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: lecture plan

Overview

- The road ahead
- Program compilation

VM abstraction

- the stack
- memory segments

VM implementation

- the stack
- memory segments

VM implementation platforms

- VM emulator
- VM translator

The VM translator

- Proposed implementation
- Building it (project 7)

Hello World

Jack program

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

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:

- Program execution
- Writing on the screen
- Handling `class`, `function` ...
- Handling `do`, `while`, ...
- function call and return
- operating system
- ...

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

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;
    }
}
```

abstraction

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

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

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;
    }
}
```

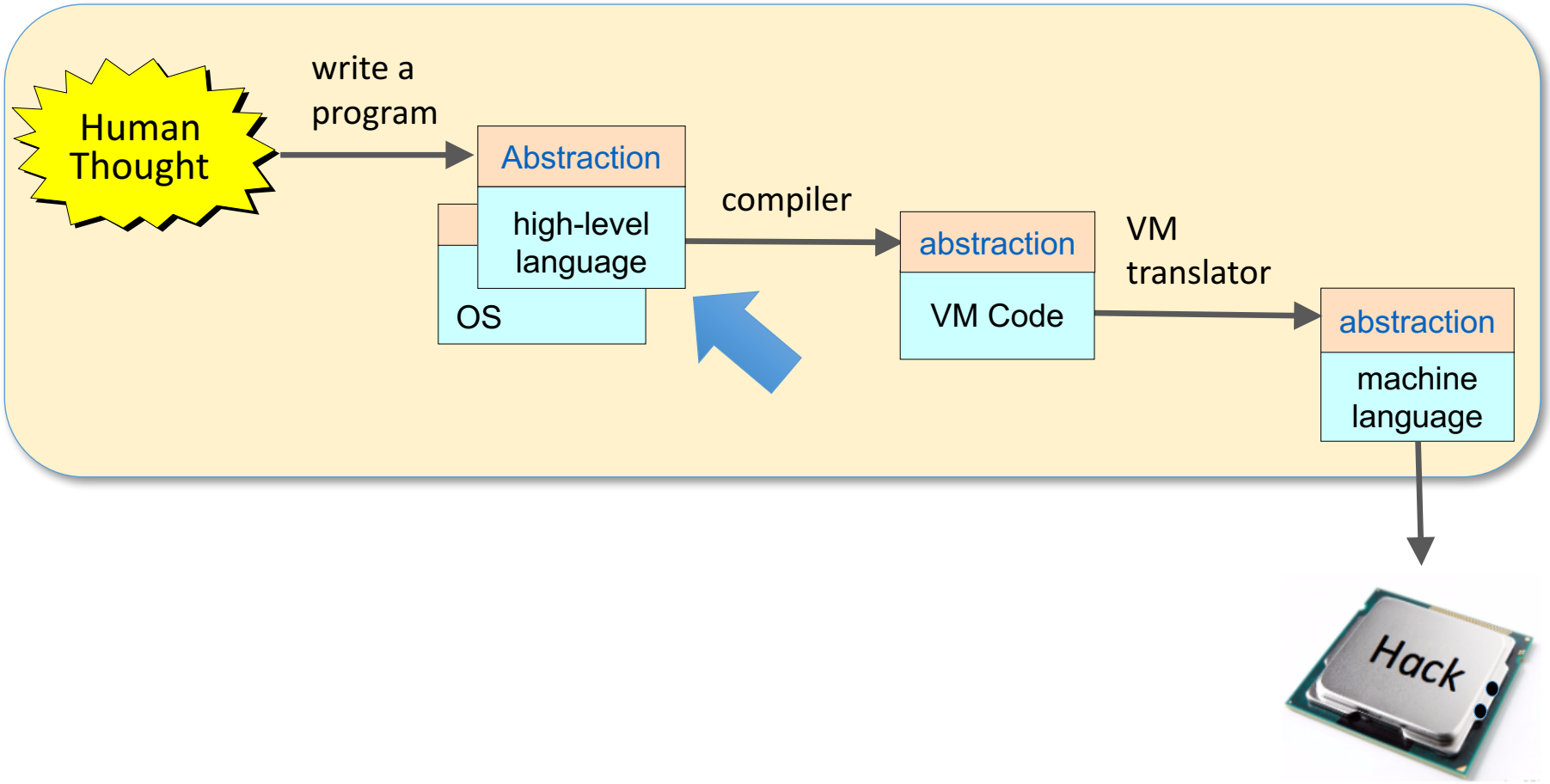
abstraction

Q: What makes the abstraction work?

A:
- Assembler
- Virtual machine
- Compiler
- Operating system

Chapters 6-12

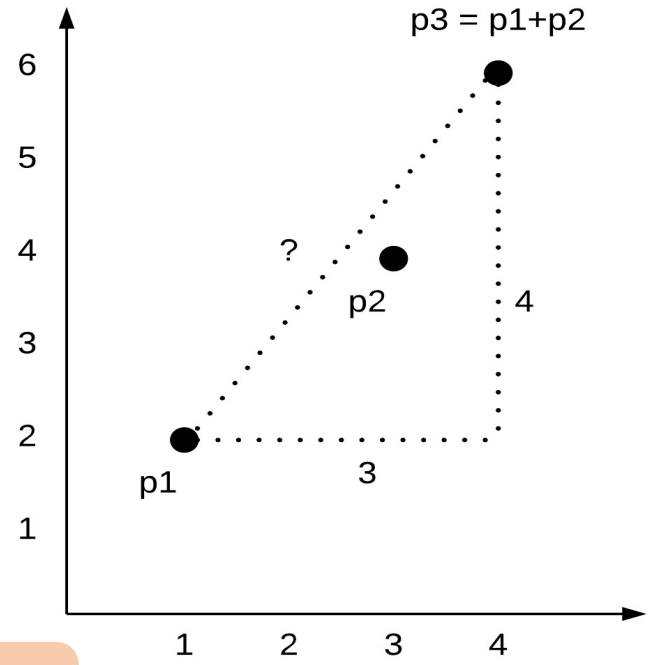
The big picture



High-level programming

```
/** Demo: working with 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;
  }
}
```

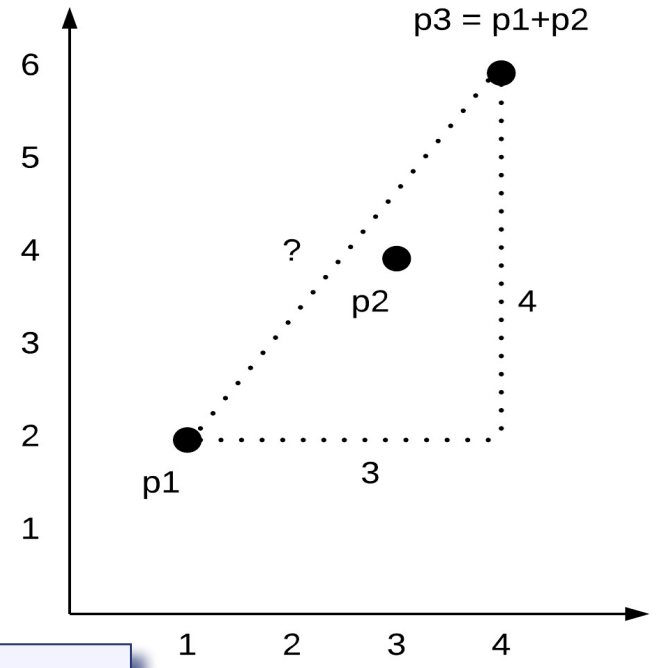
Written in the Jack language



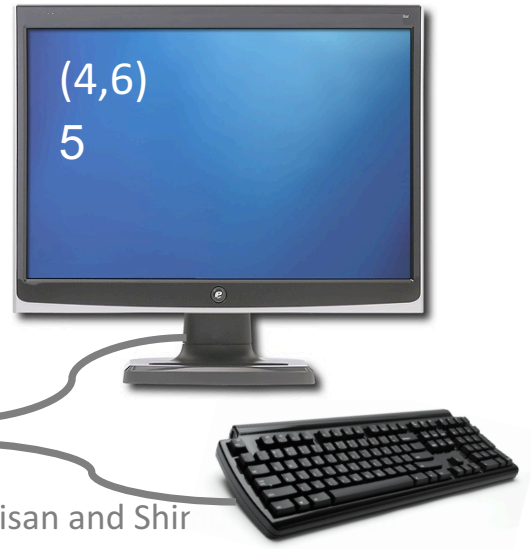

```
class Point {    // API

   /** 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 programming

```
/** Demo: working with 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;
  }
}
```




```
/** Represents a Point */
class Point {
  field int x, y;
  static int pointCount;
  /** Constructs a new point */
  constructor Point new(int ax, int ay) {
      let x = ax;
      let y = ay;
      let pointCount = pointCount + 1;
      return this;
  }
  // More Point methods...
}
```



High-level programming / `Point` class

```
/** Represents a Point; Stored in the file Point.jack */
class Point {
    field int x, y; // the coordinates of this point
    static int pointCount; // the number of Point objects constructed so far
    /** Constructs a new point 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 Point objects constructed so far */
    method int getPointCount() { return pointCount; }

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

High-level programming / `Point` class

```
/** Represents a Point; Stored in the file Point.jack */
class Point {
    field int x, y; // the coordinates of this point
    static int pointCount; // the number of Point objects constructed so far

    // The methods from the previous slide...

    /** Returns the Cartesian 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;
    }
}  // class Point ends here
```

Jack has the look and feel of a typical high-level, object-based language

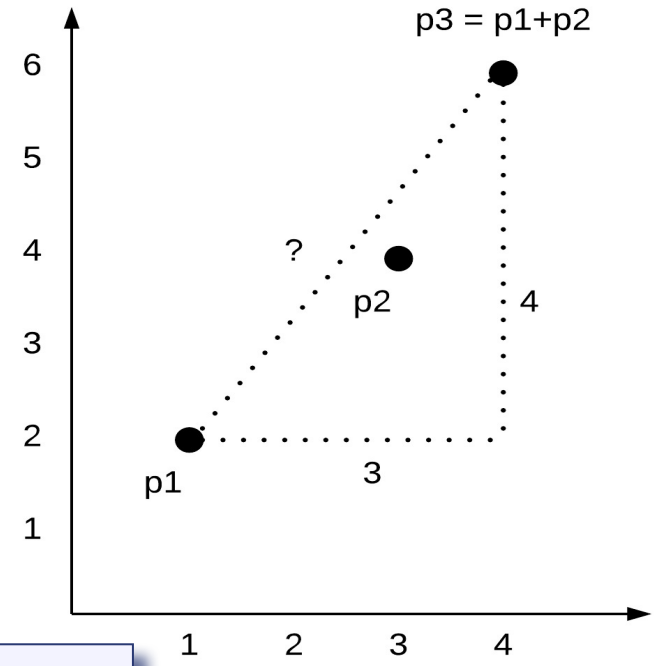
Recap

```
/** Demo: working with 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;
  }
}
```

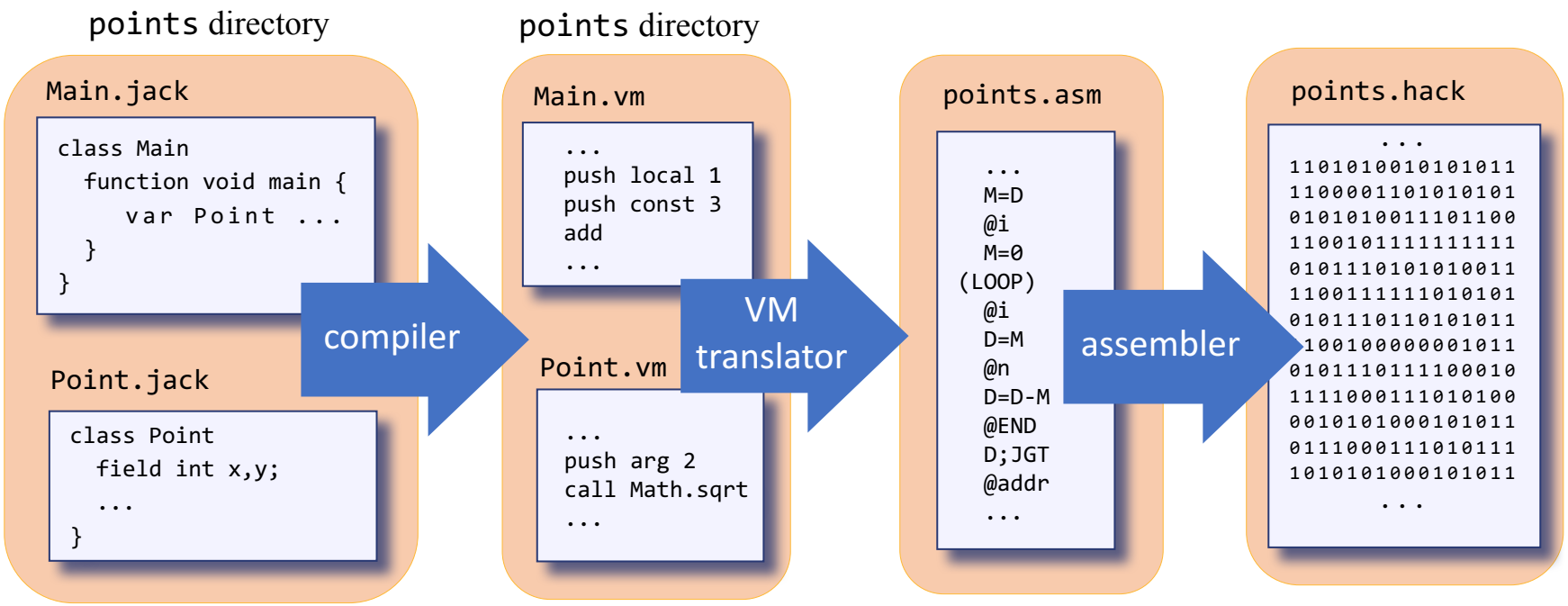
Main.jack

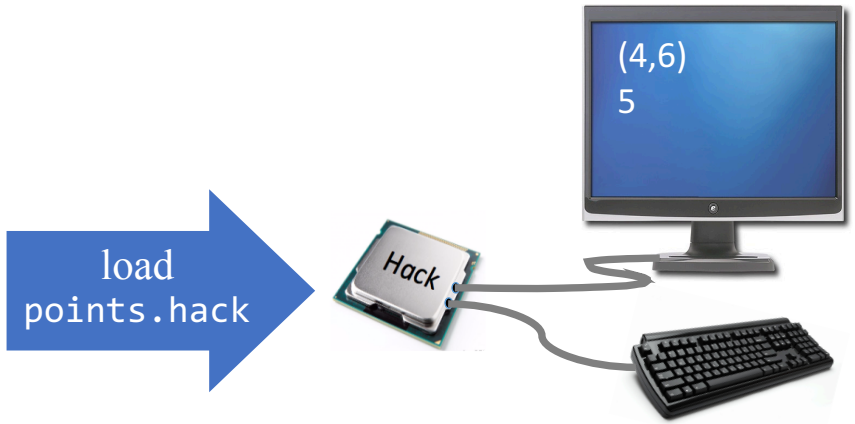
```
/** Represents a Point */
class Point {
  field int x, y;
  static int pointCount;
  /** Constructs a new point */
  constructor Point new(int ax, int ay) {
      let x = ax;
      let y = ay;
      let pointCount = pointCount + 1;
      return this;
  }
  // More Point methods...
}
```

Point.jack

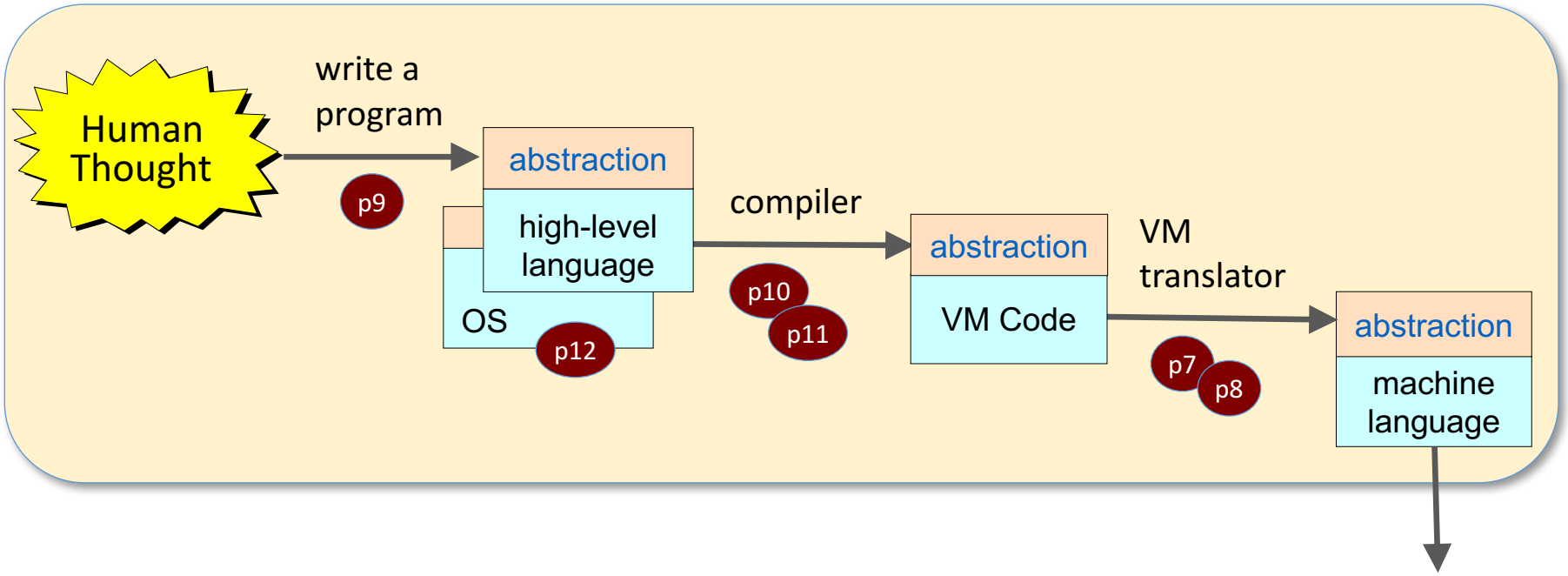


From high-level to low-level





The road ahead



Projects:

p7, p7: building a virtual machine

p9: writing a computer game

p10, p11: developing a compiler

p12: developing an operating system

Virtual machine: lecture plan

Overview
- ✓ The road ahead
- Program compilation

VM abstraction
- the stack
- memory segments

VM implementation
- the stack
- memory segments

VM implementation platforms
- VM emulator
- VM translator

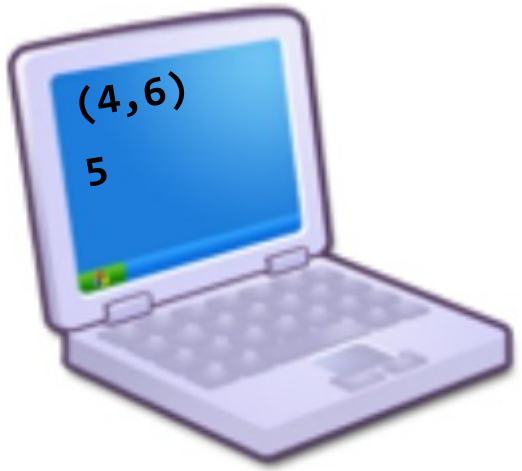
The VM translator
- Proposed implementation
- Building it (project 7)

The big picture

High-level program

```
/** Demo: working with 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();  // prints (4,6)
     do Output.println();
     do Output.printInt(p1.distance(p3));
     return;
  }
}
```

```
/** Represents a Point */
class Point {
  field int x, y;
  static int pointCount;
  /** Constructs a new point */
  constructor Point new(int ax, int ay) {
      let x = ax;
      let y = ay;
      let pointCount = pointCount + 1;
      return this;
  }
  ...
}
```



