`

Abstraction:



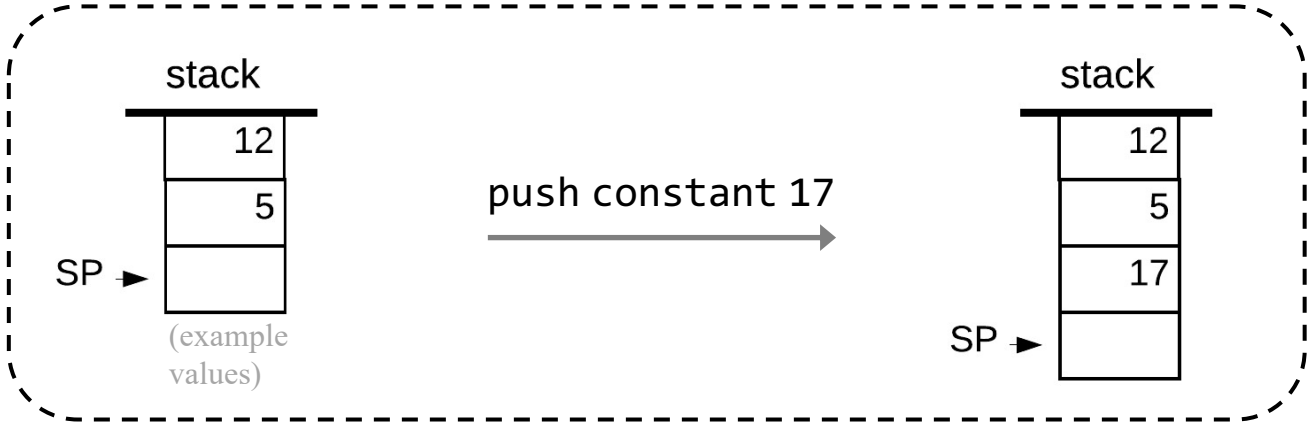
Implementation:

- The stack is stored in the RAM, from address 256 onward
- The stack pointer is stored in `RAM[0]`



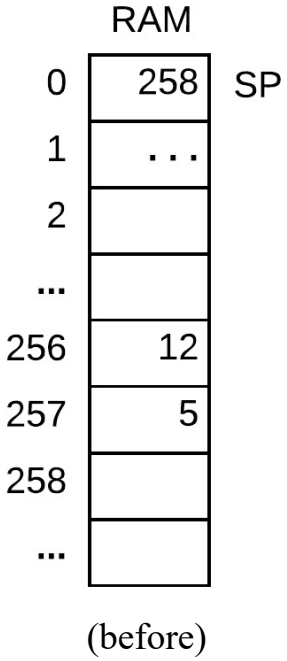
Implementing `push constant i`

Abstraction:



Implementation:

- The stack is stored in the RAM, from address 256 onward
- The stack pointer is stored in `RAM[0]`

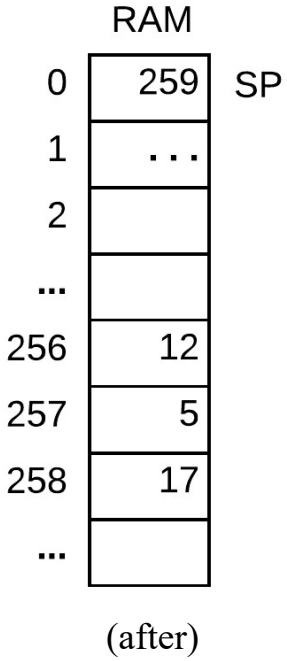


Pseudocode

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

Hack assembly

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



Implementing `push constant` *i*

Abstraction

VM code

```
push constant i
```

VM translator

Implementation (general solution)

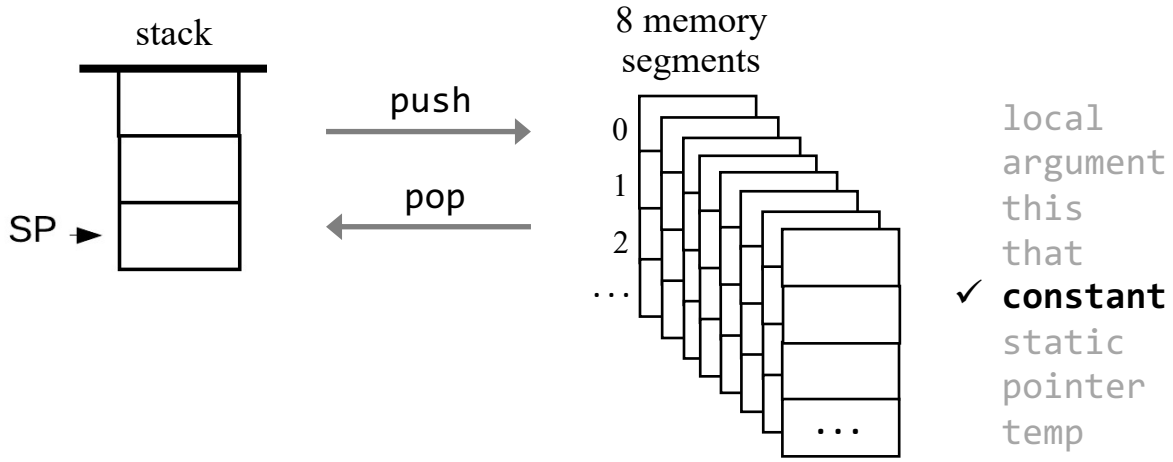
Assembly code

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

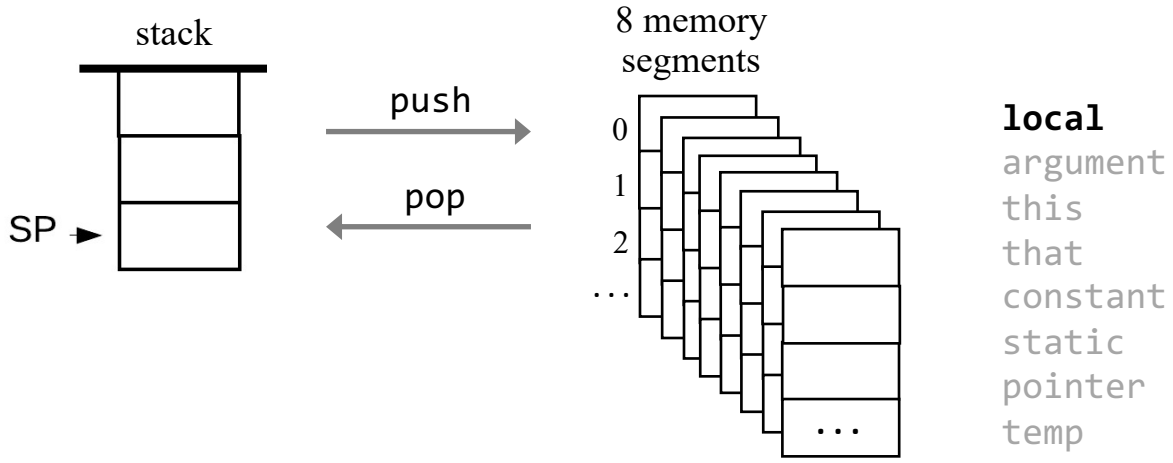
Notes

- `constant` is not a "real segment"
- Used for VM syntax consistency
- There is no `pop constant` *i* command.

Implementing `push` / `pop` commands



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



<u>Abstraction:</u> `push/pop local` ***i***

<u>Implementation:</u>

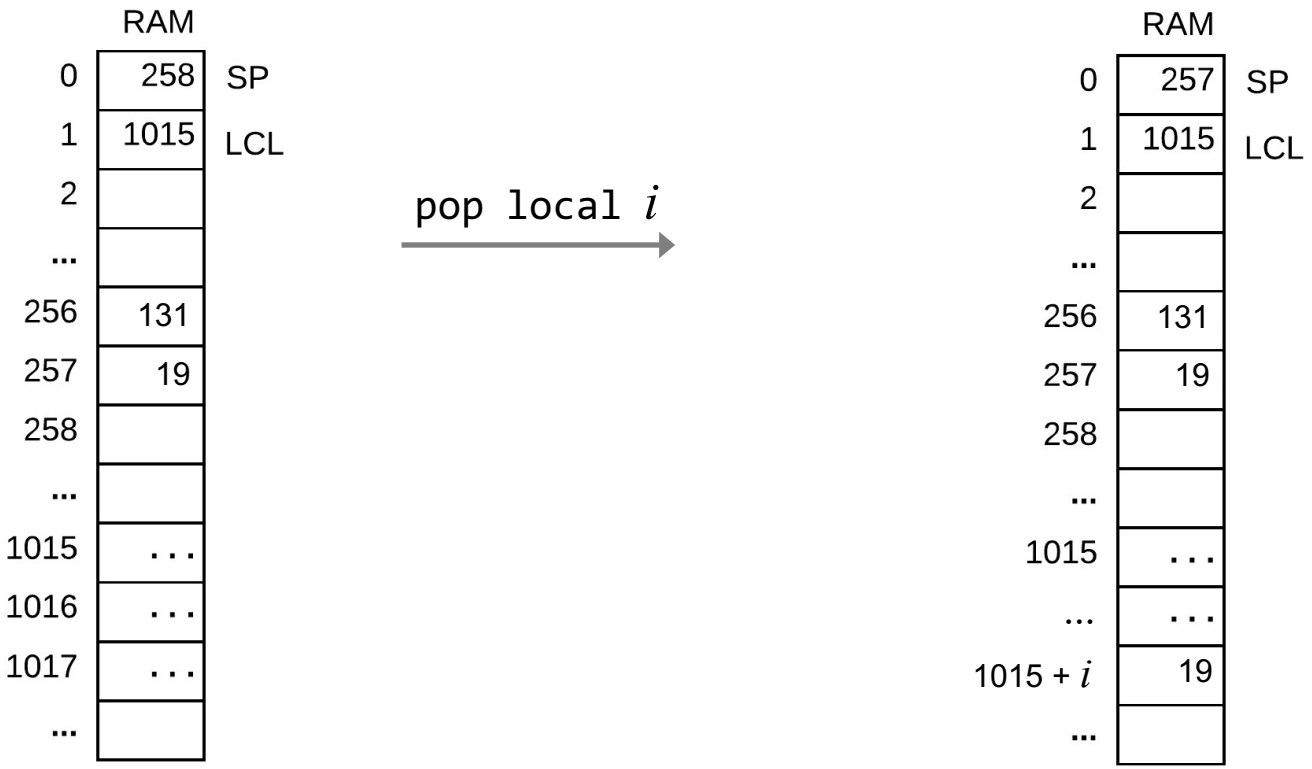
- The big picture: When the compiler translates a high-level *method*, it allocates a RAM block for representing the method's *local variables*
- At the VM level, this block is referred to as a virtual memory segment named `local`
- The base address of the local segment is kept in a pointer named `LCL`
- To access `local` *i*, access RAM[`LCL` + *i*] .

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

Abstraction:

Implementation:

- The `local` segment is allocated to a RAM block
- Here, the block's base address happens to be 1015 (arbitrary example)
- The base address of the `local` segment is kept in `LCL`

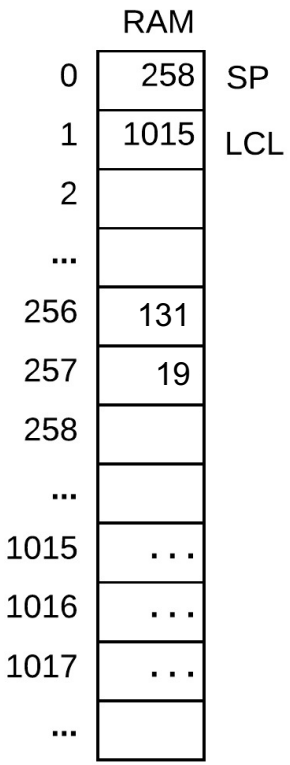


Implementing `push/pop local` *i*

Abstraction:

Implementation:

- The `local` segment is allocated to a RAM block
- Here, the block's base address happens to be 1015 (arbitrary example)
- The base address of the `local` segment is kept in `LCL`

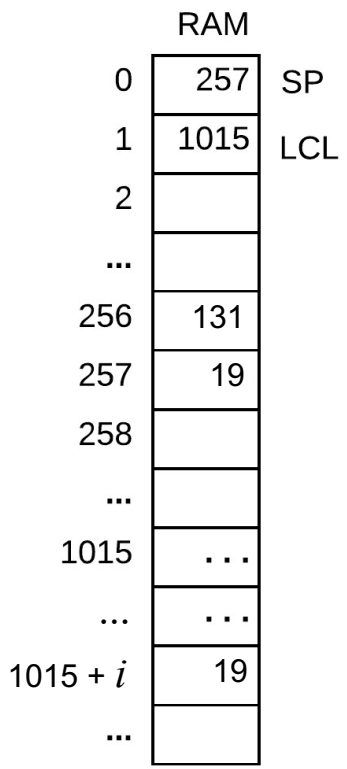


Pseudocode

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

Hack assembly:

You figure it out!



Implementing `push/pop local` *i*

VM code:

```
pop local i
```

VM translator

Assembly pseudo code:

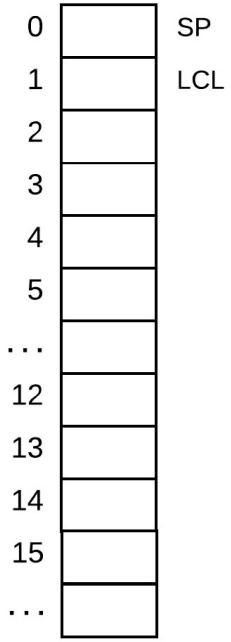
```
// pop local i
addr ← LCL + i
SP--
RAM[addr] ← RAM[SP]
```

```
push local i
```

VM translator

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

Host RAM



<u>The big picture:</u>

When the compiler compiles a high-level method, it 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 *entries* of the current array in the virtual segment `that`

Implemented the same way at the VM level

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

VM code:

```
pop local i
```

VM translator

Assembly pseudo code:

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

```
push local i
```

VM translator

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

Host RAM

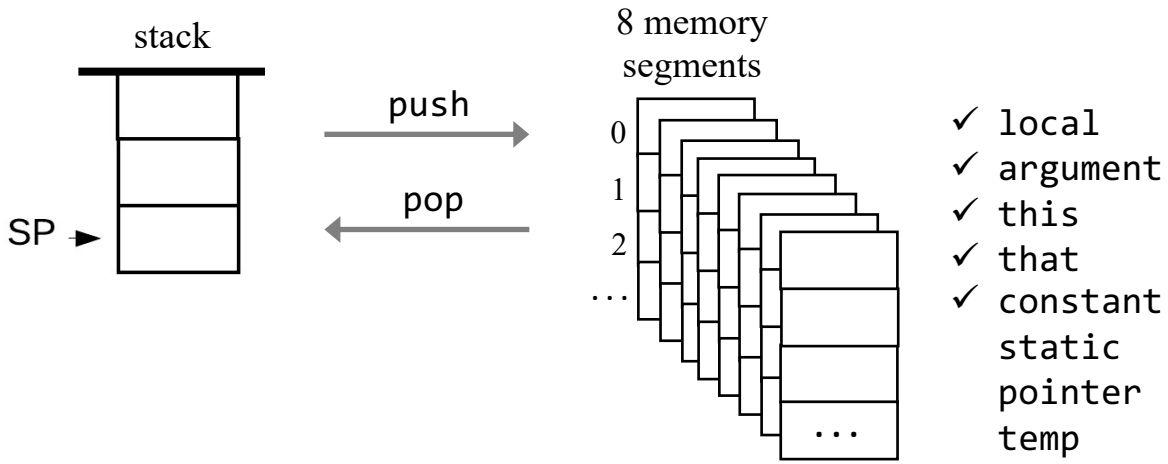
0		SP
1		LCL
2		ARG
3		THIS
4		THAT
5		
...		
12		
13		
14		
15		
...		