The big picture

High-level program

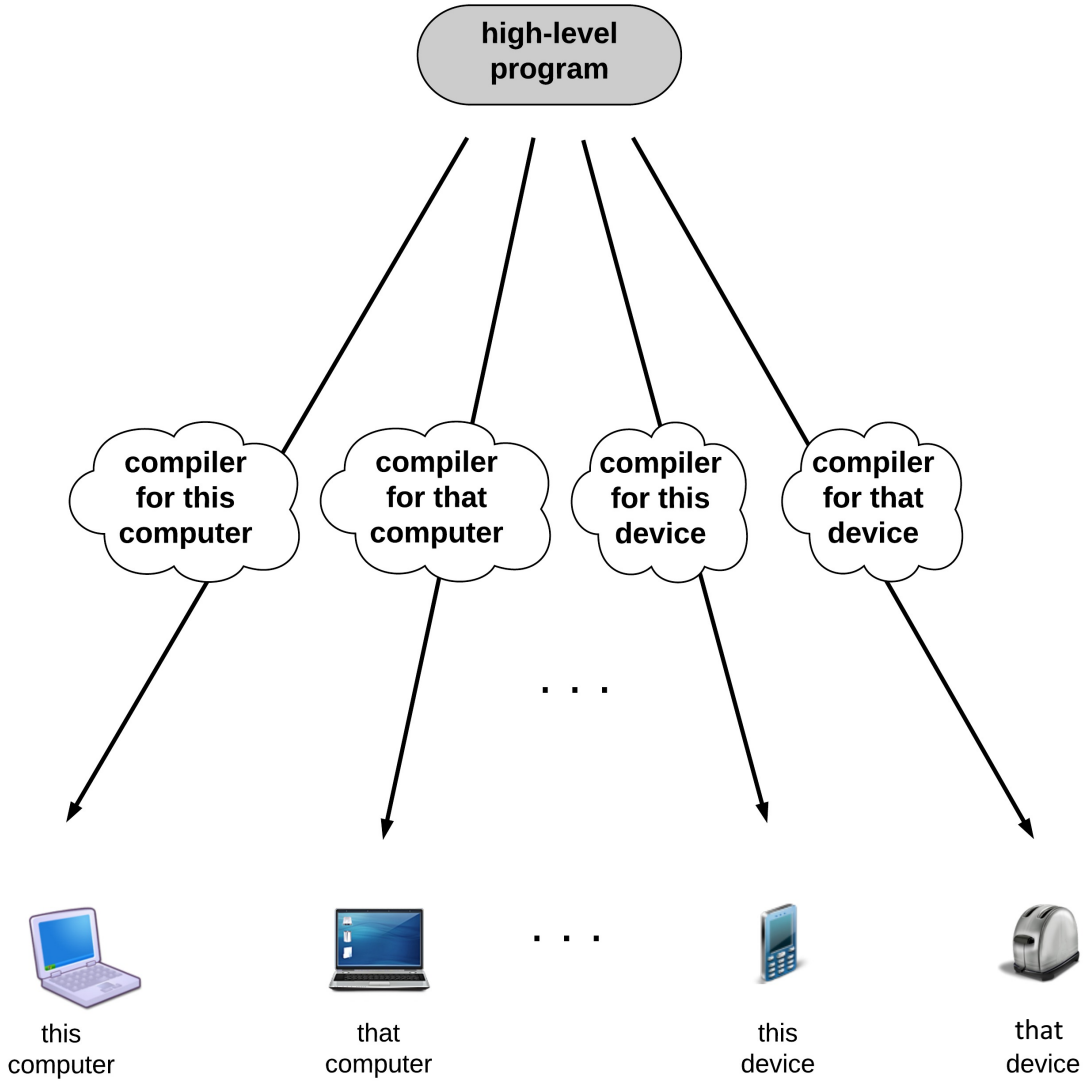
```
/** Demo: working with Point objects */
class Main {
  function void main() {
     var Point p1, p2, p3;
     let p1 = Point.new(1,2);
     let p2 = Point.new(3,4);
     let p3 = p1.plus(p2);
     do p3.print();  // prints (4,6)
     do Output.println();
     do Output.printInt(p1.distance(p3));
     return;
  }
}
```

```
/** Represents a Point */
class Point {
  field int x, y;
  static int pointCount;
  /** Constructs a new point */
  constructor Point new(int ax, int ay) {
      let x = ax;
      let y = ay;
      let pointCount = pointCount + 1;
      return this;
  }
  ...
}
```

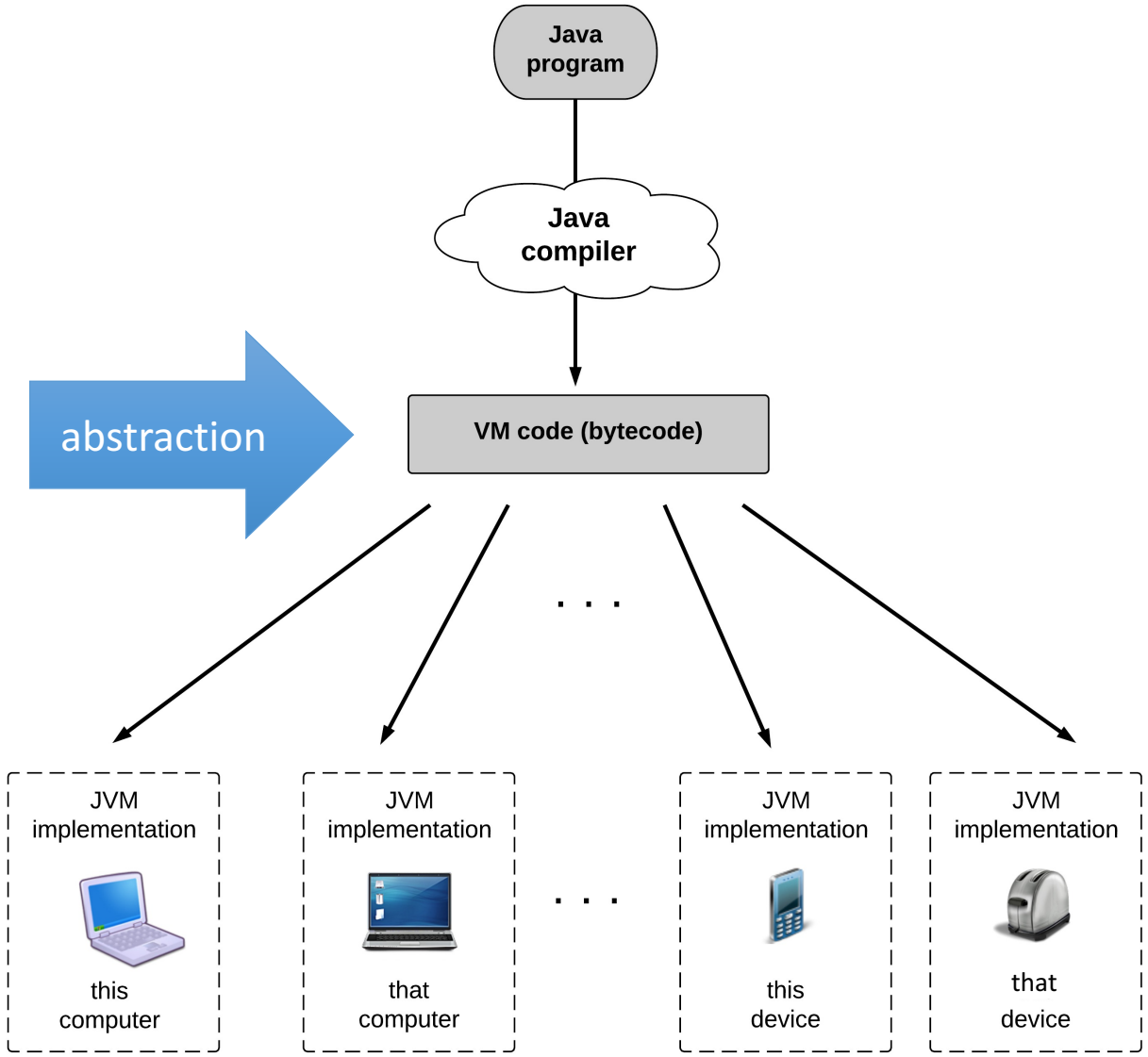
Low-level code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111110000010000
 ...
```

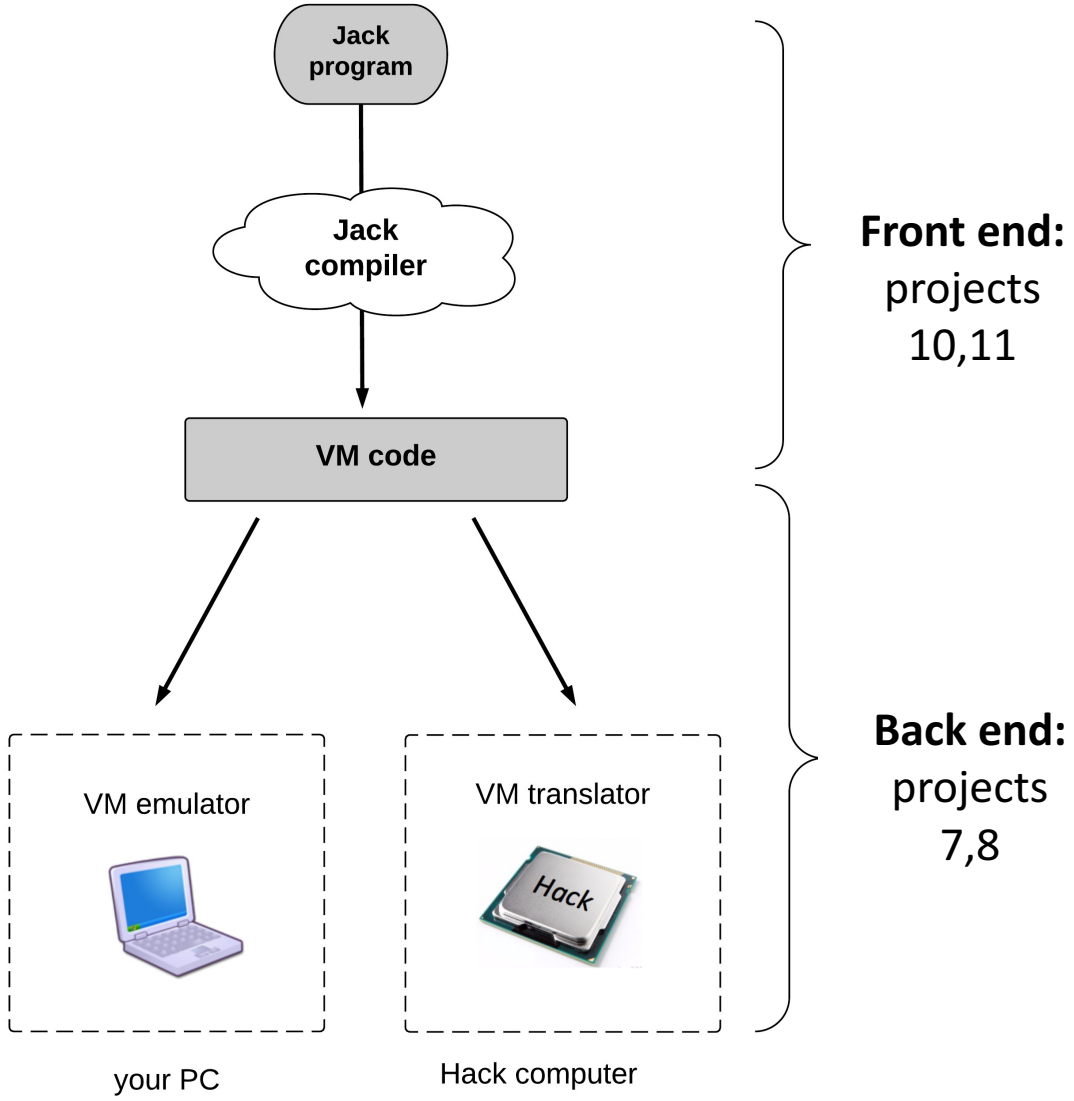
Program compilation: 1-tier



Program compilation: 2-tier



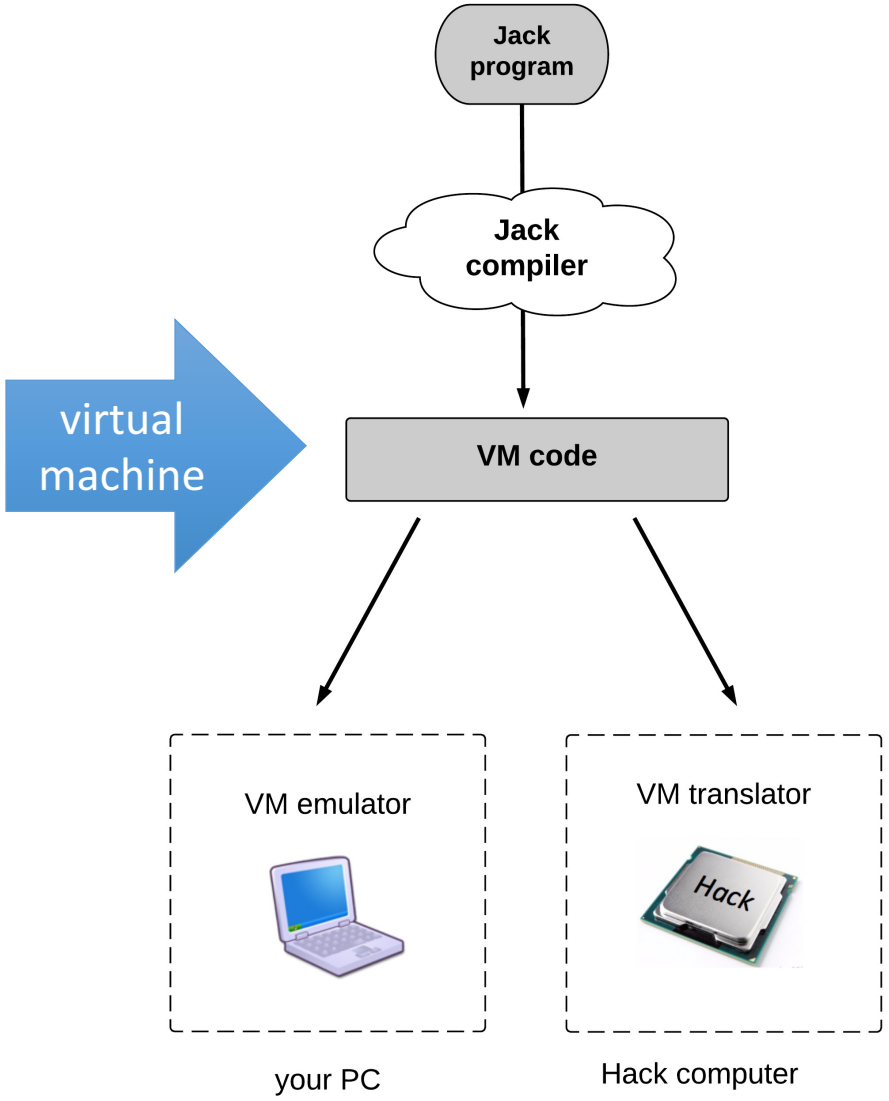
Jack compilation

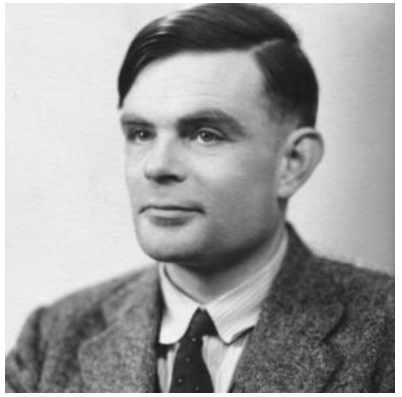


Front end:
projects
10,11

Back end:
projects
7,8

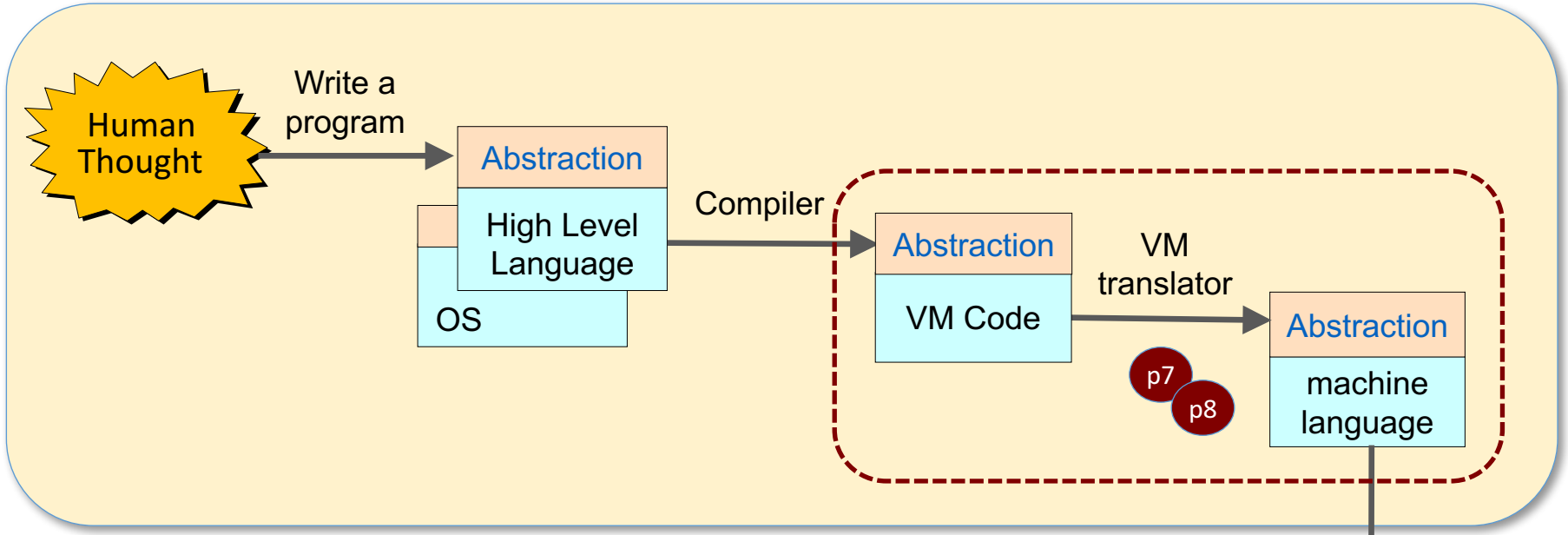
Jack compilation





Alan Turing (1912-1954)

The big picture



Virtual Machine (chapters 7-8)

- Understanding the VM abstraction
- Building a VM implementation

Take home lessons

- Compilation (big picture)
- Virtualization
- VM abstraction
- Stack processing
- VM implementation
- Pointers
- Programming.

Virtual machine: lecture plan

Overview
- ✓ The road ahead
- ✓ Program compilation

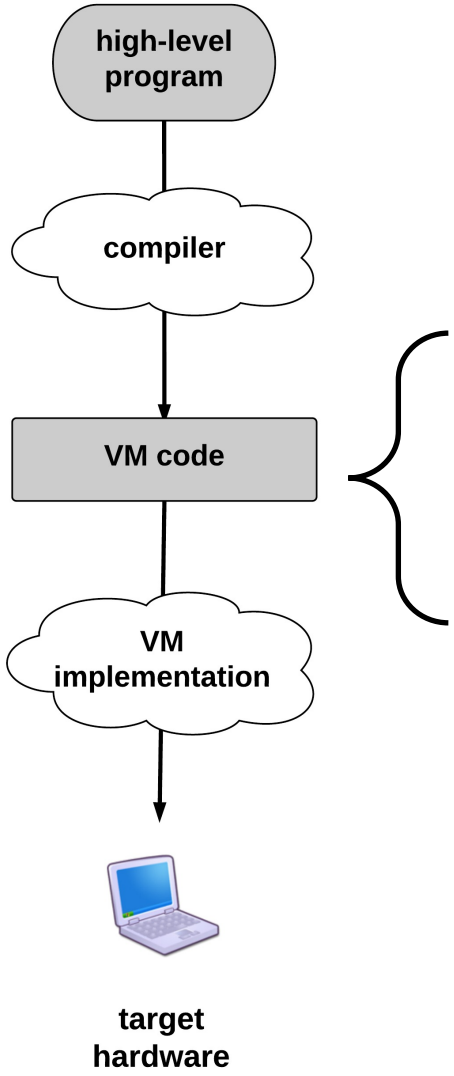
VM abstraction
- the stack
- memory segments

VM implementation
- the stack
- memory segments

VM implementation platforms
- VM emulator
- VM translator

The VM translator
- Proposed implementation
- Building it (project 7)

The big picture



Stack machine abstraction

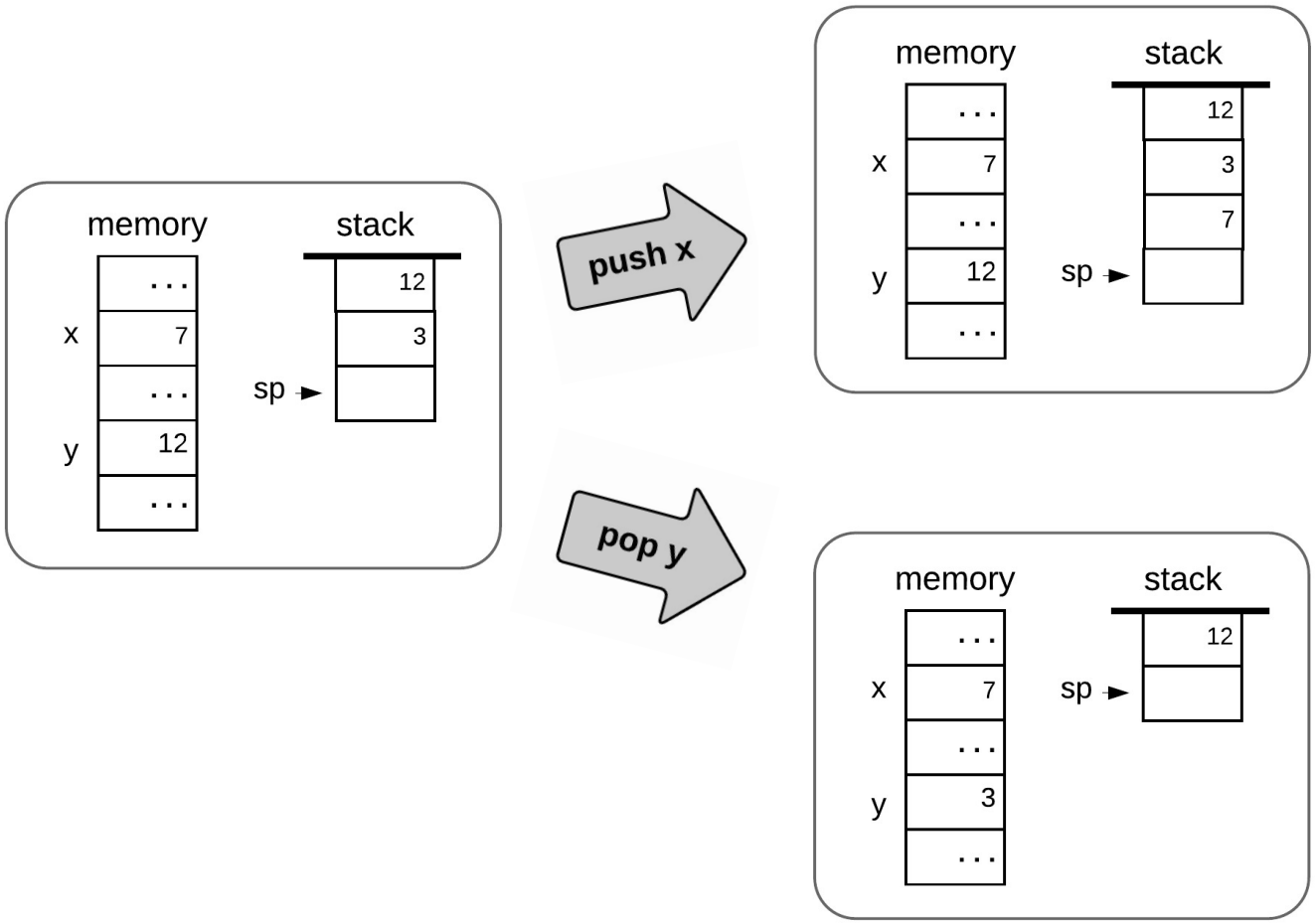
- Architecture
- Commands

Stack

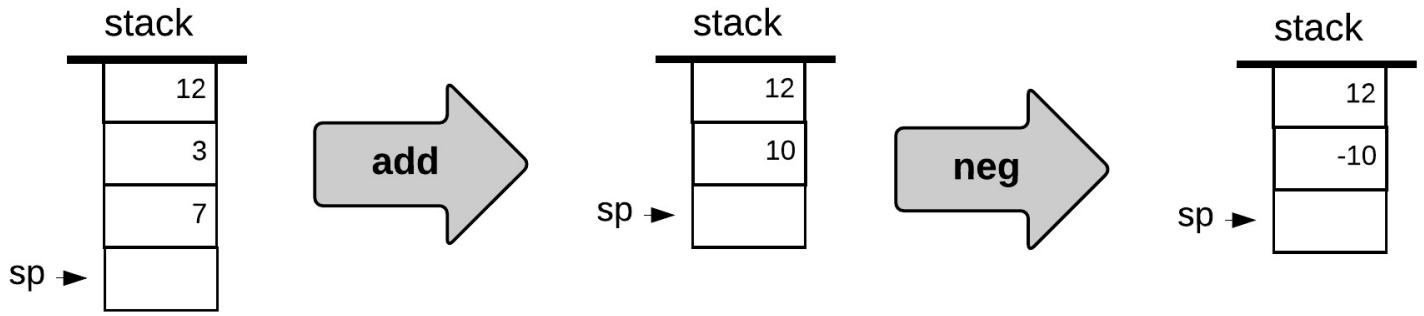
Stack operations:

- `push`: add an element at the stack's top
- `pop`: remove the top element

Stack

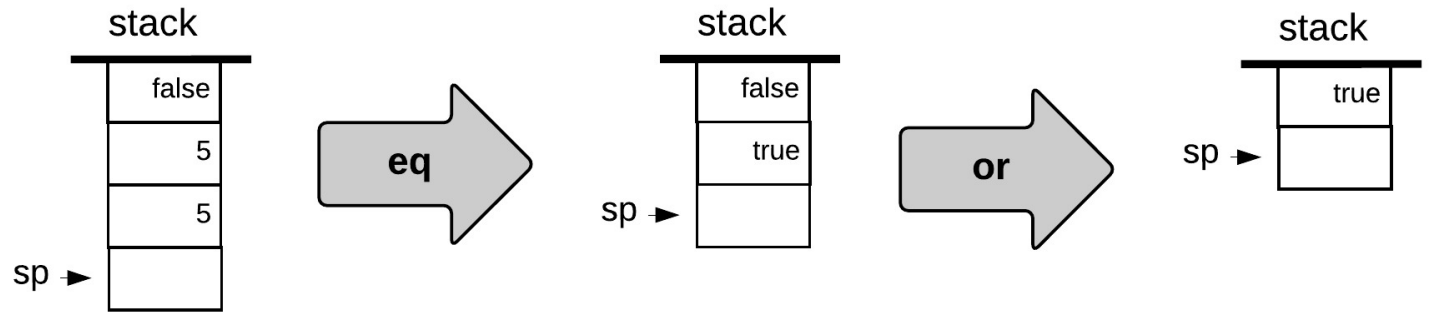


Stack arithmetic



Applying a function *f* on the stack:

- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.



Stack arithmetic (big picture)

high-level