To access *segment* i where *segment* is `local`, `argument`, `this`, `that`:

access `RAM[`*segmentPointer* + i`]`

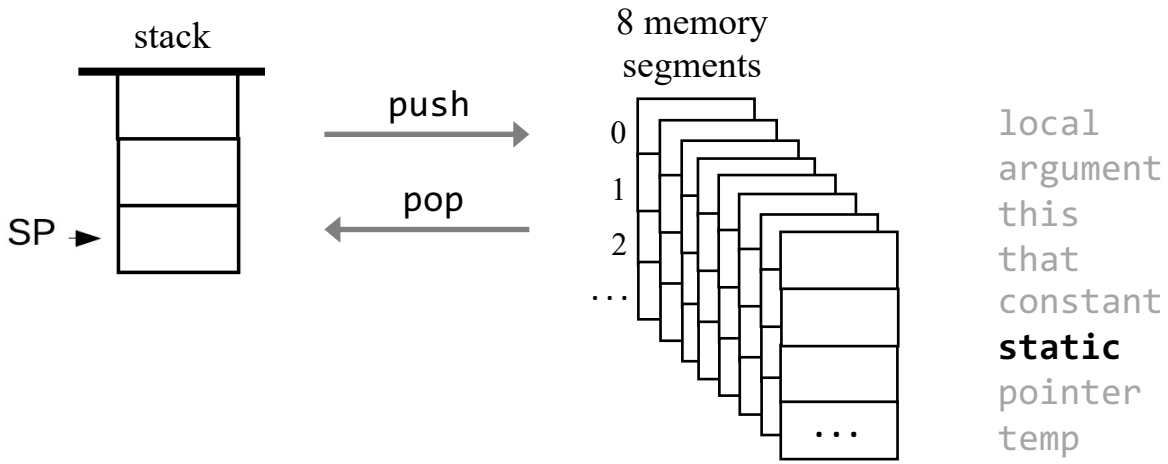
where *segmentPointer* is `LCL`, `ARG`, `THIS`, `THAT`

Implementing `push` / `pop` commands



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

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



The Big Picture

- A Jack program is a collection of one or more Jack classes
- Each class is a separate compilation unit
- A "class file" `Foo.jack` is compiled into a "VM file" named `Foo.vm`
- The compiler maps the class-level *static variables* on the VM segment `static`

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

<u>Implementation</u>

- The `static` segment is stored in a fixed RAM block, starting at address 16
- When translating a VM file named `Foo.vm`:

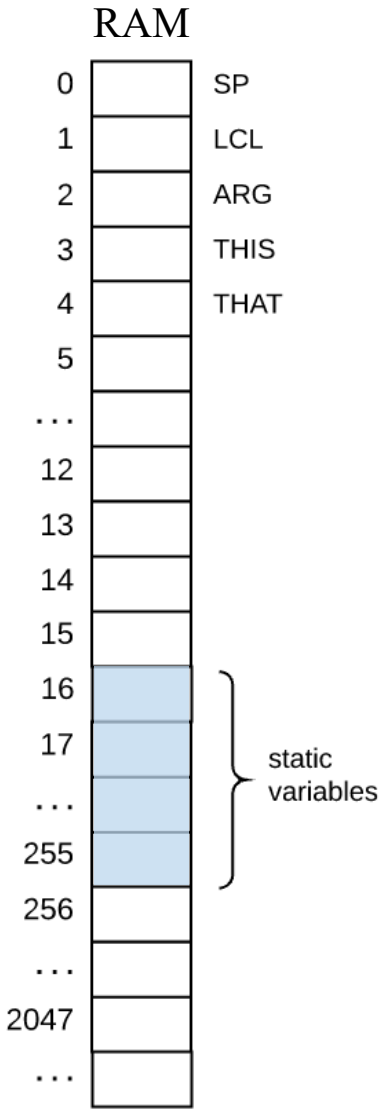
  To `push/pop static` *i*, generate assembly code that executes

  `push/pop Foo.`*i*

<u>Explanation:</u>

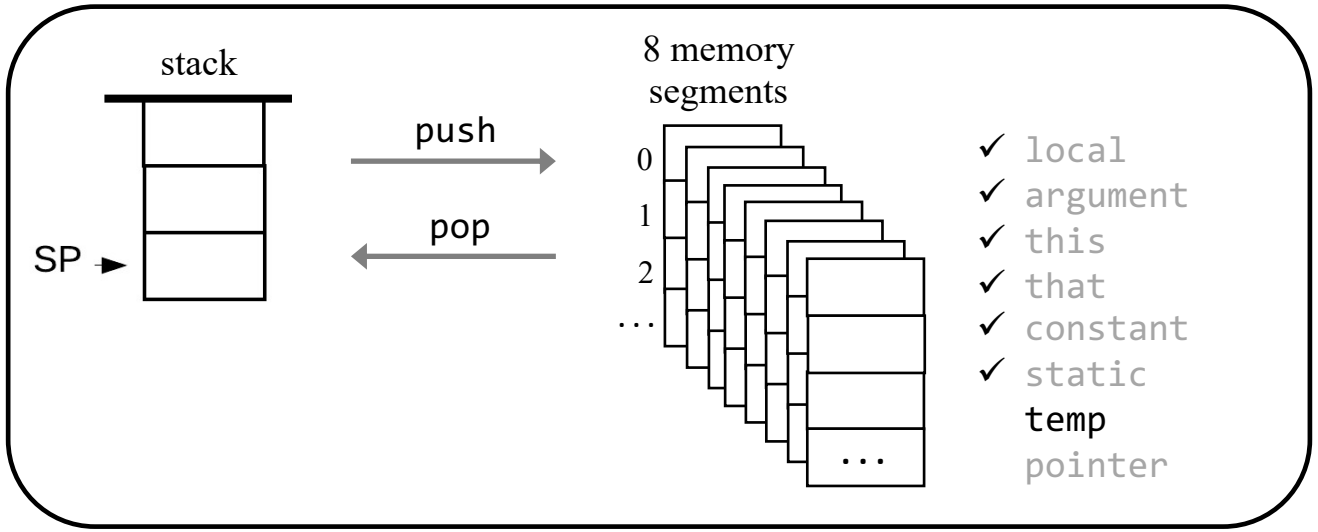
The Hack assembler will subsequently map every symbol `Foo.`*i* in the generated assembly code onto `RAM[16]`, `RAM[17]`, ..., `RAM[255]`

(exactly what we want…)