```
x = 17 + 19
```

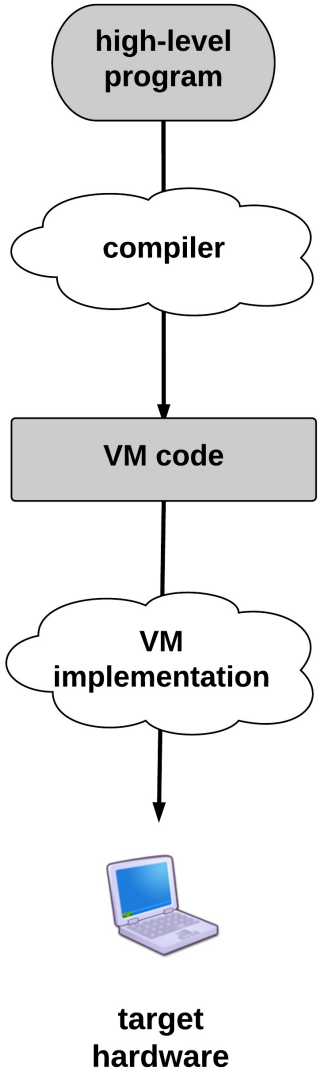
lower-level

```
push 17
push 19
add
pop x
```

Abstraction / implementation

- The high-level language is an abstraction;
- It can be implemented by a stack machine.
- The stack machine is also an abstraction;
- It can implemented by... Stay tuned.

The stack machine model



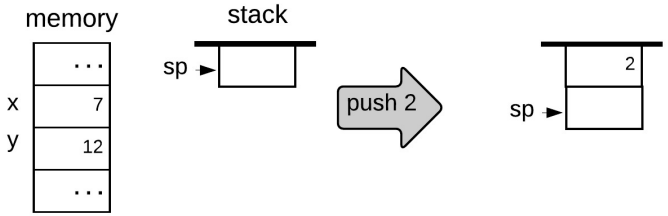
Stack machine, manipulated by:

- Arithmetic / logical commands
- Memory segment commands
- Branching commands
- Function commands

Arithmetic commands

VM code

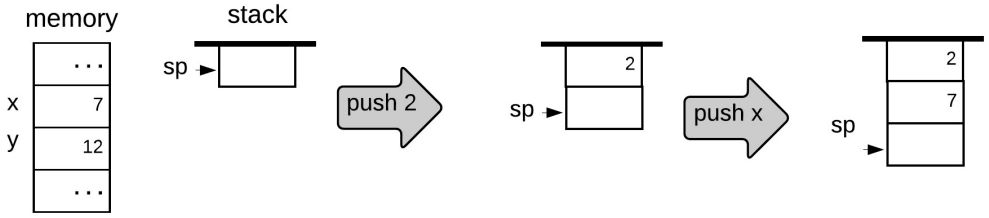
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

VM code

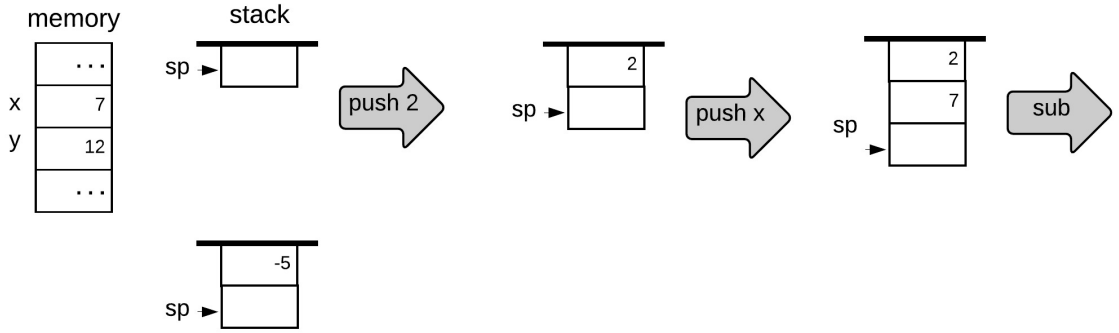
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

VM code

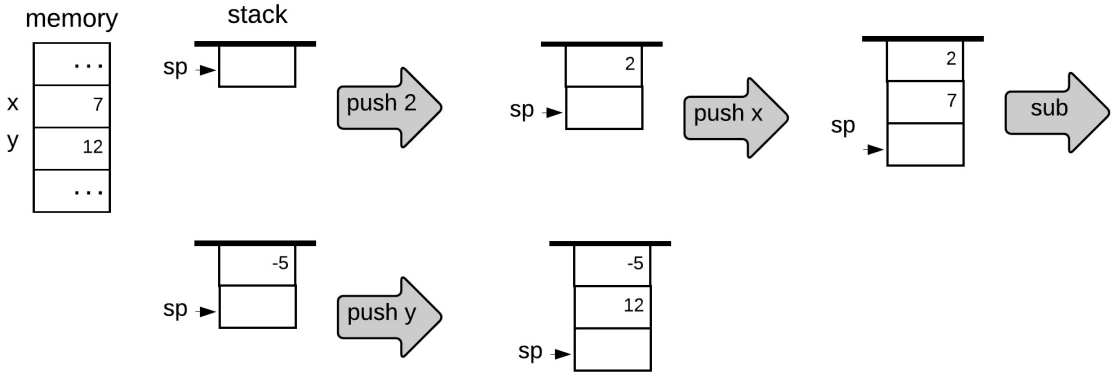
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

VM code

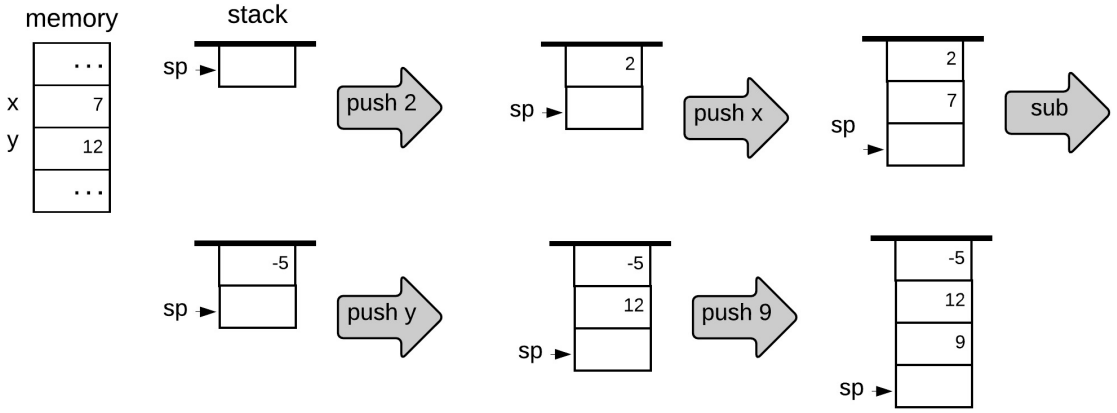
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

VM code

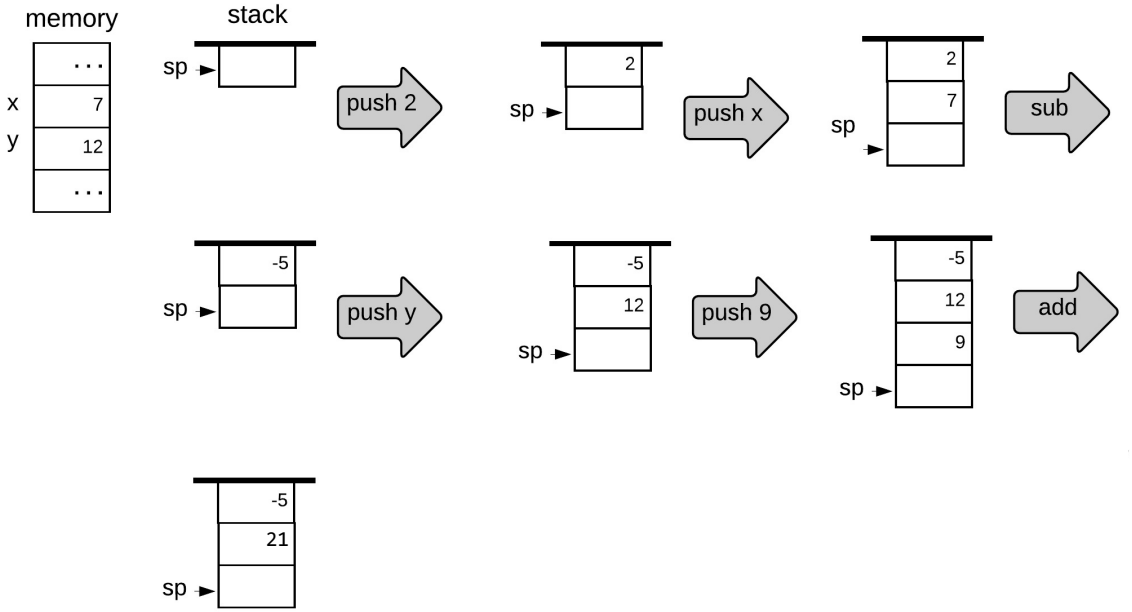
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

VM code

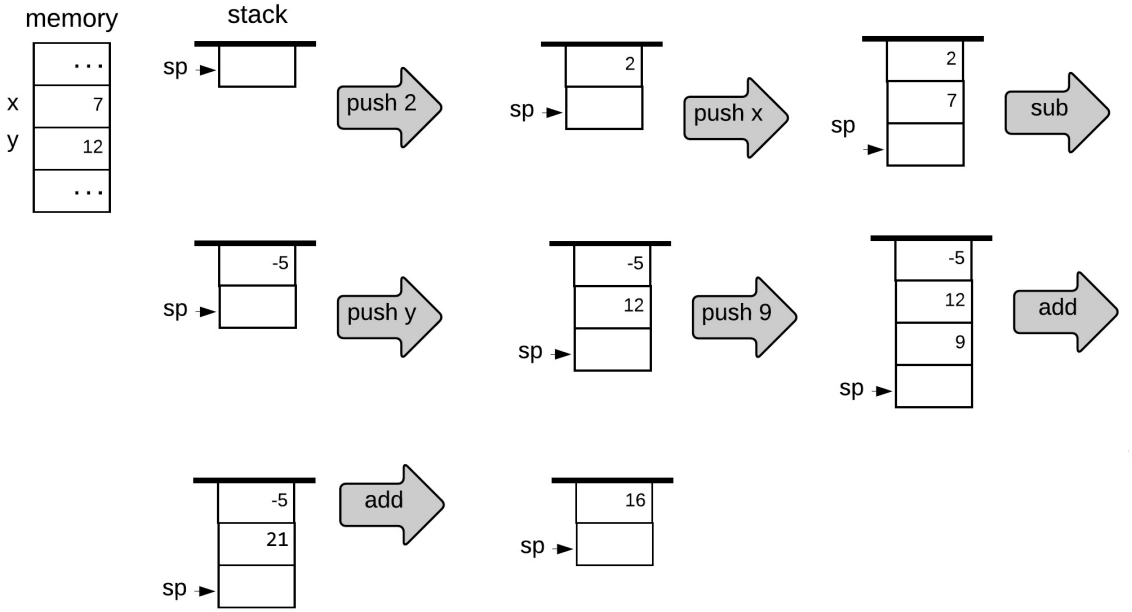
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

VM code

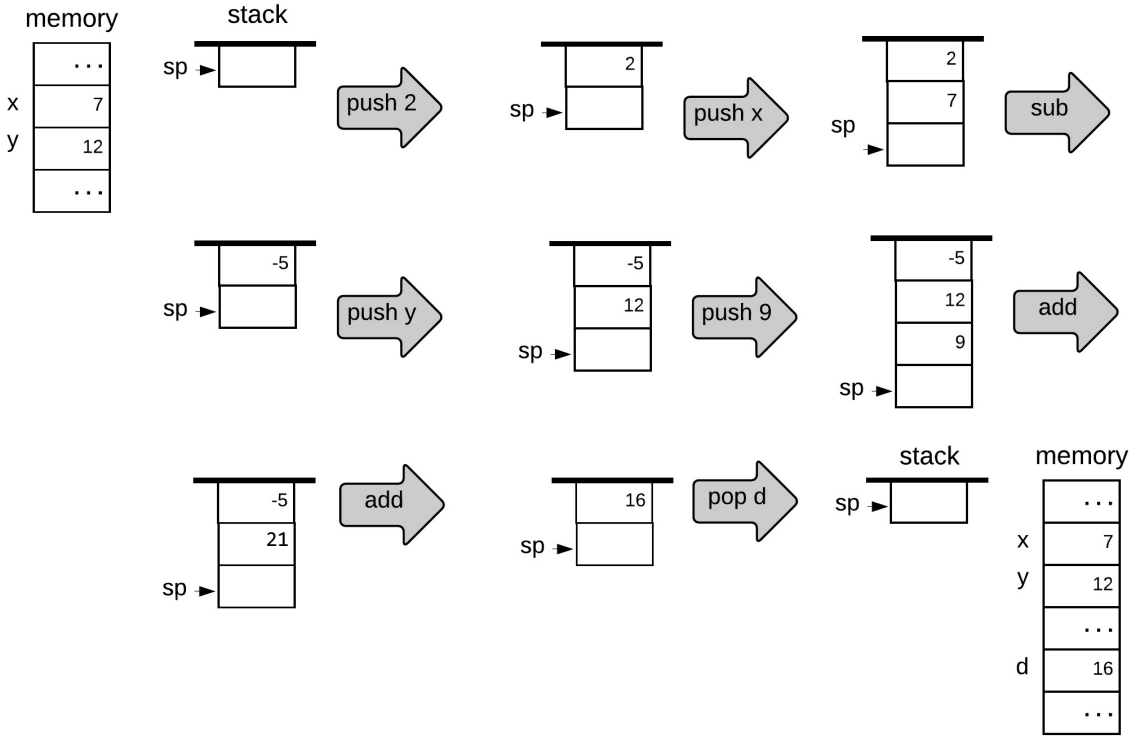
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

VM code

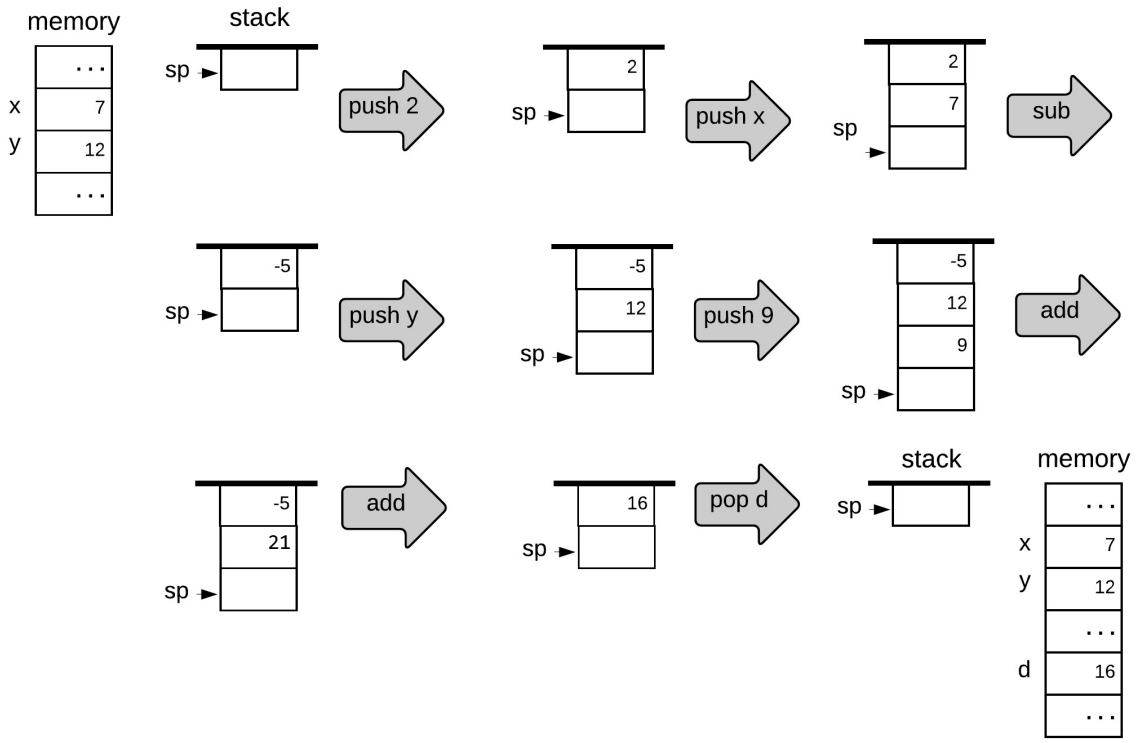
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

VM code

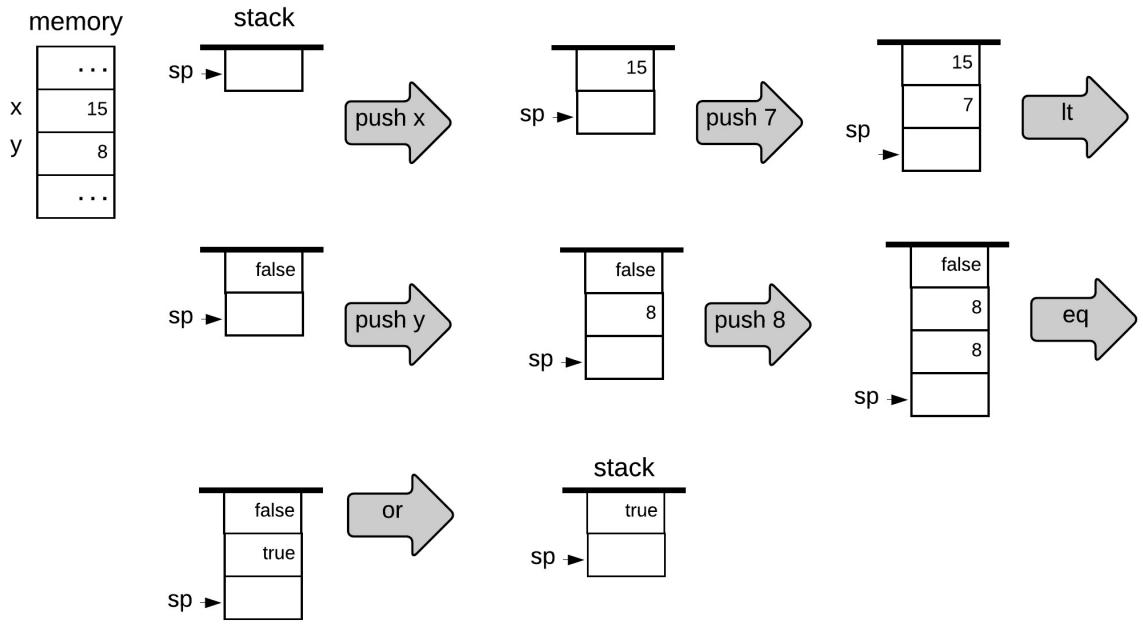
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Logical commands

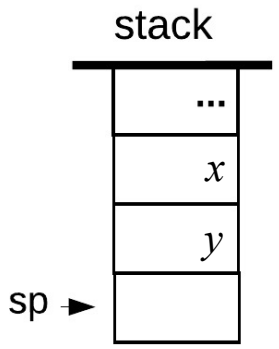
VM code

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



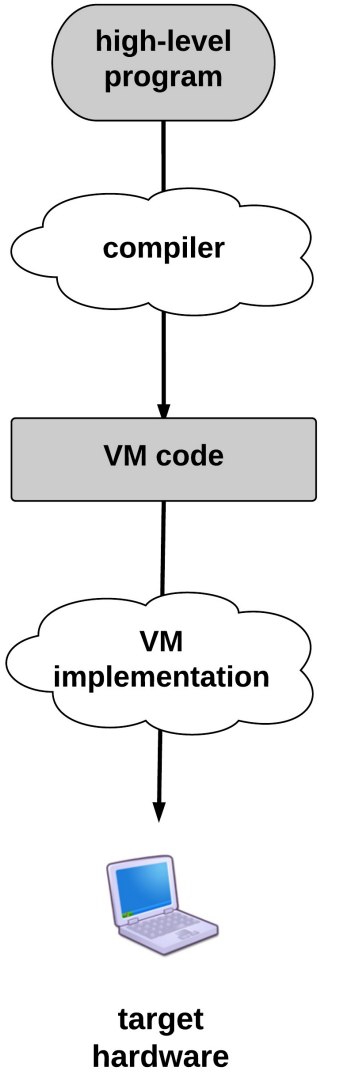
Arithmetic / Logical commands

Command	Return value	Return value
add	$x + y$	integer
sub	x - y	integer
neg	-y	integer
eq	x==0	boolean
gt	x > y	boolean
lt	x < y	boolean
and	x and y	boolean
or	x or y	boolean
not	not x	boolean



Observation: Any arithmetic or logical expression can be expressed and evaluated by applying some sequence of the above operations on a stack.

The stack machine model



Stack machine, manipulated by:

- Arithmetic / logical commands ✓
- Memory segment commands
- Branching commands
- Function commands

Virtual machine: lecture plan

Overview
- ✓ The road ahead
- ✓ Program compilation

VM abstraction
- ✓ the stack
- • memory segments

VM implementation
- • the stack
- • memory segments

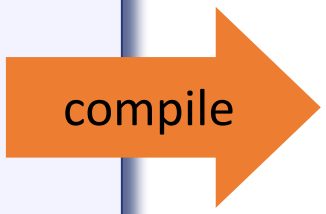
VM implementation platforms
- • VM emulator
- • VM translator

The VM translator
- • Proposed implementation
- • Building it (project 7)

The big picture

Source code (Jack)

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



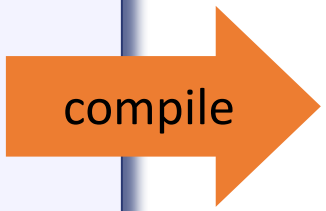
Compiled VM code

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

Variable kinds

Source code (Jack)

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



Compiled VM code

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

Variable kinds

- Argument variables
- Local variables
- Static variables

(More kinds later)

Variable kinds and memory segments

Source code (Jack)

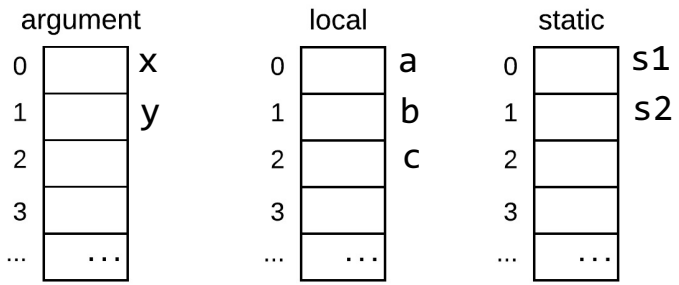
```
class Foo {
   static int s1, s2;
   function int bar (int x, int y) {
      var int a, b, c;
      ...
      let c = s1 + y;
      ...
   }
}
```

compile

Compiled VM code

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

Virtual memory segments:



Variable kinds and memory segments

Source code (Jack)

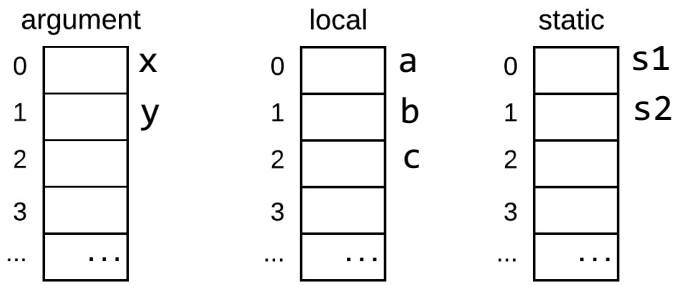
```
class Foo {
   static int s1, s2;
   function int bar (int x, int y) {
      var int a, b, c;
      ...
      let c = s1 + y;
      ...
   }
}
```

compile

Compiled VM code

```
...
...
...
...
push static 0
push y
add
pop c
...
...
```

Virtual memory segments:



Variable kinds and memory segments

Source code (Jack)

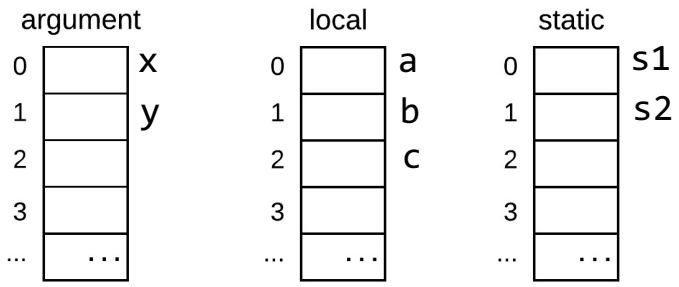
```
class Foo {
   static int s1, s2;
   function int bar (int x, int y) {
      var int a, b, c;
      ...
      let c = s1 + y;
      ...
   }
}
```

compile

Compiled VM code

```
...
...
...
...
push static 0
push argument 1
add
pop c
...
...
```

Virtual memory segments:



Variable kinds and memory segments

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;
      ...
   }
}
```

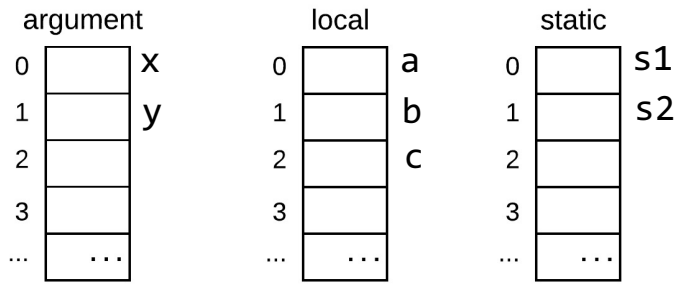
compile

Compiled VM code

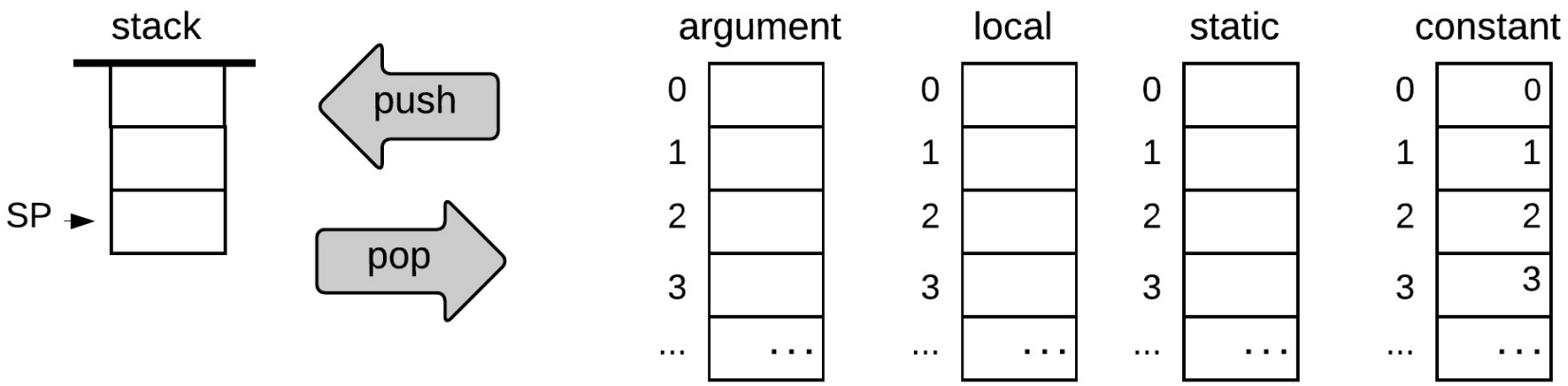
```
...
...
...
...
push static 0
push argument 1
add
pop local 2
...
...
```

Following compilation, all the symbolic references are replaced with references to virtual memory segments

Virtual memory segments:



Memory segments

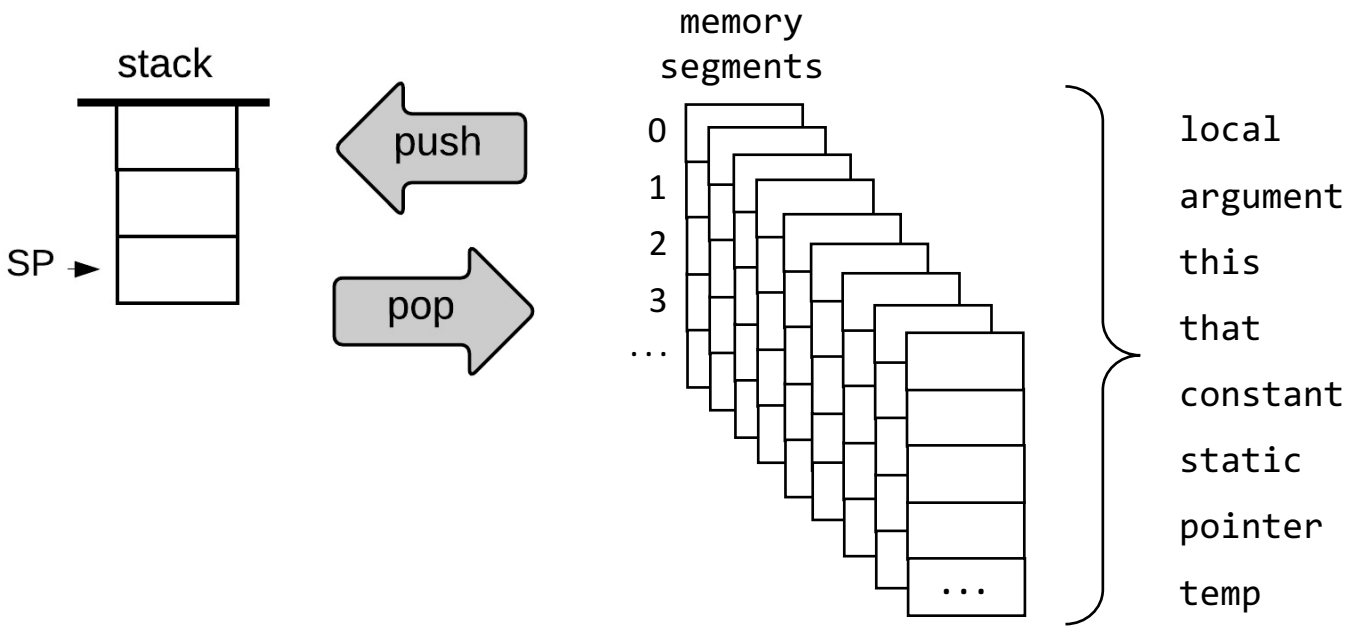


Syntax: `push` / `pop` *segment i*

Examples:

- `push constant 17`
- `pop local 2`
- `pop static 5`
- `push argument 3`
- `...`

Memory segments



Syntax: `push` *segment i*

Where *segment* is: `argument`, `local`, `static`, `constant`, `this`, `that`, `pointer`, or `temp`

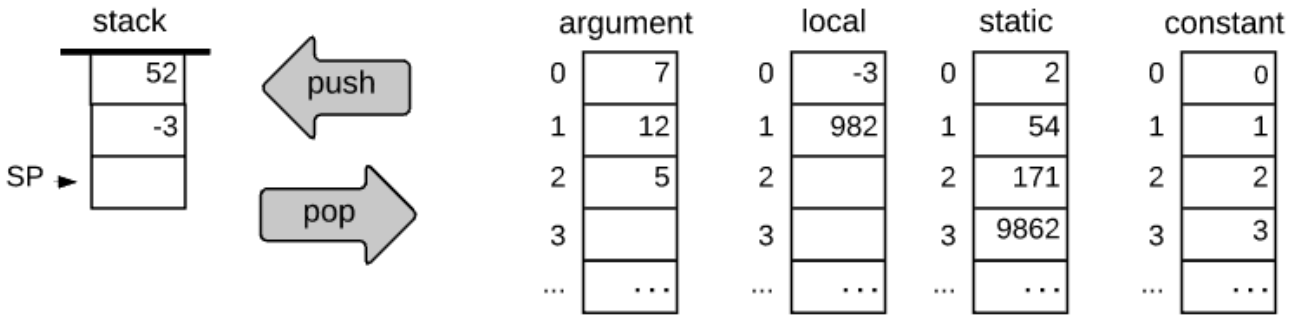
and *i* is a non-negative integer.

Syntax: `pop` *segment i*

Where *segment* is: `argument`, `local`, `static`, `this`, `that`, `pointer`, or `temp`

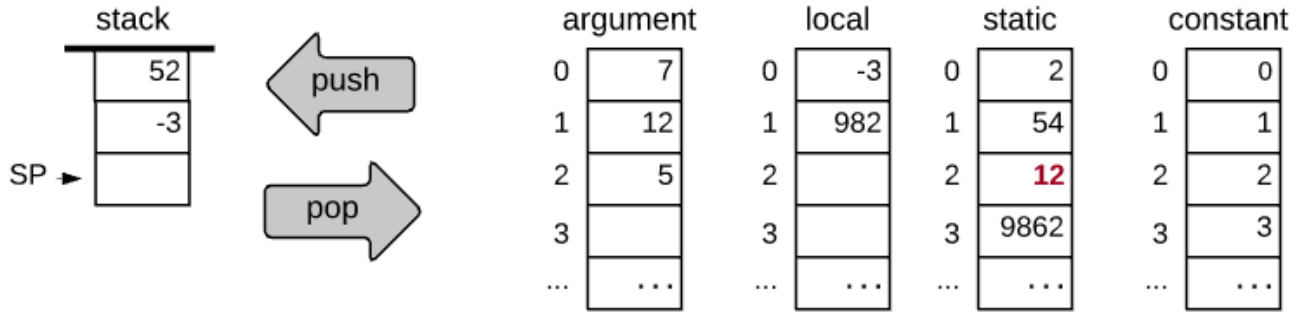
and *i* is a non-negative integer.

Quiz: write the missing code

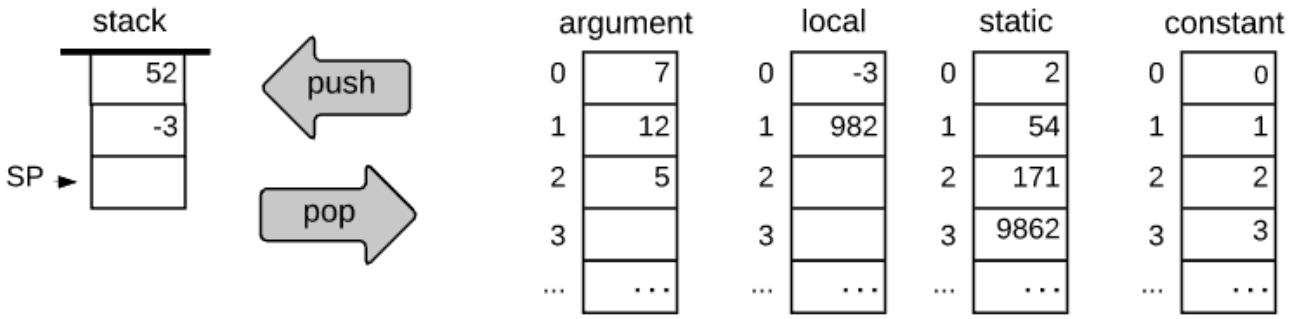


let `static 2 = argument 1`

?