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

Abstraction

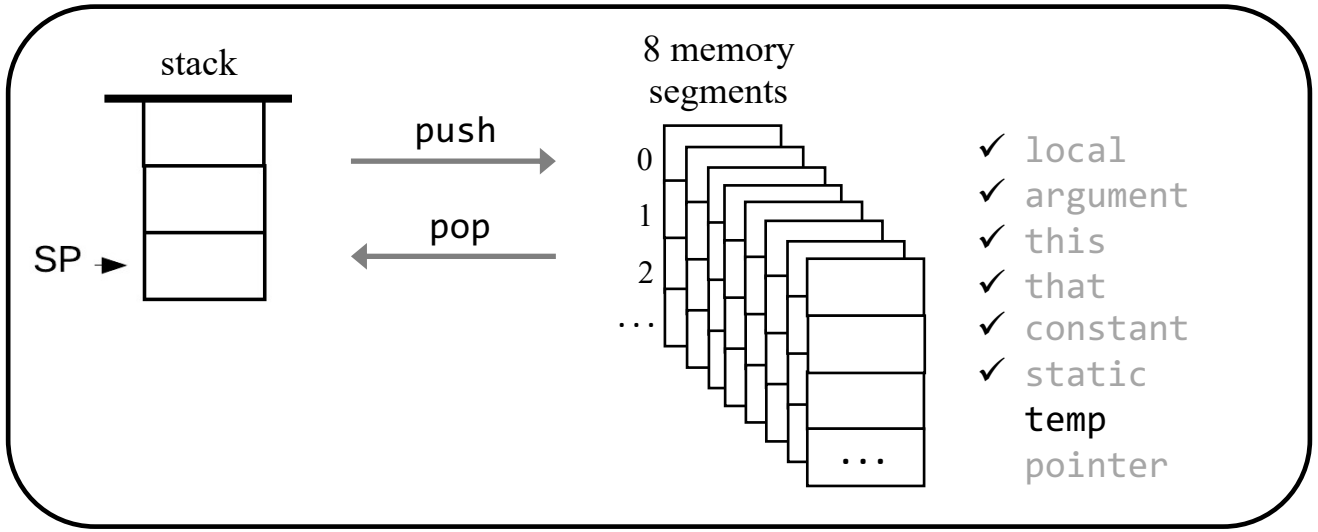


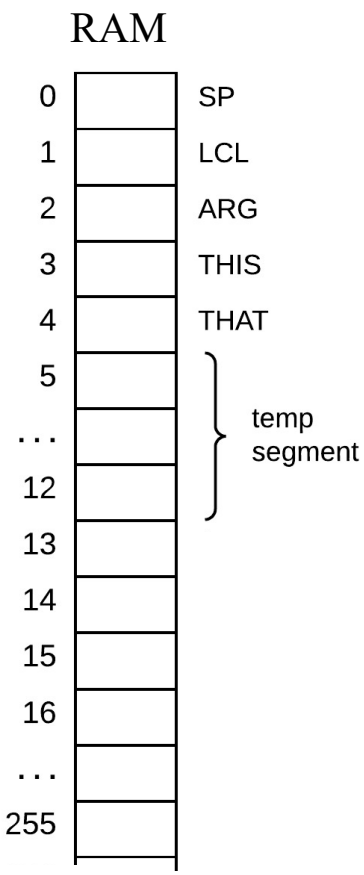
The Big Picture

- So far, all the variable kinds that we discussed came from the source code
- Sometimes, the compiler needs to add to the generated VM code some additional working variables
- Our VM implementation provides 8 such variables, stored in a segment named `temp`

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

Abstraction





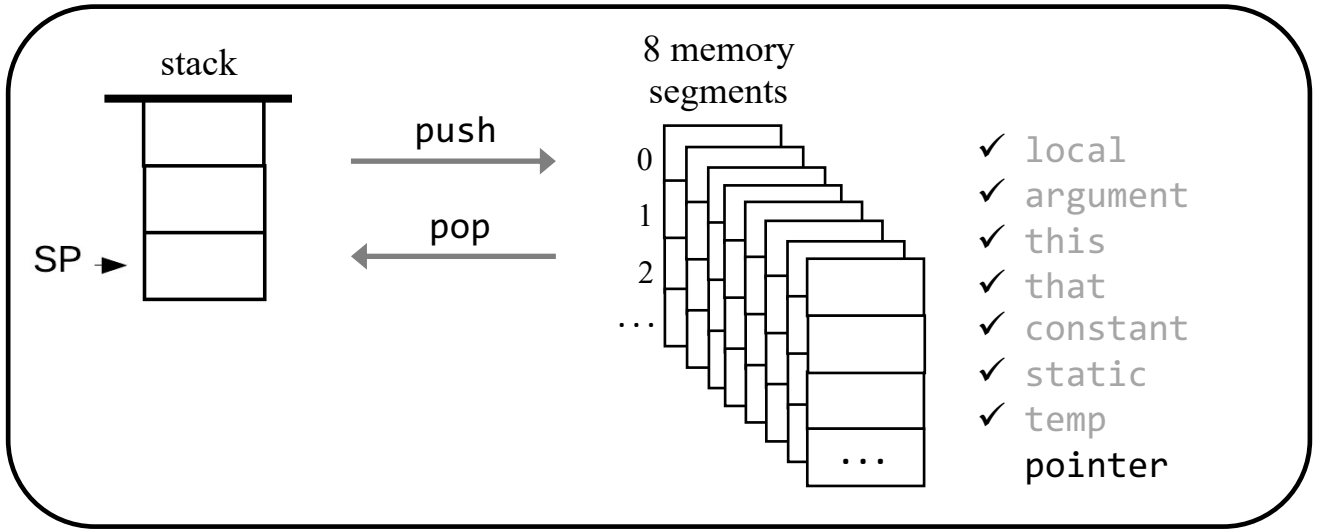
Implementation

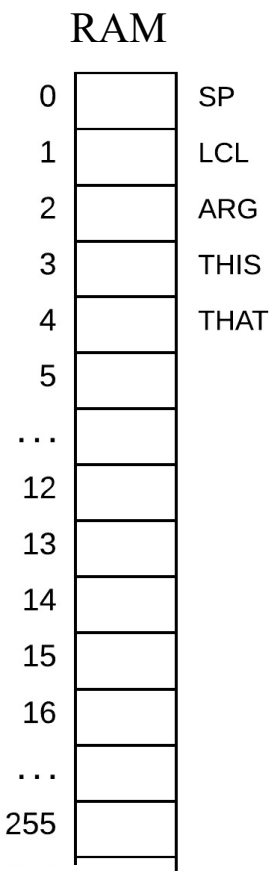
- The `temp` segment is stored in a fixed RAM block, from address 5 to 12
- To `push/pop temp` *i*, generate assembly code that executes

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

Implementing `push/pop` **`pointer`** *i*

Abstraction



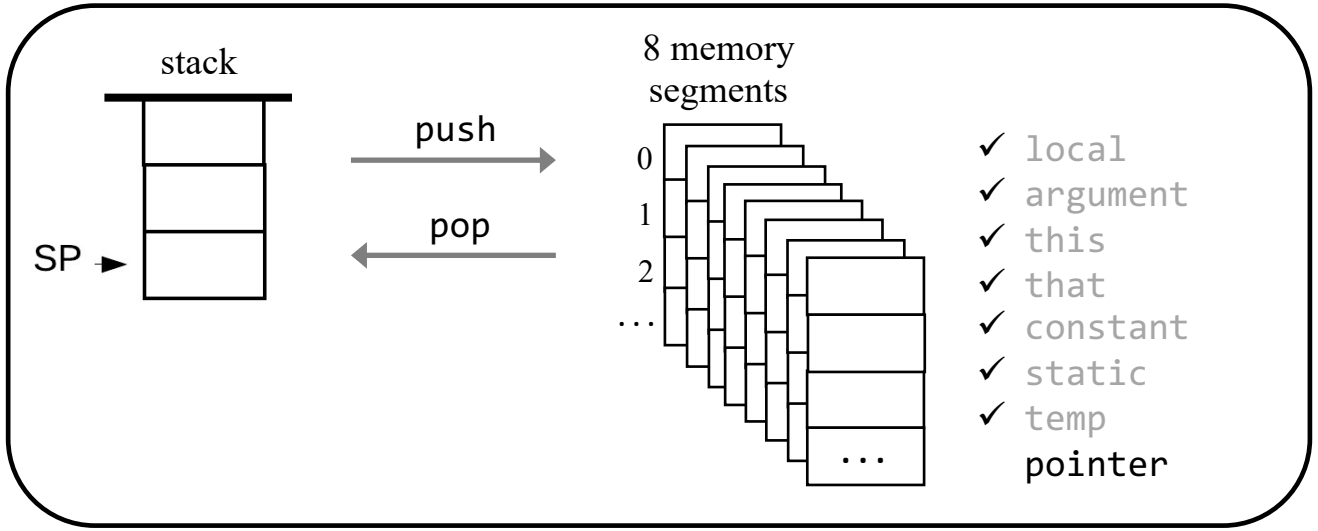


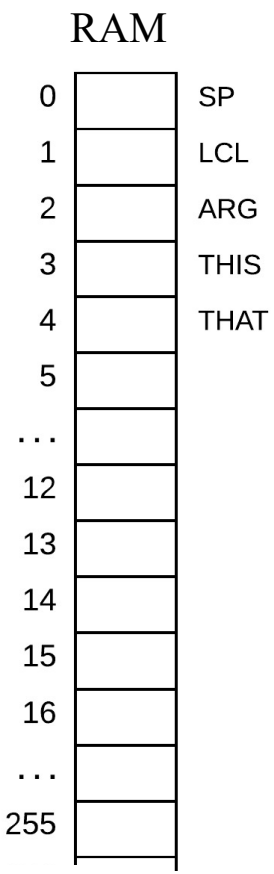
The big picture

- The `this` and `that` segments are allocated to the RAM dynamically
- Their role will be explained in chapter 10
- The compiler uses a 2-place memory segment called `pointer` for storing the current base addresses of `this` and `that`