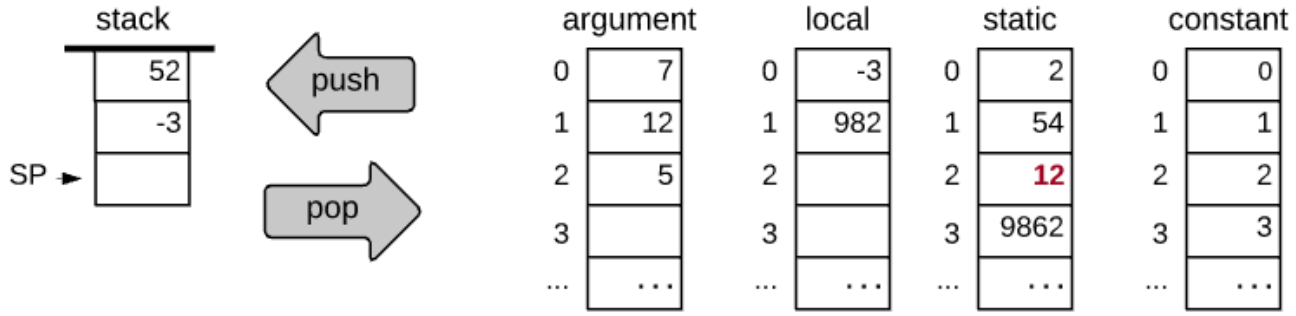


Quiz: write the missing code

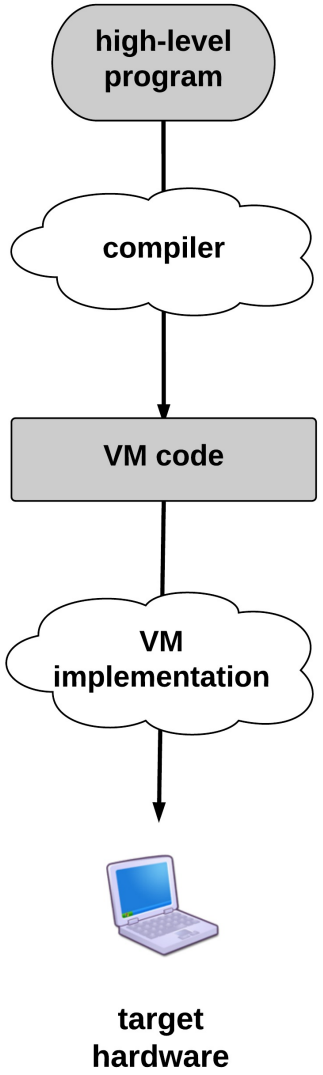


let `static 2 = argument 1`

```
push argument 1
pop static 2
```



The stack machine model



VM language:

- Arithmetic / logical commands ✓
- Memory segment commands ✓
- Branching commands
- Function commands

Virtual machine: lecture plan

Overview
- ✓ The road ahead
- ✓ Program compilation

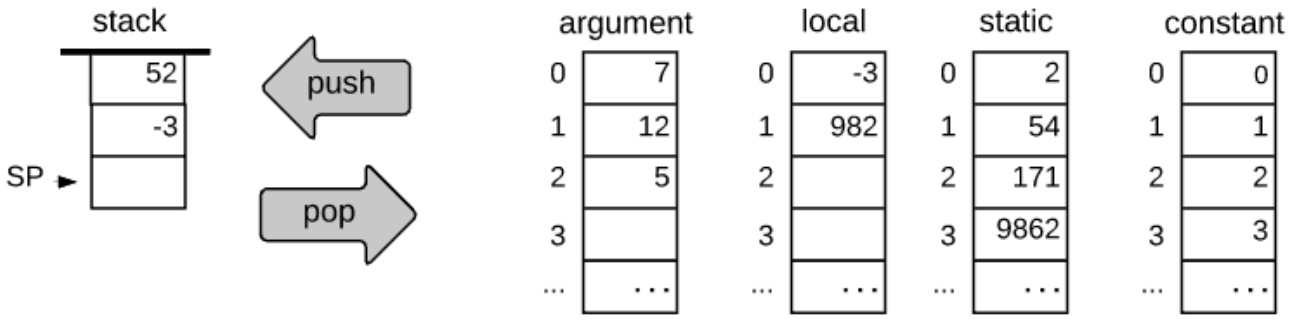
VM abstraction
- ✓ the stack
- ✓ memory segments

VM implementation
- the stack
- memory segments

VM implementation platforms
- VM emulator
- VM translator

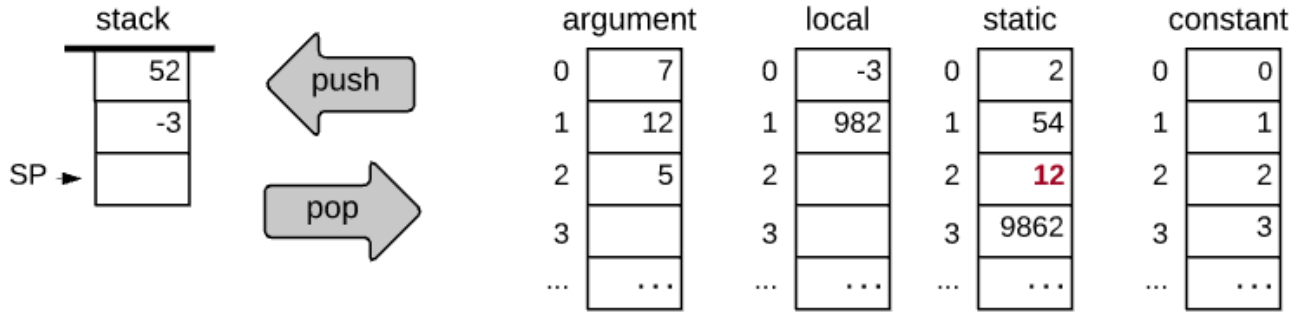
The VM translator
- Proposed implementation
- Building it (project 7)

The VM abstraction



let `static 2 = argument 1`

```
push argument 1
pop static 2
```



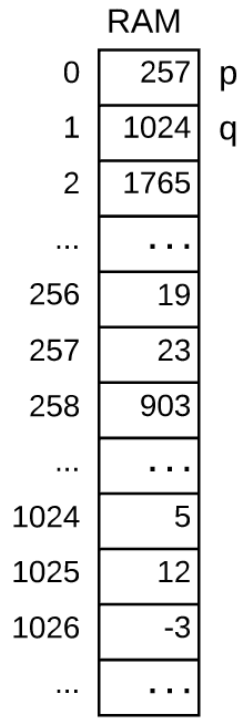
Pointer manipulation

Pseudo assembly code

```
D = *p          // D becomes 23
```

In Hack:

```
@p
A=M
D=M
```



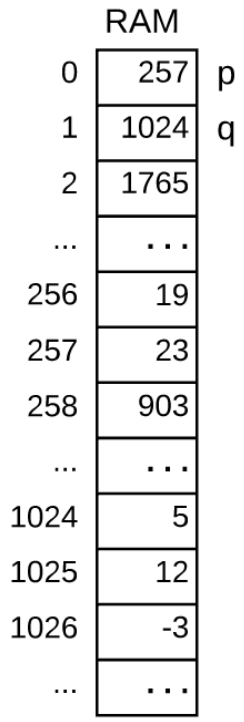
Notation:

`*p` // the memory location that `p` points at

Pointer manipulation

Pseudo assembly code

```
D = *p          // D becomes 23

p--             // RAM[0] becomes 256
D = *p          // D becomes 19

*q = 9          // RAM[1024] becomes 9
q++             // RAM[1] becomes 1025
```



Notation:

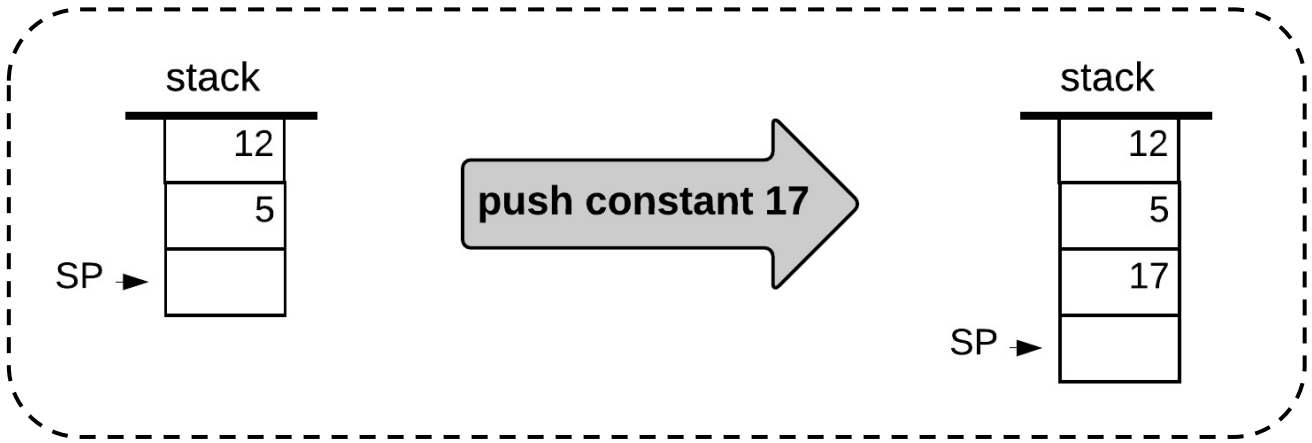
`*p` // the memory location that `p` points at

`x++` // increment: `x = x + 1`

`x--` // decrement: `x = x - 1`

Stack implementation

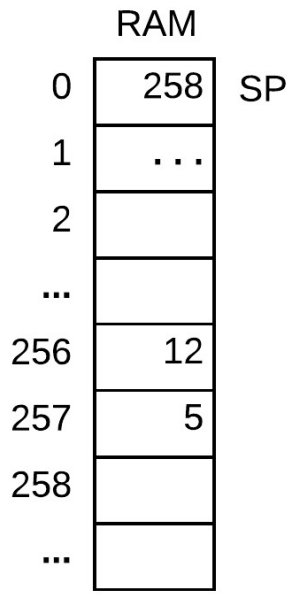
Abstraction:



Implementation:

Assumptions:

- `SP` stored in `RAM[0]`
- Stack base addr = `256`

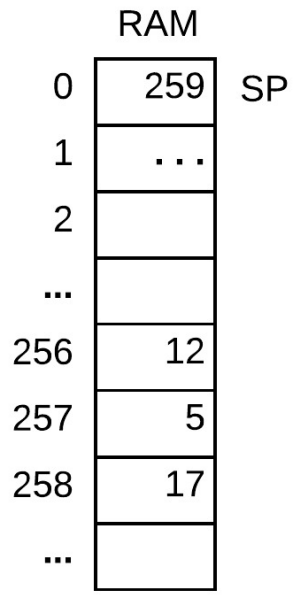


Logic:

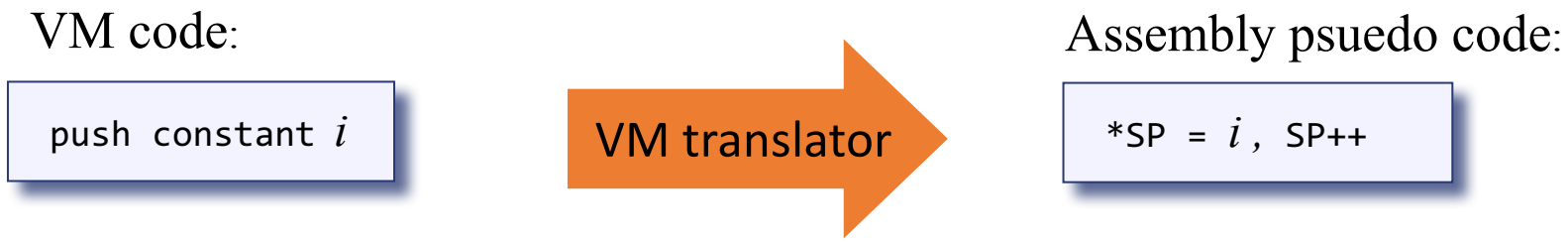
```
*SP = 17
SP++
```

Hack assembly:

```
@17   // D=17
D=A
@SP   // *SP=D
A=M
M=D
@SP   // SP++
M=M+1
```



Stack implementation



VM Translator

- A program that translates VM commands into lower-level commands of some host platform (like the Hack computer)
- Each VM command generates one or more low-level commands
- The low-level commands end up realizing the stack and the memory segments on the host platform.

Virtual machine: lecture plan

Overview
- ✓ The road ahead
- ✓ Program compilation

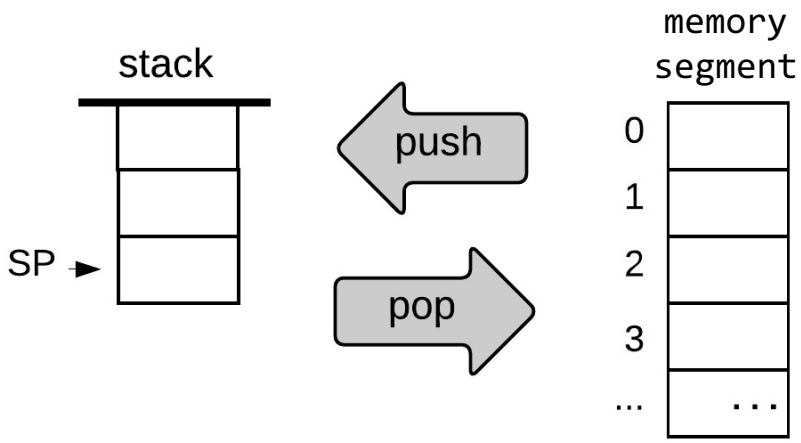
VM abstraction
- ✓ the stack
- ✓ memory segments

VM implementation
- ✓ the stack
- • memory segments

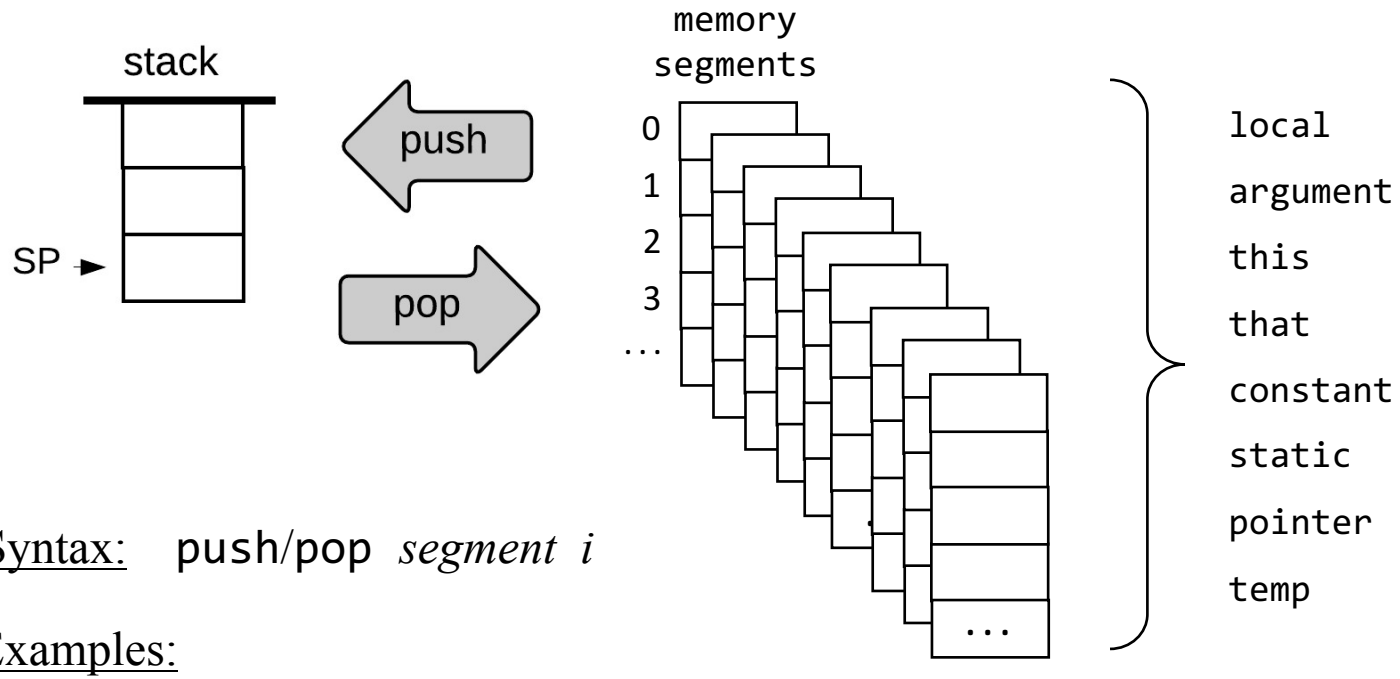
VM implementation platforms
- • VM emulator
- • VM translator

The VM translator
- • Proposed implementation
- • Building it (project 7)

Memory segments (abstraction)



Memory segments (abstraction)

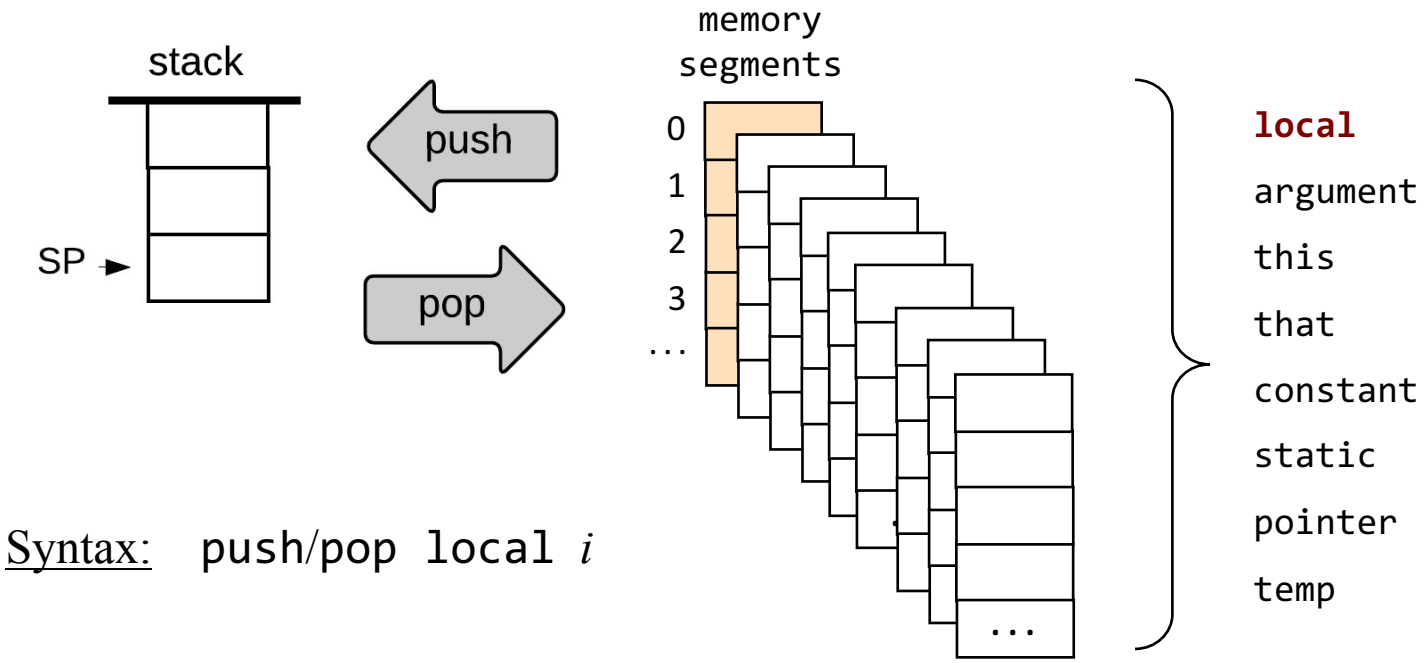


Syntax: `push/pop` *segment i*

Examples:

- `push constant 17`
- `pop local 2`
- `pop static 5`
- `push argument 3`
- `pop this 2`
- `...`

Implementing push/pop local *i*



Syntax: push/pop local *i*

Why do we need a local segment?

When the compiler translates high-level code into VM code...

- high-level operations on *local variables* are translated into VM operations on the entries of the segment local
- We now turn to describe how the local segment can be realized on the host platform.

Implementing `pop local i`

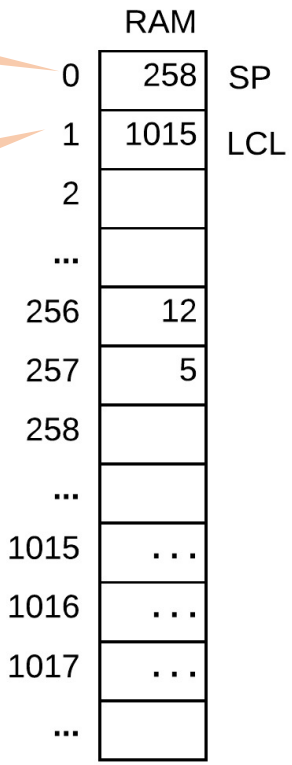
Abstraction:

stack pointer

base address of the `local` segment

Implementation:

the `local` segment is stored somewhere in the RAM

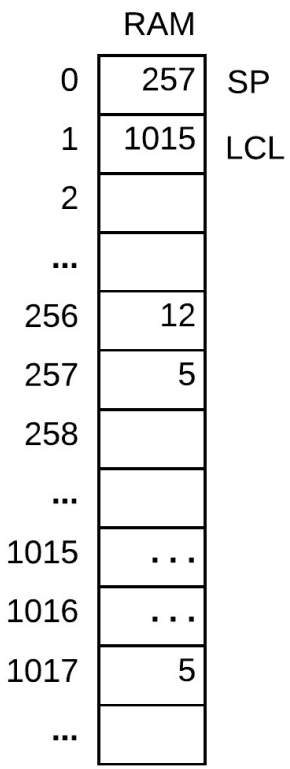


Implementation:

```
addr=LCL+i, SP--, *addr=*SP
```

Hack assembly:

You write it!