Implementing `push/pop` **`pointer`** *i*

Abstraction





Implementation

- To `push/pop pointer 0`, generate assembly code that executes

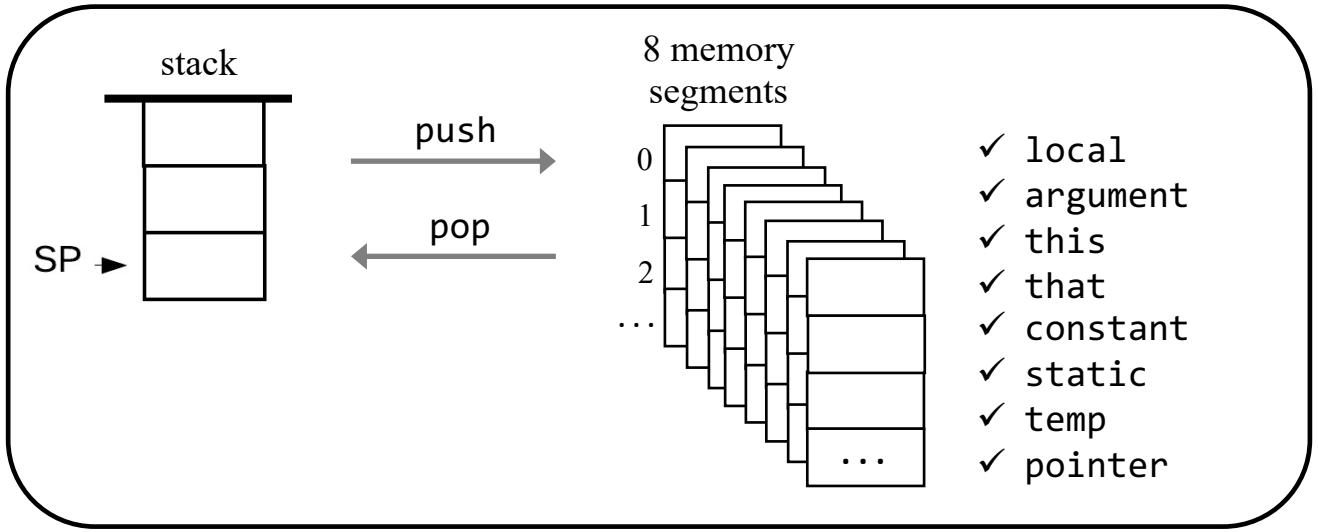
  `push/pop THIS`

- To `push/pop pointer 1`, generate assembly code that executes

  `push/pop THAT`

Implementing `push/pop`

Abstraction



Recap: We've implemented all the `push/pop` *segment i* commands

Note: We'll discuss the role (*semantics*) of the VM memory segments later.
For now, we focus only on how to handle their *syntax*.

The VM language: Implementation

<u>Push / pop commands</u>

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

<u>Arithmetic / Logical commands</u>

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

<u>Branching commands</u>

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

<u>Function commands</u>

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

Implementing the 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…

- `pop`*s* one or more values from the stack
- Computes a simple function on these values
- `push`es the computed value onto the stack

Implementation:

- We already know how to implement `push` / `pop`
- The rest is easy.

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`

This lecture

Branching commands

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

Function commands

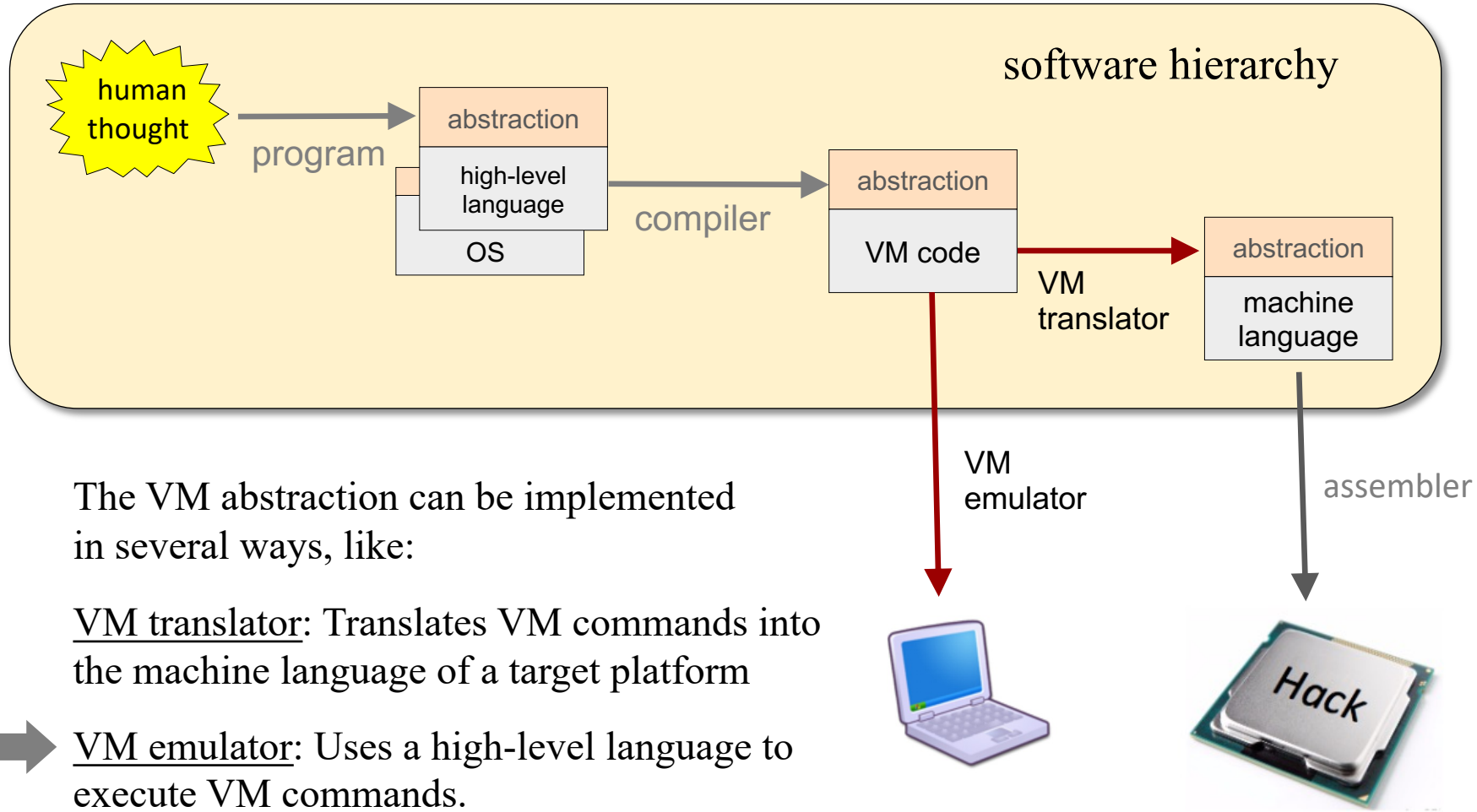
- `Function` *functionName nVars*
- `Call` *functionName nArgs*
- `return`

Next lecture

Virtual Machine I: Lecture plan

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

VM implementations

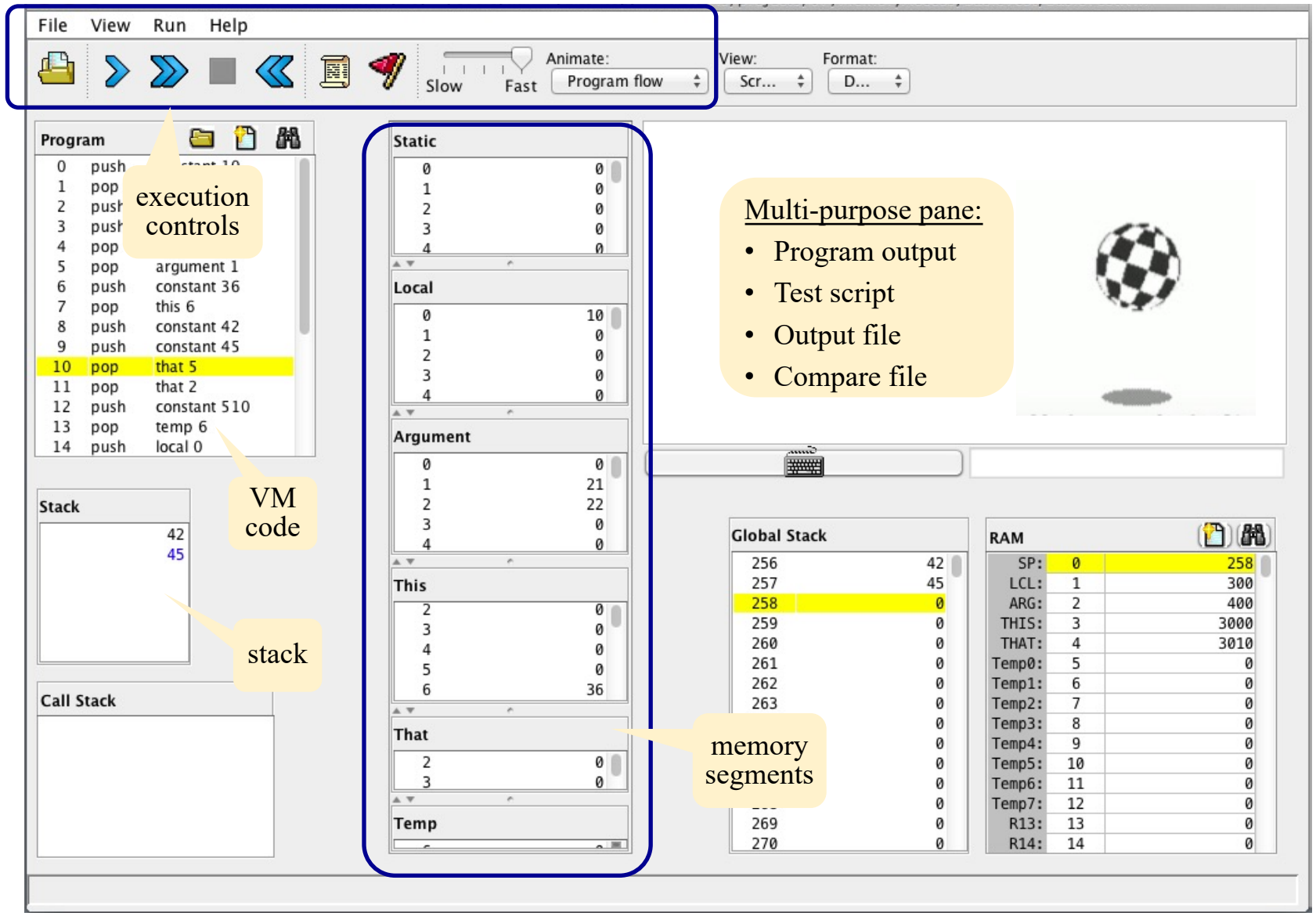


The VM abstraction can be implemented in several ways, like:

<u>VM translator</u>: Translates VM commands into the machine language of a target platform

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

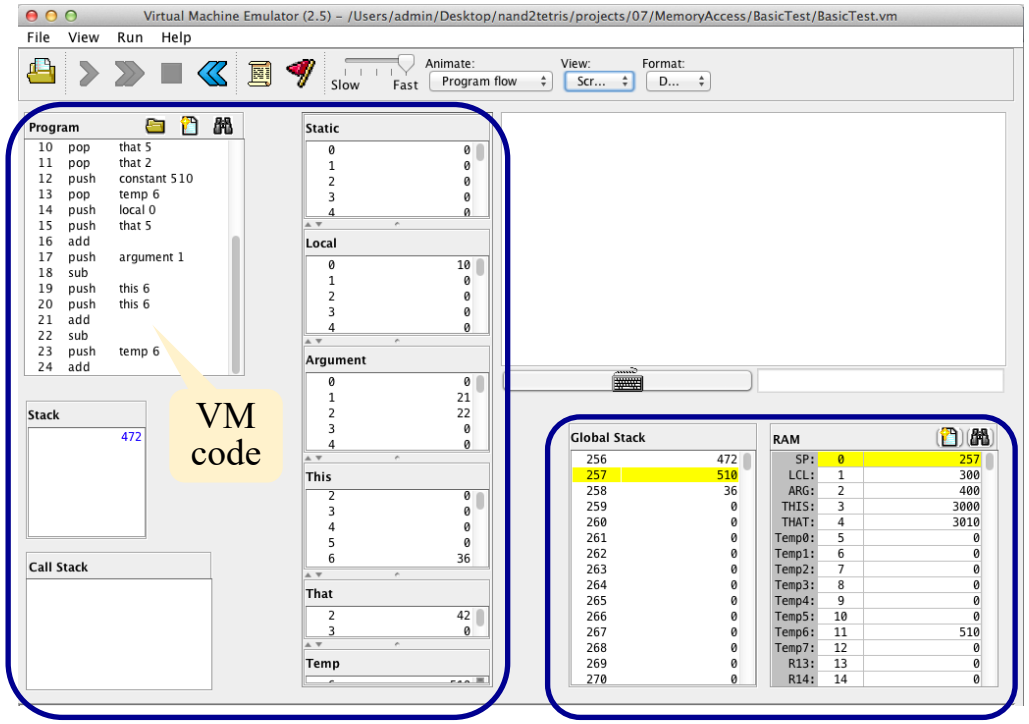
Our VM emulator



Emulating a VM program: Example

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
...
```



abstraction

implementation

Things to watch for:

- Stack state
- Memory segments states

How the stack and memory segments are realized on the host platform (Hack RAM)

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
...
```

BasicTestVME.tst

```
load BasicTest.vm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
// Specifies selected values to output, and their format:
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 ...

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

repeat 25 {         // BasicTest.vm has 25 VM commands
  vmstep;
}

// Outputs the values specified by the output-list
// (stack base + selected values from selected memory segments)
output;
```

In project 7, we supply test scripts that enable experimenting with the source VM programs in the VM emulator

- There is no need to delve into the code details of the supplied test scripts
- But, it helps to understand the general testing logic.

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
...
```

BasicTestVME.tst

```
load BasicTest.vm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
// Specifies selected values to output, and their format:
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 ...
```

Instructs to output the *stack base*, and selected *values* from *selected memory segments* (as they are stored in the host RAM)

BasicTest.out

BasicTest.cmp

```
|RAM[256]|RAM[300]|RAM[401]|RAM[402]|RAM[3006|RAM[3012|RAM[3015|RAM[11] |
|    472 |     10 |     21 |     22 |     36 |     42 |     45 |    510 |
```

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
...
```

BasicTestVME.tst

```
load BasicTest.vm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
// Specifies selected values to output, and their format:
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 ...





repeat 25 {        // BasicTest.vm has 25 VM commands
  vmstep;
}
// Outputs the values specified by the output-list
// (stack base + selected values from selected memory segments)
output;
```

The output file and the compare file are identical: The test is successful

BasicTest.out

```
|RAM[256]|RAM[300]|RAM[401]|RAM[402]|RAM[3006|RAM[3012|RAM[3015|RAM[11] |
|    472 |     10 |     21 |     22 |     36 |     42 |     45 |    510 |
```

BasicTest.cmp

```
|RAM[256]|RAM[300]|RAM[401]|RAM[402]|RAM[3006|RAM[3012|RAM[3015|RAM[11] |
|    472 |     10 |     21 |     22 |     36 |     42 |     45 |    510 |
```

Emulating a VM program: Testing

BasicTest.vm

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

BasicTestVME.tst

```
load BasicTest.vm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
// Specifies selected values to output, and their format:
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 ...
set sp 256,         // stack pointer
set local 300,      // base address of segment local
set argument 400,   // base address of segment argument
set this 3000,      // base address of segment this
set that 3010,      // base address of segment that
repeat 25 {         // BasicTest.vm has 25 VM commands
  vmstep;
}
// Outputs the values specified by the output-list
// (stack base + selected values from selected memory segments)
output;
```

- Normally, VM code begins with `function` *functioName* and ends with `return`
- In project 7 though, these commands are not yet implemented.
  Therefore, the project 7 translator does not allocate memory segments on the host RAM;
  This will done in project 8
- Meanwhile, in project 7 we use test script `set` commands that cause the VM emulator to anchor the memory segments in fixed (and quite arbitrary) RAM locations.