Implementing `push/pop local i`

VM code:

```
pop local i
```

```
push local i
```

VM Translator

Assembly pseudo code:

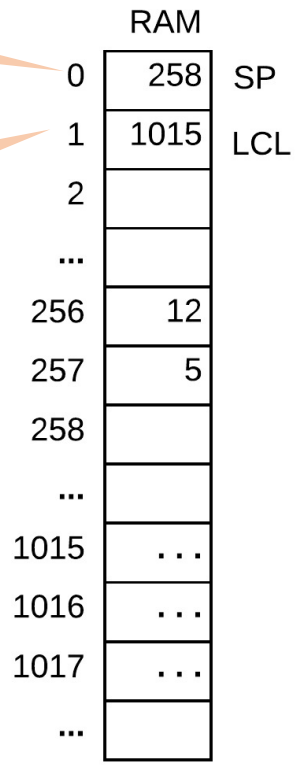
```
addr = LCL+ i, SP--, *addr = *SP
```

```
addr = LCL+ i, *SP = *addr, SP++
```

stack pointer

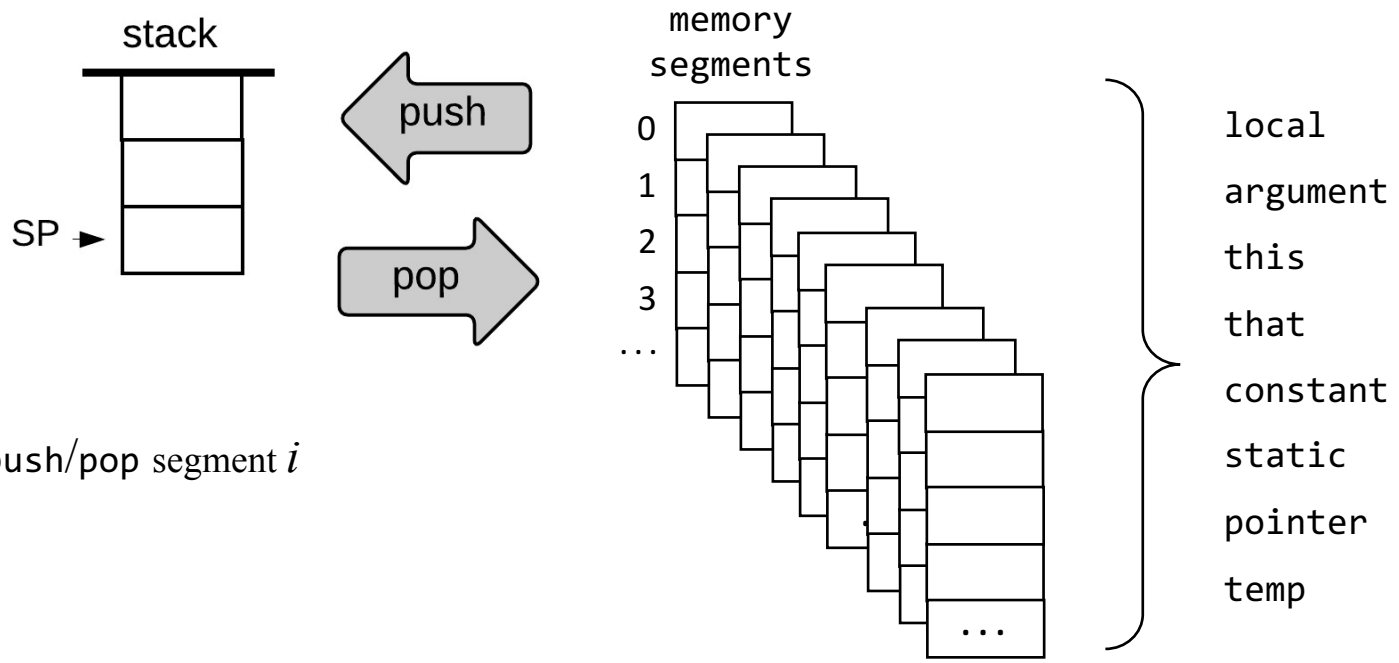
base address of the `local` segment

the `local` segment is stored somewhere in the RAM



Implementation:

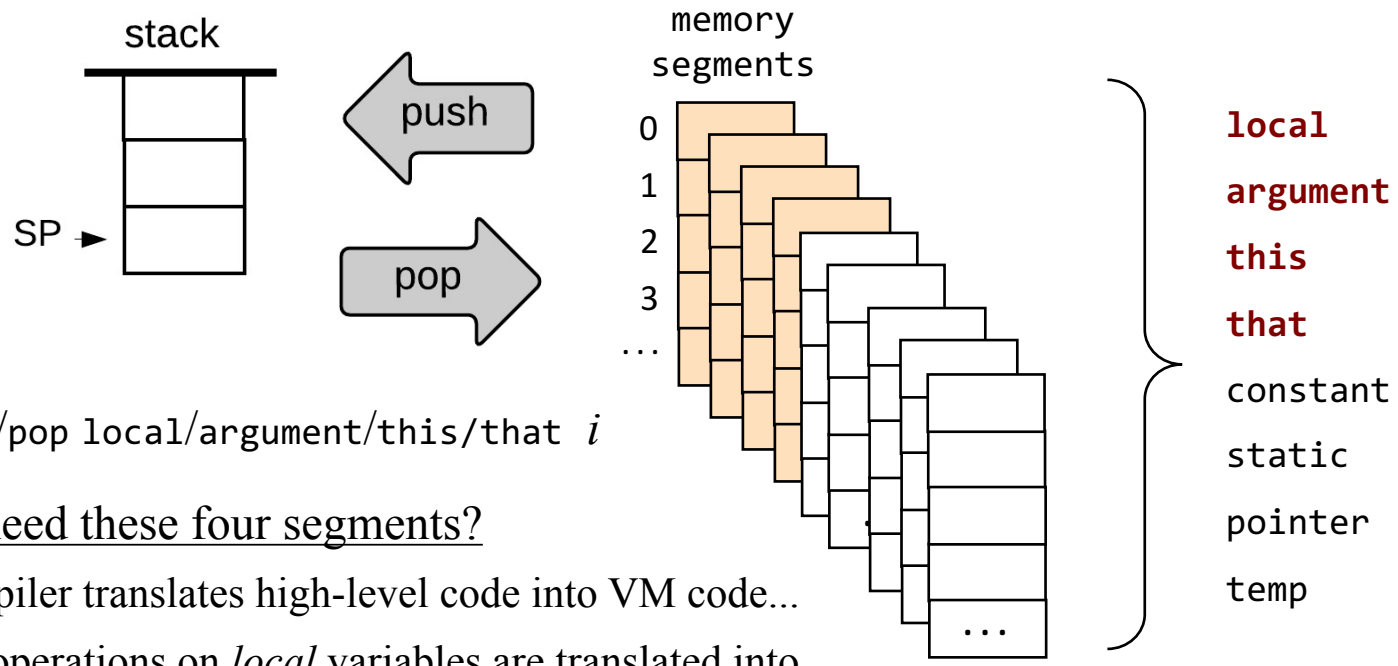
Memory segments



Syntax: `push`/`pop` segment *i*

- We know how to implement `push`/`pop` `local` *i*
- We now turn to implementing `push`/`pop` operations on the segments `argument`, `this`, and `that`.

Implementing `push/pop local/argument/this/that i`



Syntax: `push/pop local/argument/this/that i`

Why do we need these four segments?

When the compiler translates high-level code into VM code...

- high-level operations on *local* variables are translated into VM operations on the entries of the segment `local`
- high-level operations on *argument* variables are translated into VM operations on the entries of the segment `argument`
- high-level operations on the *field* variables of the current object are translated into VM operations on the entries of the segment `this`
- high-level operations on *array entries* are translated into VM operations on the entries of the segment `that`

We now turn to describe how these four segments can be realized on the host platform.

Implementing `push/pop local/argument/this/that i`

VM code:

`push segment i`

`pop segment i`

VM translator

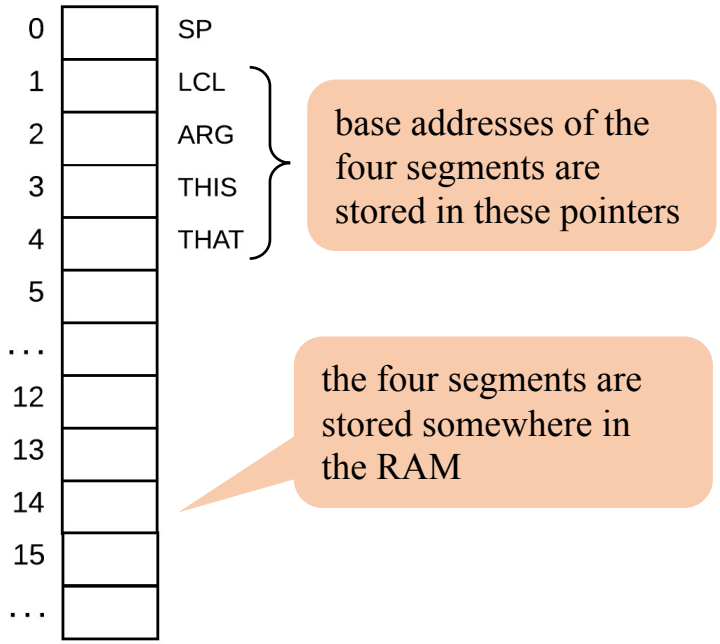
Assembly pseudo code:

`addr = segmentPointer + i, *SP = *addr, SP++`

`addr = segmentPointer + i, SP--, *addr = *SP`

segment = {`local`, `argument`, `this`, `that`}

Host RAM



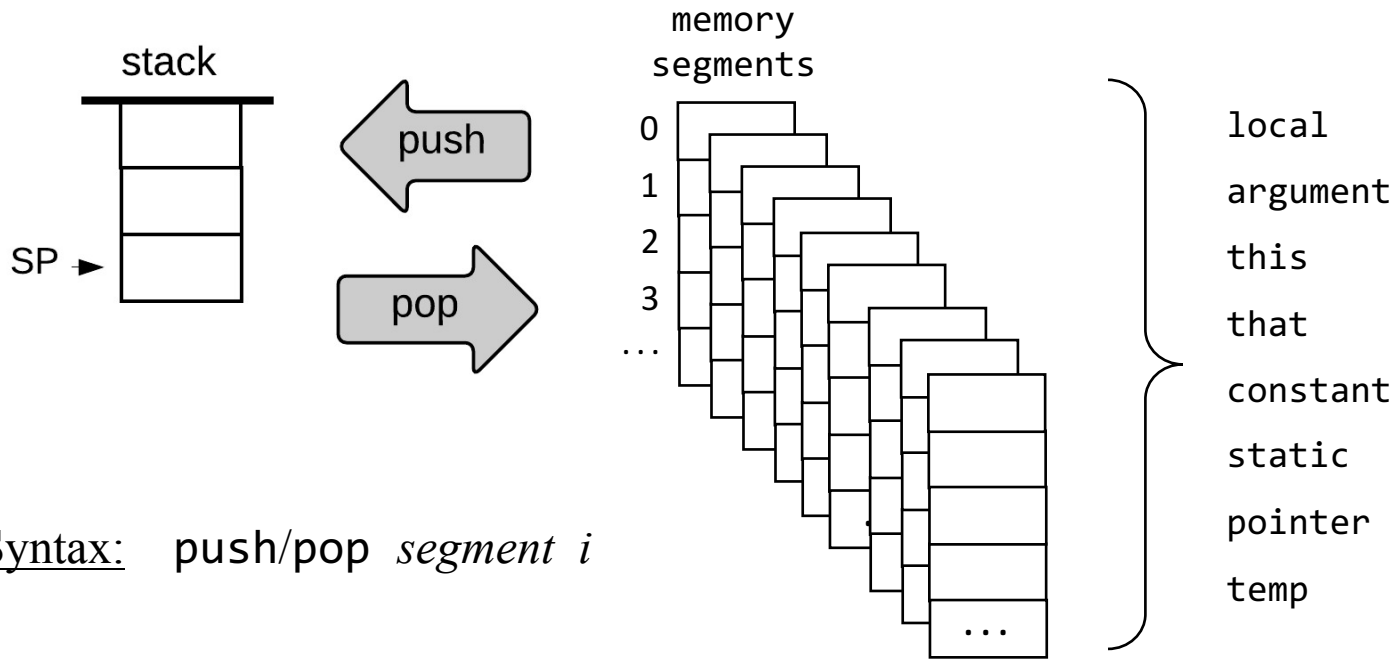
base addresses of the four segments are stored in these pointers

the four segments are stored somewhere in the RAM

- `push/pop local i`
- `push/pop argument i`
- `push/pop this i`
- `push/pop that i`

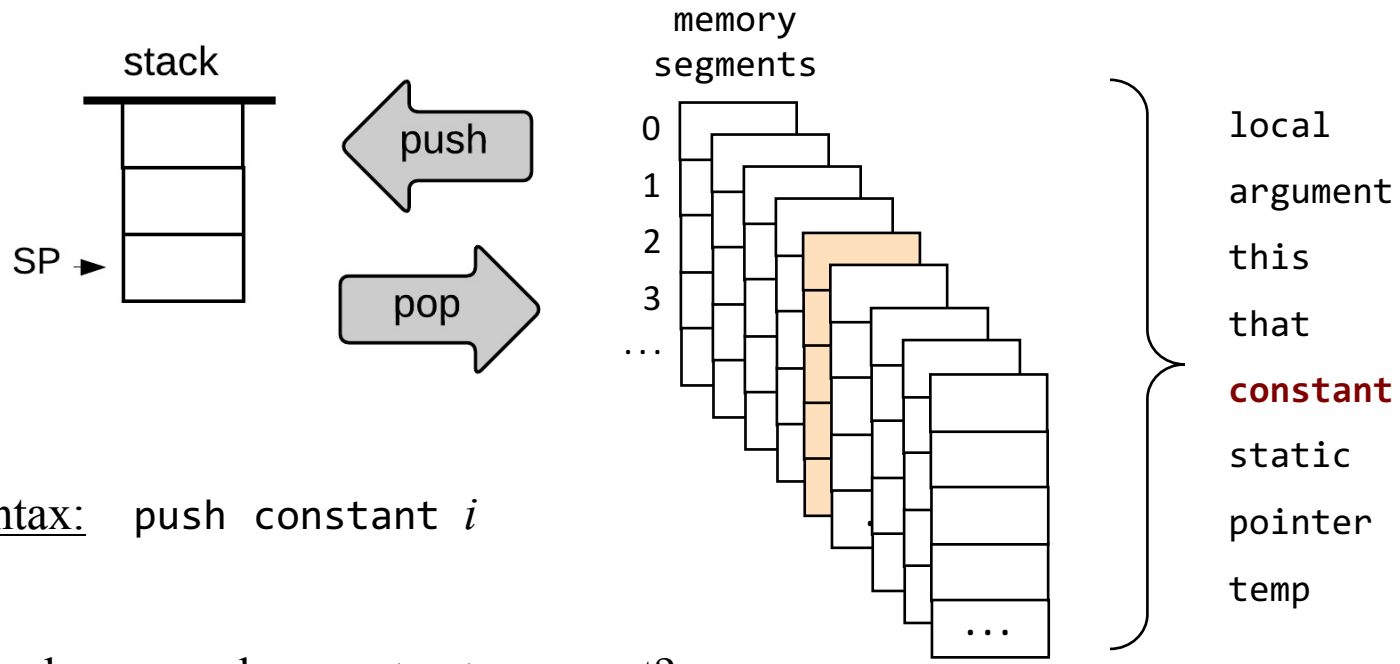
are implemented precisely the same way

Memory segments



Syntax: `push/pop` *segment i*

Implementing `push constant i`



Syntax: `push constant i`

Why do we need a `constant` segment?

Because we need to represent constants somehow in the VM level.

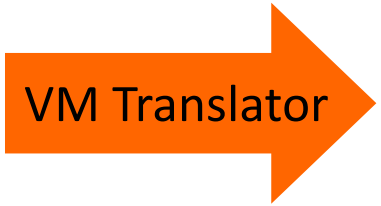
When the compiler translates high-level code into VM code...

- high-level operations on the *constant i* are translated into VM operations on the segment entry `constant` *i*
- This synthactical convention will make sense when we write the compiler.

Implementing `push constant` *i*

VM code:

```
push constant i
```



Assembly psuedo code:

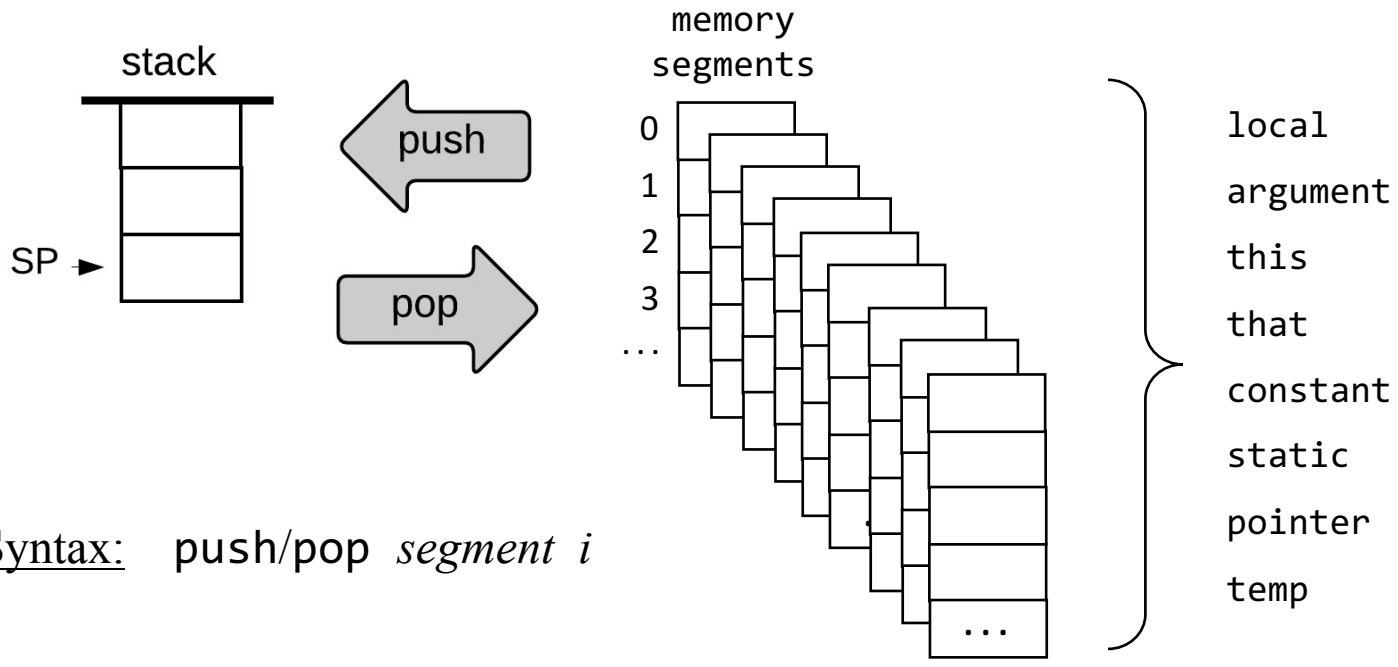
```
*SP = i, SP++
```

(no `pop constant` operation)

Implementation:

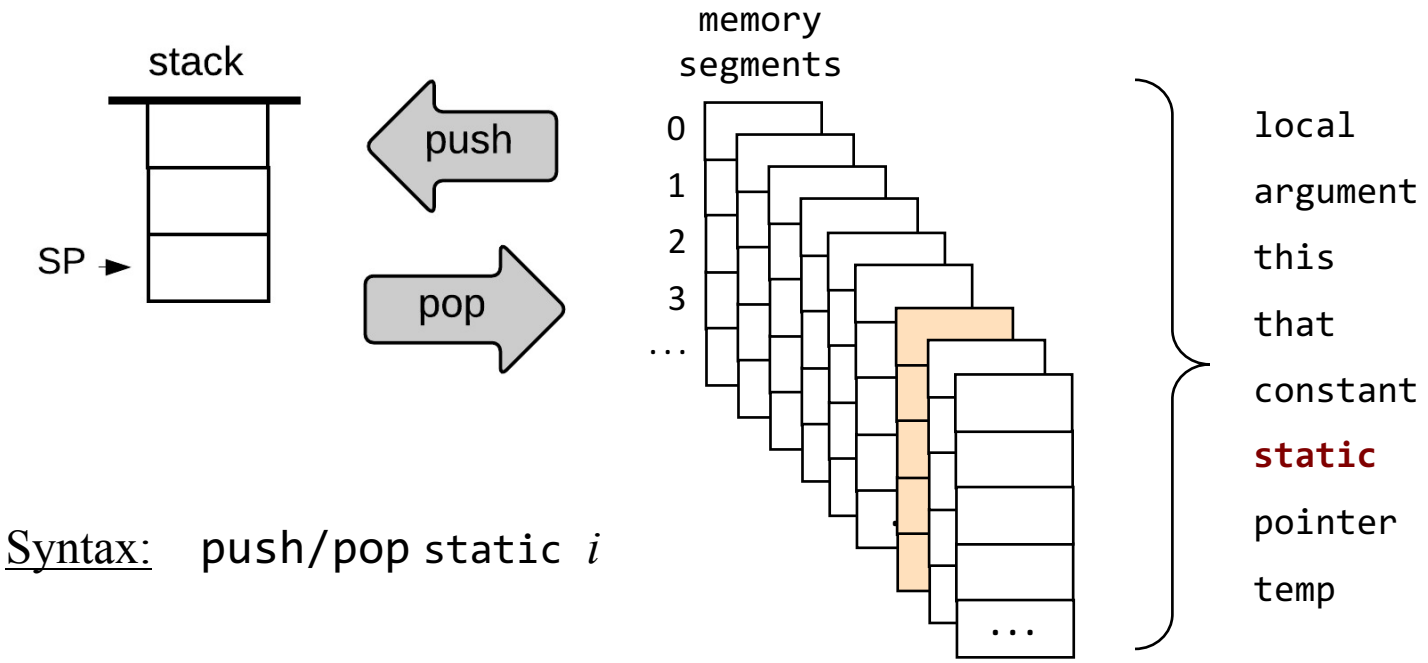
Supplies the specified constant

Memory segments



Syntax: `push/pop` *segment i*

Implementing push/pop static *i*



Syntax: push/pop static *i*

Why do we need a static segment?