Demo

Virtual Machine I: Lecture plan

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

The big picture

Source code

```
push constant 2
push local 0
sub
push local 1
push constant 5
add
sub
pop local 2
...
```

Translation

Each VM command is translated into several assembly instructions

Target code

```
// push constant 2
@2
D=A
@SP
A=M
M=D
@SP
M=M+1
// push local 0
...
```

To write a VM translator, one must be familiar with:

- The source language (VM commands)
- The target language (Hack assembly language)

➡ How to map the VM stack and memory segments on the target platform in a standard (agreed-upon) way.

Standard VM mapping

The challenge

- You propose (to the world) a new VM model / language
- Developers can then map the VM's stack and memory segments on target platforms in several different ways
- This can lead to incompatible VM implementations

The solution: Standard VM mapping

Specifies how to map the VM's stack and memory segments on the target platform in an *agreed-upon way*

Benefits: Compatibility with…

- Other software systems (VM emulator, OS, …)
- Test files

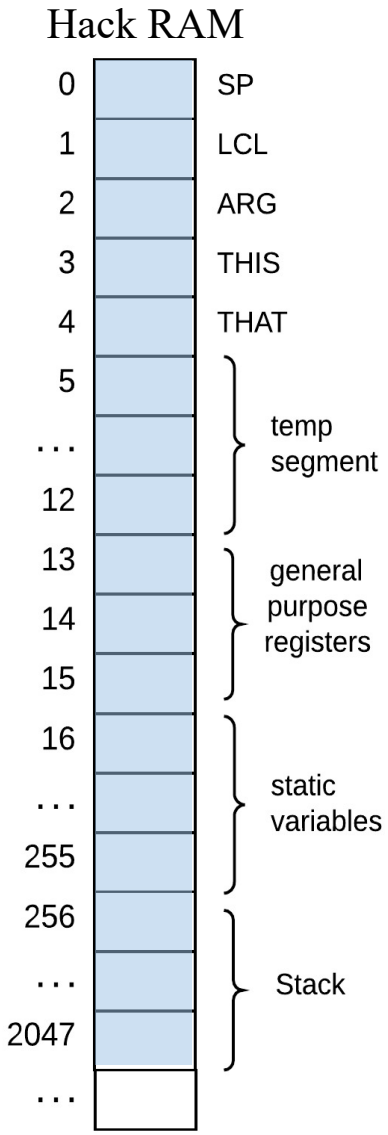
That conform to that standard

Standard VM mapping on the Hack platform

Hack RAM

0	
1	
2	
3	
4	
5	
...	
12	
13	
14	
15	
16	
...	
255	
256	
...	
2047	
...	

Standard VM mapping on the Hack platform



`local, argument, this, that:`

allocated dynamically to the `RAM` (in project 8).
The base addresses of these allocations are kept in the segment pointers `LCL`, `ARG`, `THIS`, `THAT`

accessing *segment i* should result in accessing `RAM[`*segmentPointer* + *i*`]`

`constant:` accessing `constant` *i* should result in supplying the constant *i*

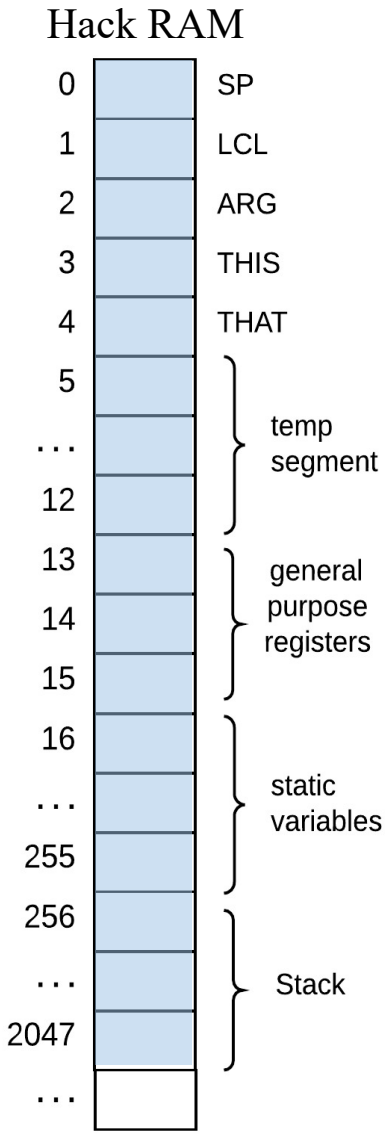
`static:` accessing `static` *i* within file `Foo.vm` should result in accessing the assembly variable `Foo.`*i*

`temp:` fixed segment, mapped on `RAM` addresses 5-12.
accessing `temp` *i* should result in accessing `RAM[5 +` *i*`]`

`pointer:` fixed segment, mapped on `RAM` addresses 3-4.

accessing `pointer 0` should result in accessing `THIS`
accessing `pointer 1` should result in accessing `THAT`

Standard VM mapping on the Hack platform



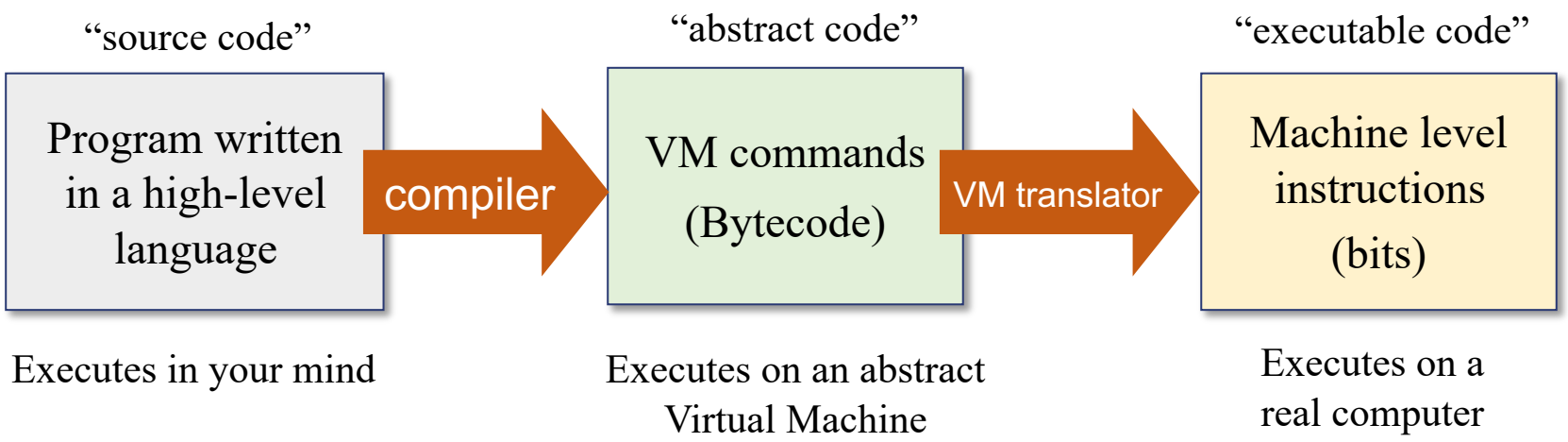
The assembly code generated by a VM translator that conforms to the standard mapping should use the following variable symbols:

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.