When translating high-level code into VM code, the compiler...

- high-level operations on *static* variables are translates into VM operations on entried of the segment static
- We now turn to discuss how the static segment is realized on the host platform.

Implementing push/pop static *i*

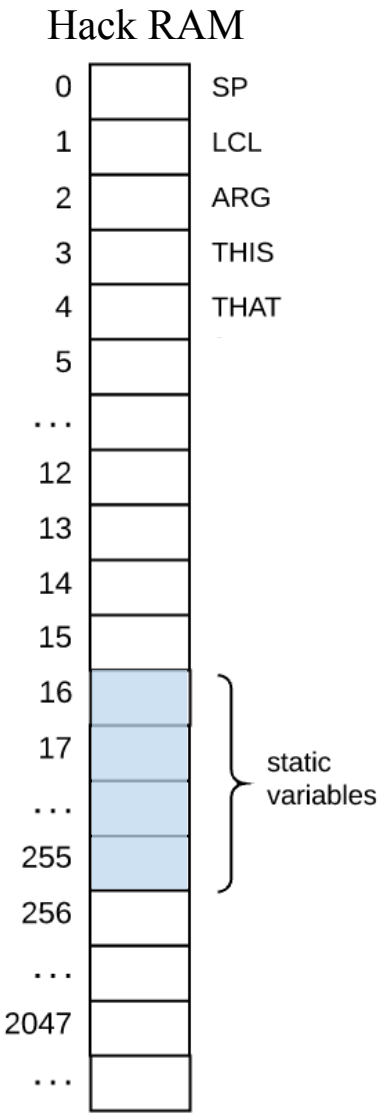
VM code:

```
// File Foo.vm
...
pop static 5
...
pop static 2
...
```

VM translator

Generated assembly code:

```
...
// D = stack.pop (code omitted)
@Foo.5
M=D
...
// D = stack.pop (code omitted)
@Foo.2
M=D
...
```



The challenge:

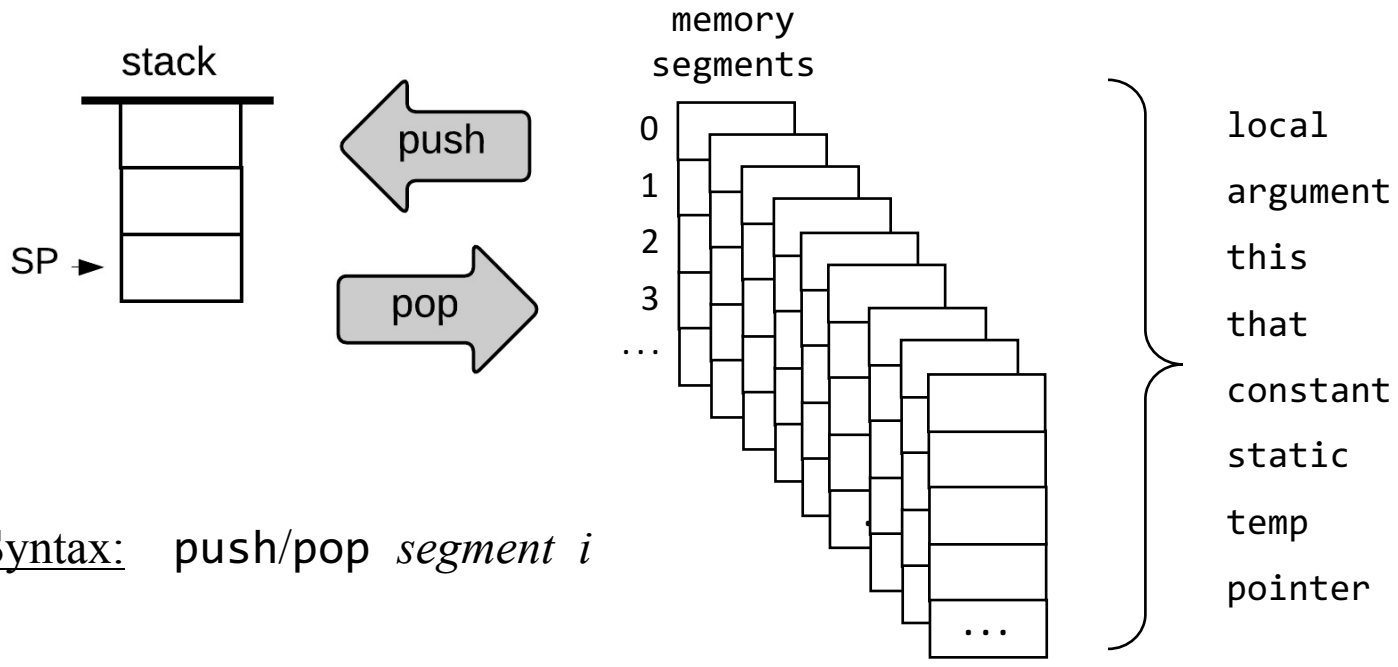
static variables should be seen by all the methods in a program

Solution:

Store them in some "global space":

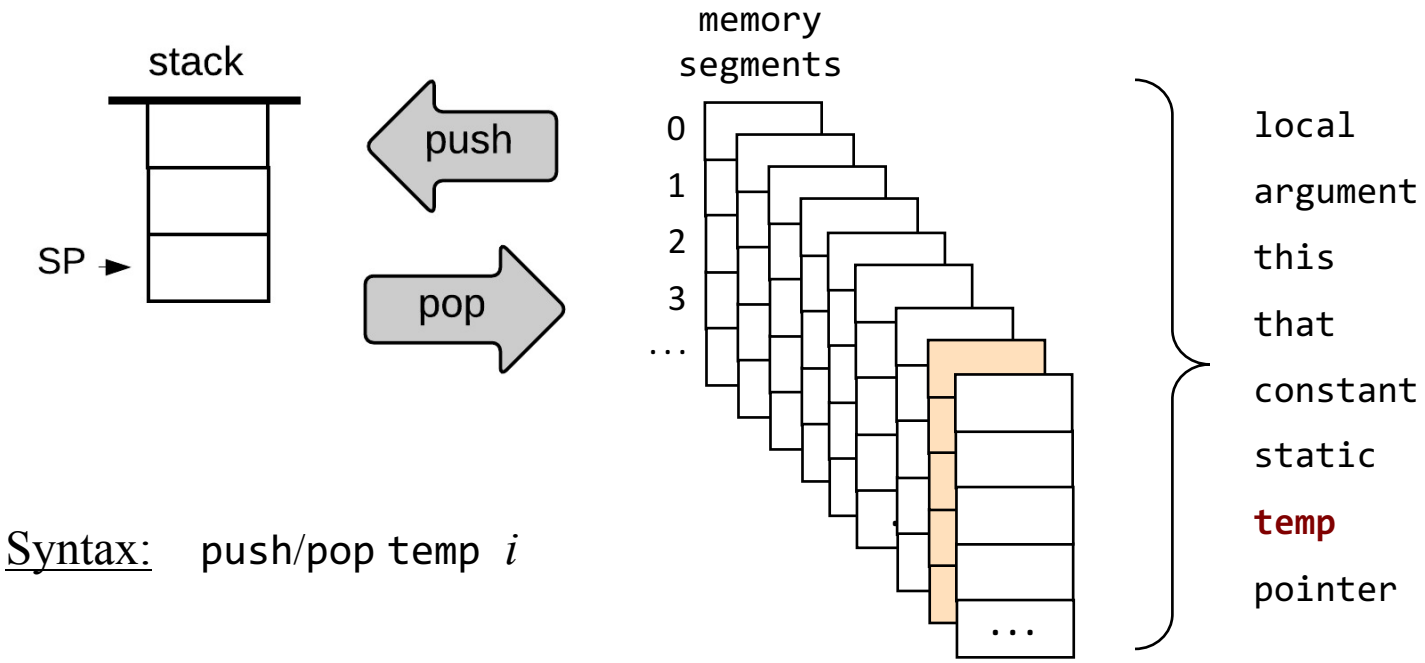
- Have the VM translator translate each VM reference `static` *i* (in file `Foo.vm`) into an assembly reference `Foo.`*i*
- Following assembly, the Hack assembler will map these references onto `RAM[16]`, `RAM[17]`, ..., `RAM[255]`
- Therefore, the entries of the `static` segment will end up being mapped onto `RAM[16]`, `RAM[17]`, ..., `RAM[255]`, in the order in which they appear in the program.

Memory segments



Syntax: `push/pop` *segment i*

Implementing `push/pop temp i`



Syntax: `push/pop temp` *i*

Why do we need the `temp` segment?

- So far, all the variable kinds that we discussed came from the source code
- Sometimes, the compiler needs to use some working variables of its own
- Our VM provides 8 such variables, stored in a segment named `temp`.

Implementing `push/pop temp i`

VM code:

`push temp i`

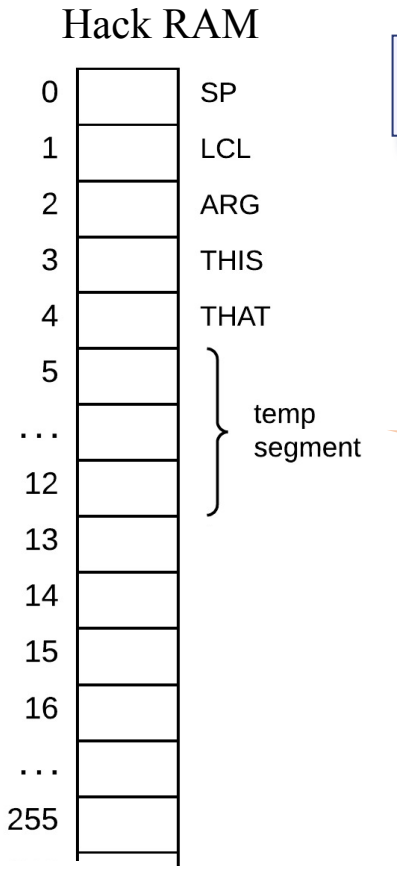
`pop temp i`

VM Translator

Assembly psuedo code:

`addr = 5 + i, *SP = *addr, SP++`

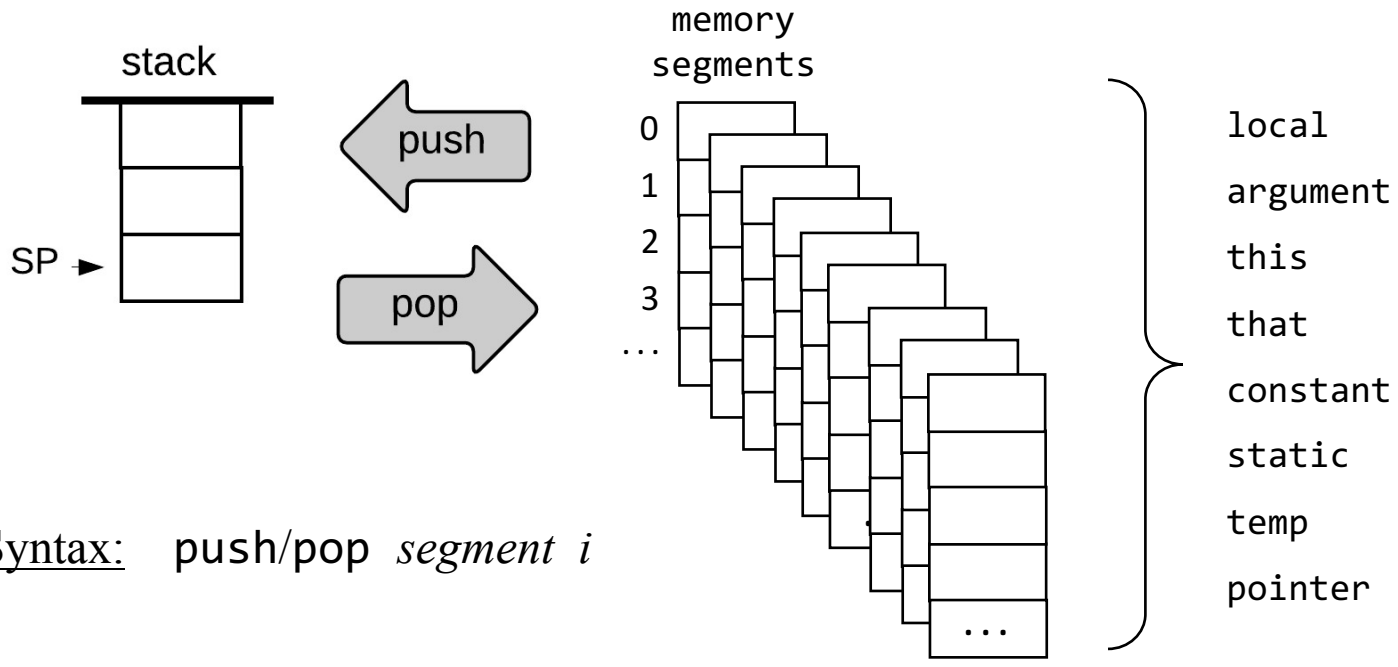
`addr = 5 + i, SP--, *addr = *SP`



A fixed, 8-place memory segment,

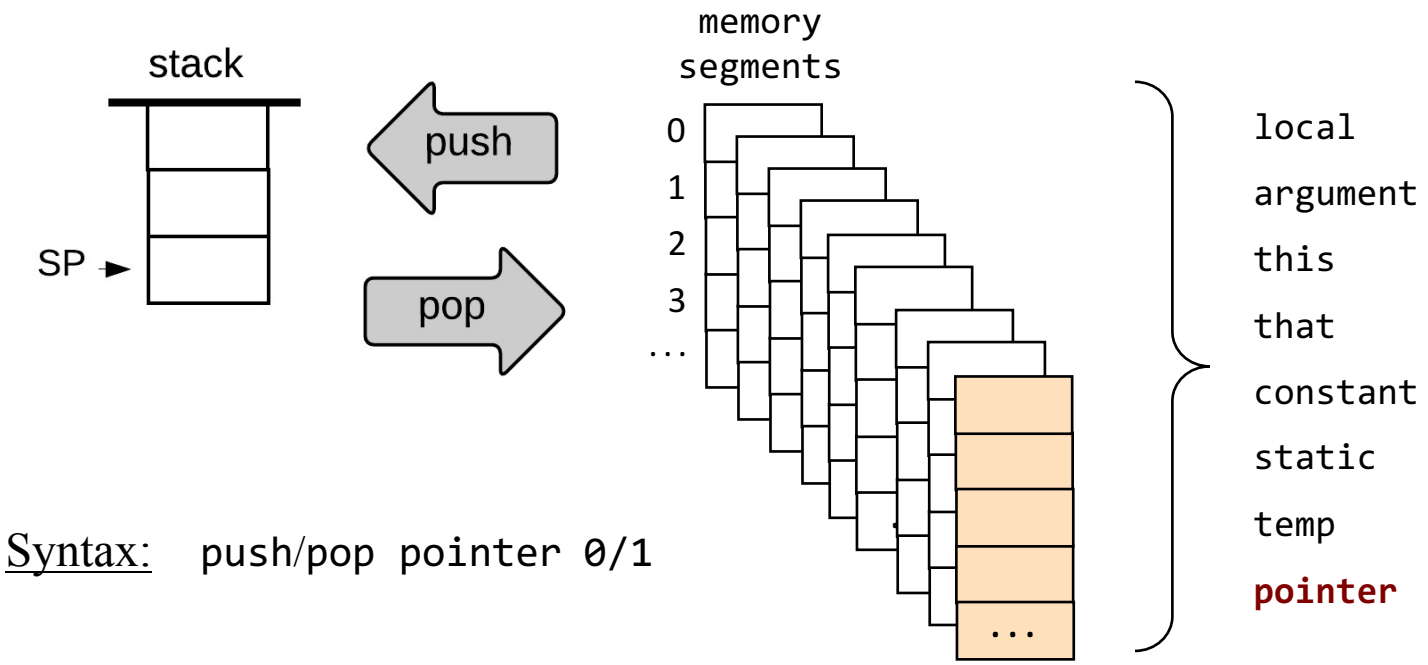
stored in RAM locations 5 to 12

Memory segments



Syntax: `push/pop` *segment i*

Implementing `push/pop pointer 0/1`



Syntax: `push/pop pointer 0/1`

Why do we need the `pointer` segment?

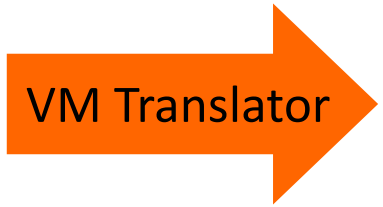
- We use it for storing the base addresses of the segments `this` and `that`
- The need for this will become clear when we'll write the compiler.

Implementing `push/pop pointer 0/1`

VM code:

```
push pointer 0/1
```

```
pop pointer 0/1
```



Assembly psuedo code:

```
*SP = THIS/THAT, SP++
```

```
SP--, THIS/THAT = *SP
```

A fixed, 2-place segment:

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

Implementation:

Supplies `THIS` or `THAT` // (the base addresses of `this` and `that`).

VM language

Arithmetic / Logical commands

`add`

`sub`

`neg`

`eq`

`get`

`lt`

`and`

`or`

`not`

Memory access commands

`pop` *segment i*

`push` *segment i*

Branching commands

`label` *label*

`goto` *label*

`if-goto` *label*

Function commands

`function` *functionName nVars*

`call` *functionName nArgs*

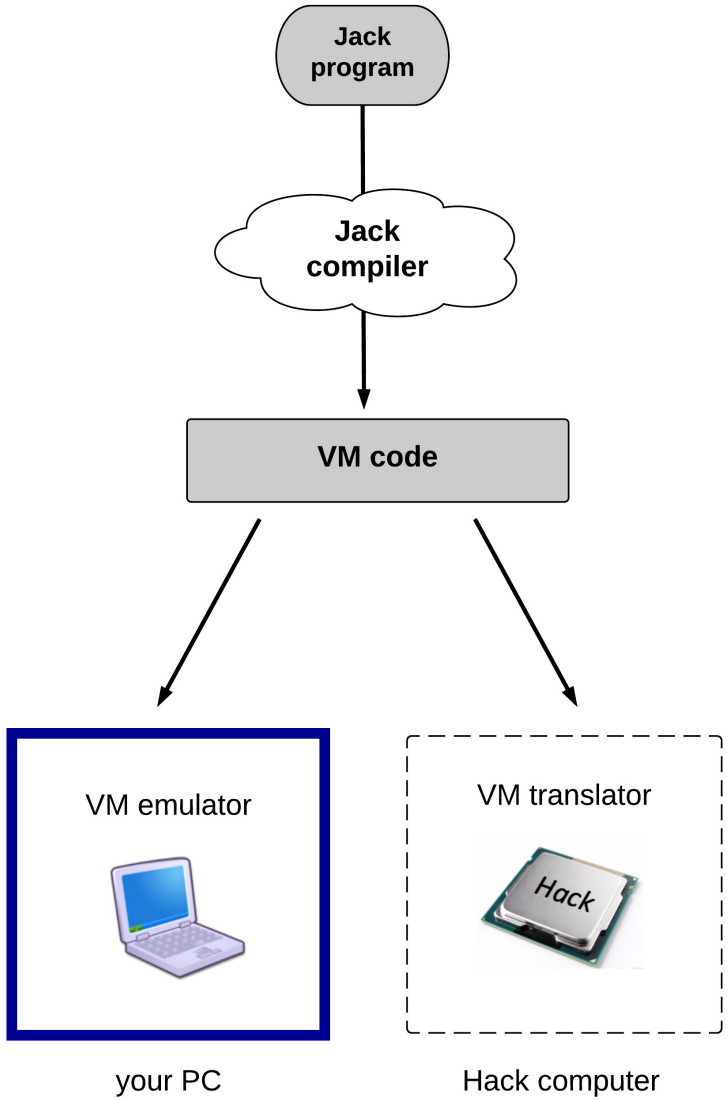
`return`

Virtual machine: lecture plan

- ✓ The road ahead
- ✓ Program compilation overview
- ✓ VM abstraction:
  - ✓ the stack
  - ✓ memory segments
- ✓ VM implementation:
  - ✓ the stack
  - ✓ memory segments

- The VM emulator
- VM implementation on the Hack platform
- The VM translator: proposed implementation
- Building the VM translator, Part I