Virtual Machine I: Lecture plan

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

The Big Picture: Program compilation



The VM translator

VM code (*fileName*.vm)

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

VM translator

Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
```

... additional assembly code that completes the implementation of `push constant 17`

The VM translator

VM code (*fileName*.vm)

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

VM translator

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
... generated assembly code that implements push local 2
```

The VM translator

VM code (*fileName*.vm)

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

VM translator

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
... generated assembly code that implements push local 2
// add
... generated assembly code that implements add
```

The VM translator

VM code (*fileName*.vm)

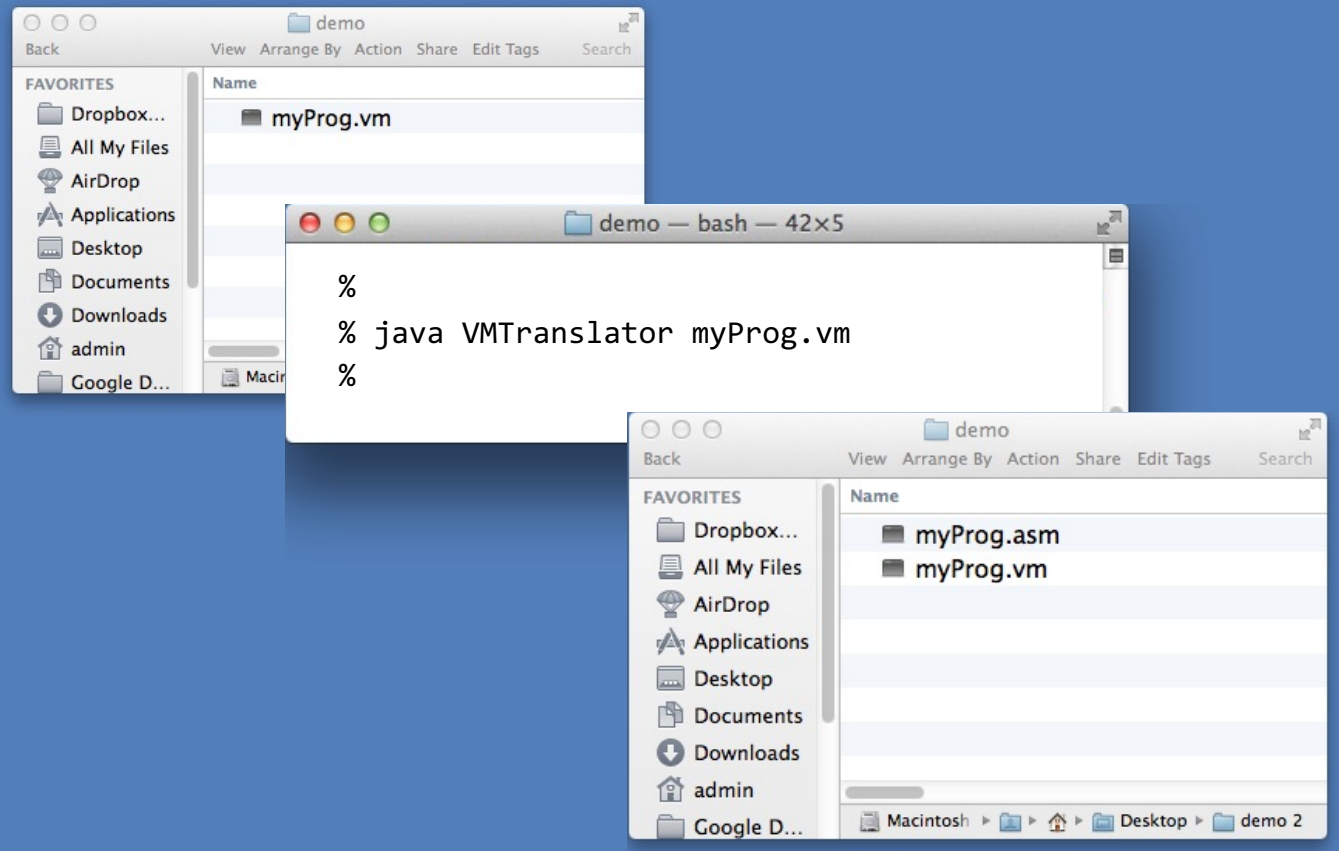
```
...
push constant 17
push local 2
add
pop argument 1
...
```

VM translator

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
... generated assembly code that implements push local 2
// add
... generated assembly code that implements add
// pop argument 1
... generated assembly code that implements push argument 1
...
```

The VM translator: usage



The VM translator: Implementation notes

Proposed design:

- `Parser`: parses each VM command into its lexical elements
- `CodeWriter`: writes the assembly code that implements the parsed command
- `Main`: drives the process (`VMTranslator`)

`Main` (`VMTranslator`)

Input: *fileName*`.vm`

Output: *fileName*`.asm`

`Main` logic:

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

Parser API

- 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.

Parser API

- 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:

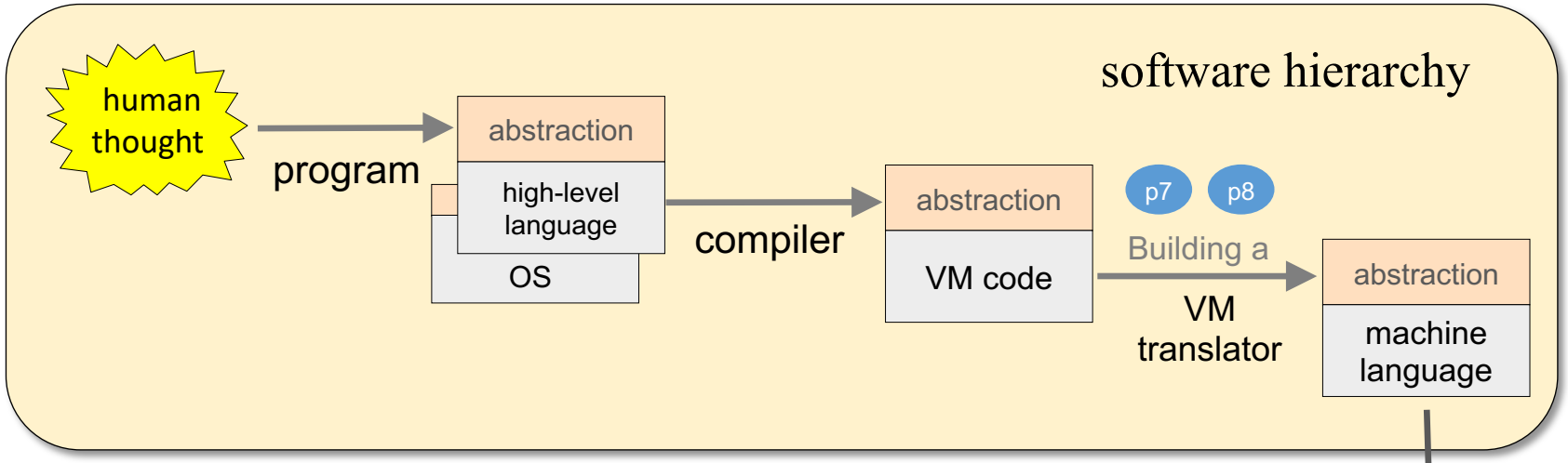
Routine	***Arguments***	***Returns***	***Function***
constructor	output file / stream	—	Opens an output file / stream and gets ready to write into it.
`writeArithmetic`	command (string)	—	Writes to the output file the assembly code that implements the given arithmetic-logical command.
`WritePushPop`	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.

More routines will be added to this module in Project 8,
when we complete the implementation of the VM translator.

Virtual Machine I: Lecture plan

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

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 handles the VM *branching* and *function* commands.

Project 7

To test the translation:

Run the generated code on the target platform (Hack)

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 that implements push local 2
// add
... generated assembly code that implements add
// pop argument 1
... generated assembly code that implements push argument 1
...
```

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 (source)

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

BasicTest.vm (example)

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

BasicTest.asm

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

Generated by your VM translator

For each test program *xxx*.vm we supply 3 test files: *xxx*VME.tst, *xxx*.tst and *xxx*.cmp

For each test program *xxx*.vm:

where *xxx* stands for SimpleAdd, StackTest, BasicTest, PointerTest, StaticTest

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.

Project 7: Tools and resources

- Source, test, compare files: `nand2tetris/projects/07`
- Experimenting with the source VM programs: the supplied *VM emulator*
- Translating the source VM programs into assembly: your *VM translator*
- Testing the resulting assembly code: the supplied *CPU emulator*
- Programming language for implementing your VM translator: Java, Python, ...
- Tutorials: VM emulator, CPU emulator (`nand2tetris` web site)
- Reference: chapter 7 in *The Elements of Computing Systems*

Project 7

Virtual Machine I: Lecture plan

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

Perspective

(A subset of historical notes and additional issues)

- Brief history of VMs and two-tier compilation:
  - p-code
  - Java origins
  - Cellphones
- How close is our VM to Java's JVM?
- Efficiency and optimization
- Different VM implementations:
  - Stack machine
  - Register machine
  - Other approaches.

Recap / Next

✓ 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

Project 7: build a basic VM translator that handles the VM language
stack arithmetic and *push*/*pop* commands

Project 8: Complete the basic translator into a full-scale VM translator.

Chapter 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