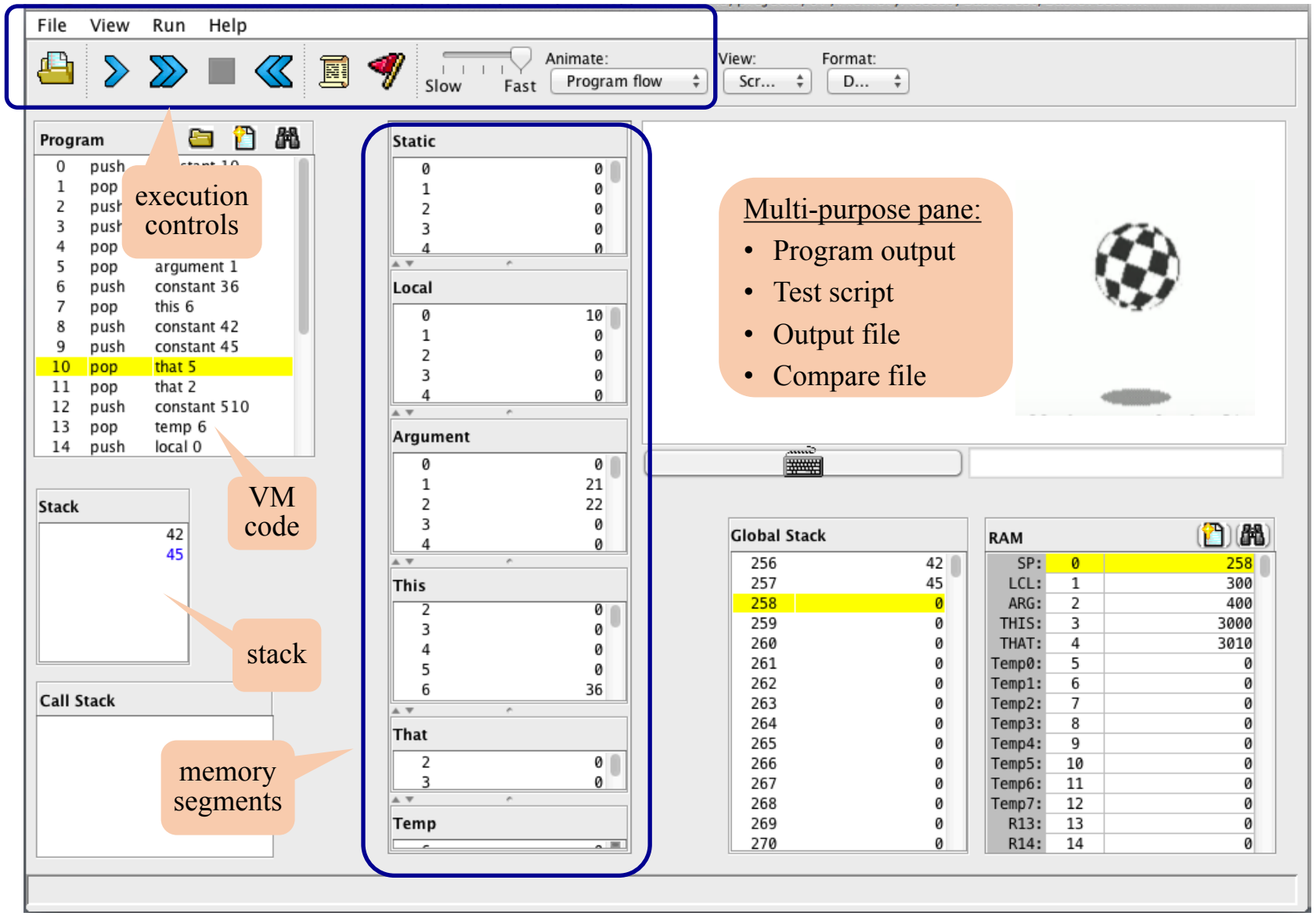
The VM Emulator



Typical uses of the VM emulator:

- Running (compiled) Jack programs
- Experimenting with VM commands
- Observing the VM internals (stack, memory segments)
- Observing how the VM is realized on the host platform.

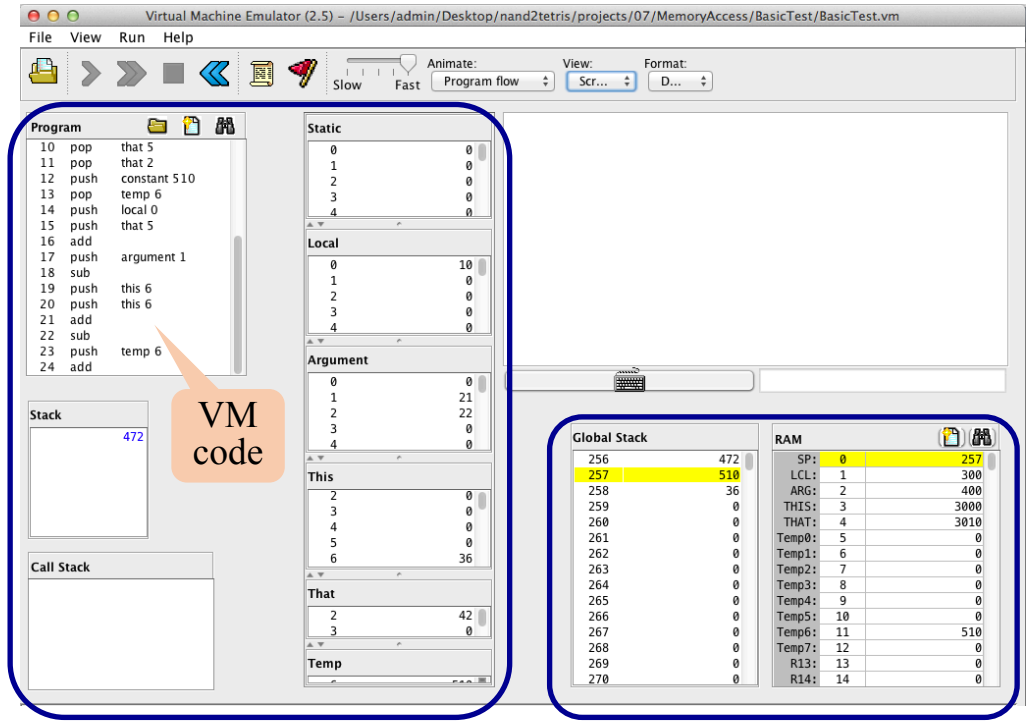
The VM Emulator



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:

(during the code's execution)

- Stack state
- Memory segments states

How the stack and memory segments are realized on the host platform

Test script

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,
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 ...




repeat 25 {        // BasicTest.vm has 25 instructions
  vmstep;
}

// Outputs some values, as specified by the output-list
// (stack base + selected values from the tested mem. segments)
output;
```

There's no need to delve into the code of test scripts

if (.out == .cmp)
  the test is successful
else
  error

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 |
```

Some missing elements

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,
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 ...

repeat 25 {        // BasicTest.vm has 25 instructions
  vmstep;
}

// Outputs some values, as specified by the output-list
// (stack base + selected values from the tested mem. segments)
output;
```

There's no need to delve into the code of test scripts

Some missing elements

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,
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 ...

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

repeat 25 {        // BasicTest.vm has 25 instructions
  vmstep;
}

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

There's no need to delve into the code of test scripts

Initialization is handled manually by the supplied test script

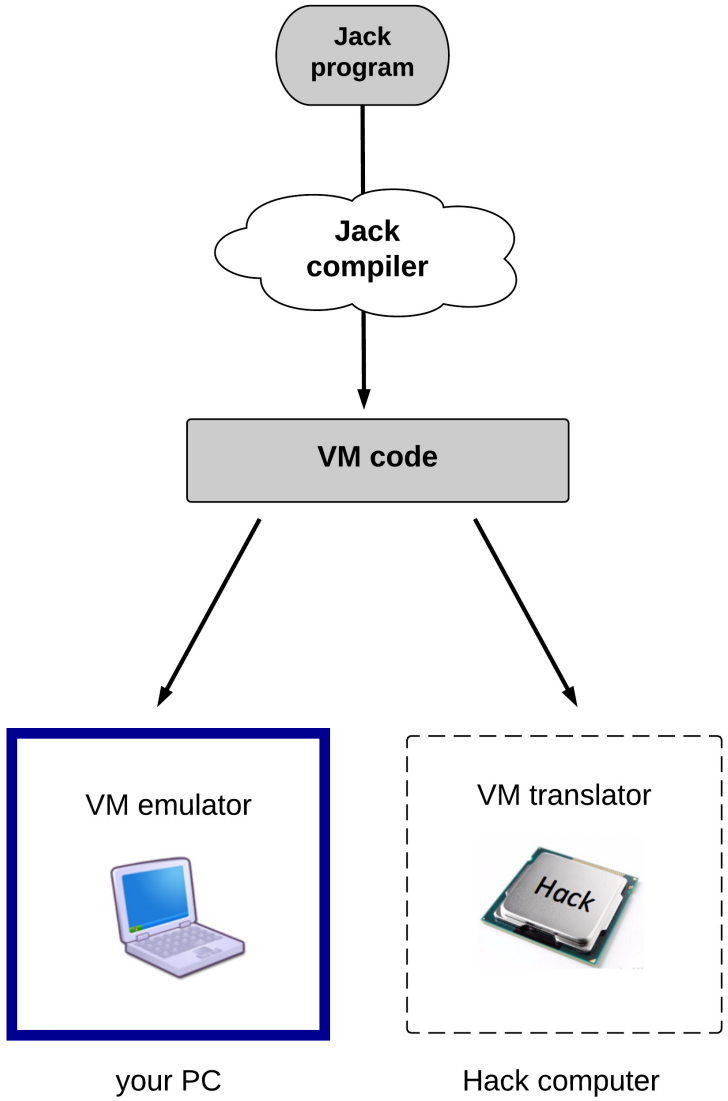
Some missing elements

- `function` / `return` "envelope"
- Initializing the stack and the memory segments on the host RAM

  (both will be added in project 8)

Demo

Recap: the VM Emulator



The VM emulator helps us:

- Run and test high-level programs
- Understand ...
  - the VM abstraction
  - the VM implementation

Virtual machine: lecture plan

- ✓ The road ahead
- ✓ Program compilation overview
- ✓ VM abstraction:
  - ✓ the stack
  - ✓ memory segments
- ✓ VM implementation:
  - ✓ the stack
  - ✓ memory segments
- ✓ The VM emulator
- • VM implementation on the Hack platform
- • The VM translator: proposed implementation
- • Building the VM translator, Part I

VM translator

VM code

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

Each VM command is translated into several assembly commands

Assembly code

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

In order to write a VM translator, we must be familiar with:

- the source language
- the target language
- the VM mapping on the target platform.

Source: VM language

Arithmetic / Logical commands

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

Memory access commands

`pop` *segment i*

`push` *segment i*

Branching commands

`label` *label*

`goto` *label*

`if-goto` *label*

Function commands

`function` *functionName nVars*

`call` *functionName nArgs*

`return`

Target: symbolic Hack code

A instruction: `@value`

where *value* is either a non-negative decimal constant or a symbol referring to such a constant

Semantics:

- sets the `A` register to *value*;
- makes `M` the `RAM` location whose address is *value*.
  (`M` stands for `RAM[A]`)

C instruction: `dest = comp ; jump` (*dest* and *jump* are optional)

where:

comp =

```
0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D&A, D|A,
          M,    !M,     -M,     M+1,     M-1, D+M, D-M, M-D, D&M, D|M
```

dest = `null, M, D, MD, A, AM, AD, AMD` (`M` stands for `RAM[A]`)

jump = `null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP`

Semantics:

- computes the value of *comp* and stores the result in *dest*;
- if (*comp jump* `0`) is true, jumps to execute the instruction in `ROM[A]`.

Standard VM mapping on the Hack platform

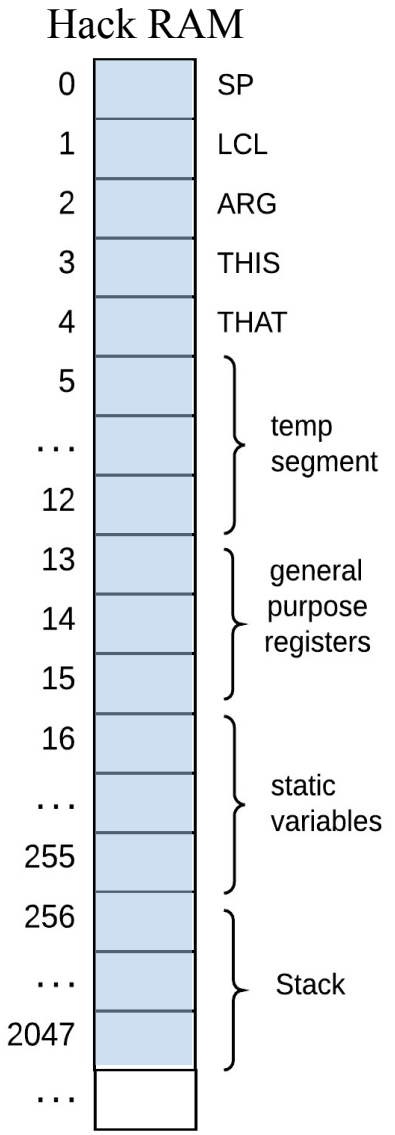
VM mapping decisions:

- How to map the VM's data structures using the host hardware platform
- How to express the VM's commands using the host machine language

Standard mapping:

- Specifies how to do the mapping in an agreed-upon way
- Benefits:
  - Compatibility with other software systems
  - Standard testing.

Standard VM mapping on the Hack platform



`local, argument, this, that:`
allocated dynamically to the `RAM`. 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*

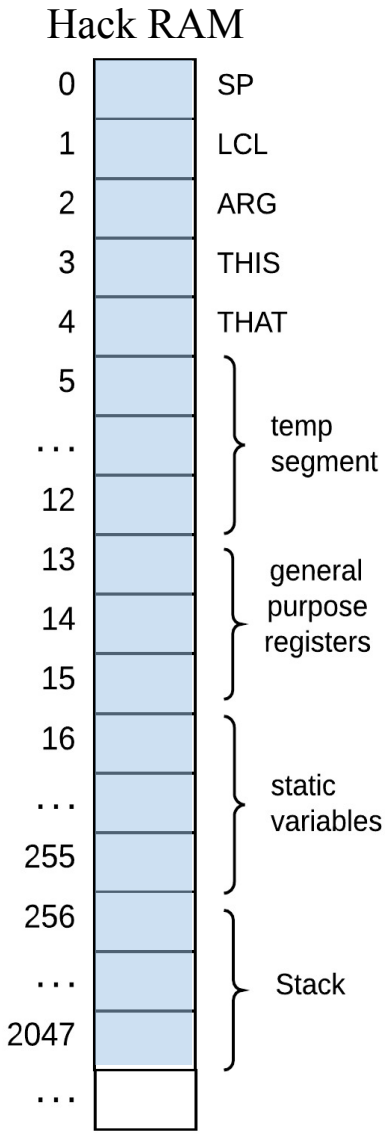
`static:` accessing `static` *i* within file `Foo.vm` should result in accessing the assembly variable `Foo.`*i*

`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



In order to realize this mapping, the VM translator should use some special variables / 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.

Implementation note:

The standard mapping will be extended in project 8, when we'll complete the VM translator's implementation.

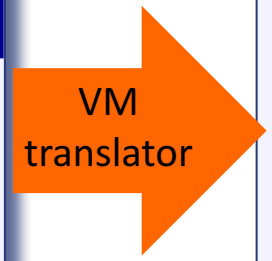
Virtual machine: lecture plan

- ✓ The road ahead
- ✓ Program compilation overview
- ✓ VM abstraction:
  - ✓ the stack
  - ✓ memory segments
- ✓ VM implementation:
  - ✓ the stack
  - ✓ memory segments
- ✓ The VM emulator
- ✓ VM implementation on the Hack platform
- The VM translator: proposed implementation
- Building the VM translator, Part I

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 commands that complete 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

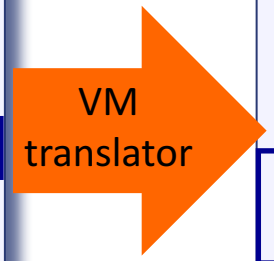
Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
... additional assembly commands that complete 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
...
```



Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
... additional assembly commands that complete 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 commands that complete 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 commands that complete 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)

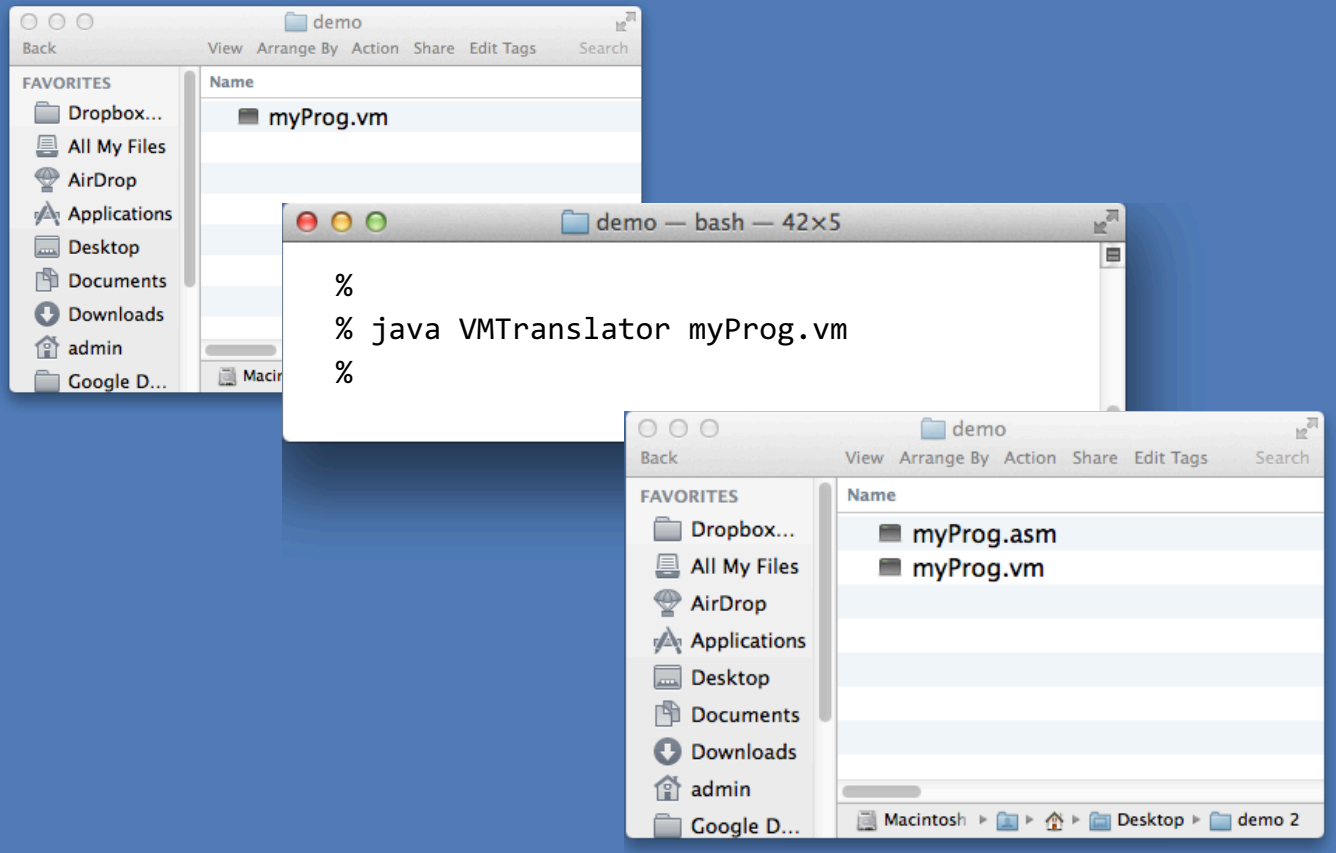
```
...
push constant 17
push local 2
add
pop argument 1
...
```

VM translator

Generated assembly code (*fileName*.asm)

```
...
// push constant 17
@17
D=A
... additional assembly commands that complete 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



Implementation

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

- 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 all white space and comments

Routine	**Arguments**	**Returns**	**Function**
Constructor	Input file / stream	—	Opens the input file/stream and gets ready to parse it.
`hasMoreCommands`	—	Boolean	Are there more commands in the input?
`advance`	—	—	Reads the next command from the input and makes it the *current command*. Should be called only if `hasMoreCommands()` is true. Initially there is no current command.

Parser

- 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 all 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`	Returns a constant representing the type of the current command. `C_ARITHMETIC` is returned for all the arithmetic/logical commands.
`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

Generates assembly code from the parsed VM command:

Routine	**Arguments**	**Returns**	**Function**
Constructor	Output file / stream	—	Opens the 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 command.
`WritePushPop`	command (`C_PUSH` or `C_POP`), `segment` (string), `index` (int)	—	Writes to the output file the assembly code that implements the given command, where command is either `C_PUSH` or `C_POP`.
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.

The big picture

VM language:

Arithmetic / Logical commands:

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

Memory access commands:

`pop` *segment i*

`push` *segment i*

Project 7

Branching commands:

`label` *label*

`goto` *label*

`if-goto` *label*

Function commands:

`function` *functionName nVars*

`call` *functionName nArgs*

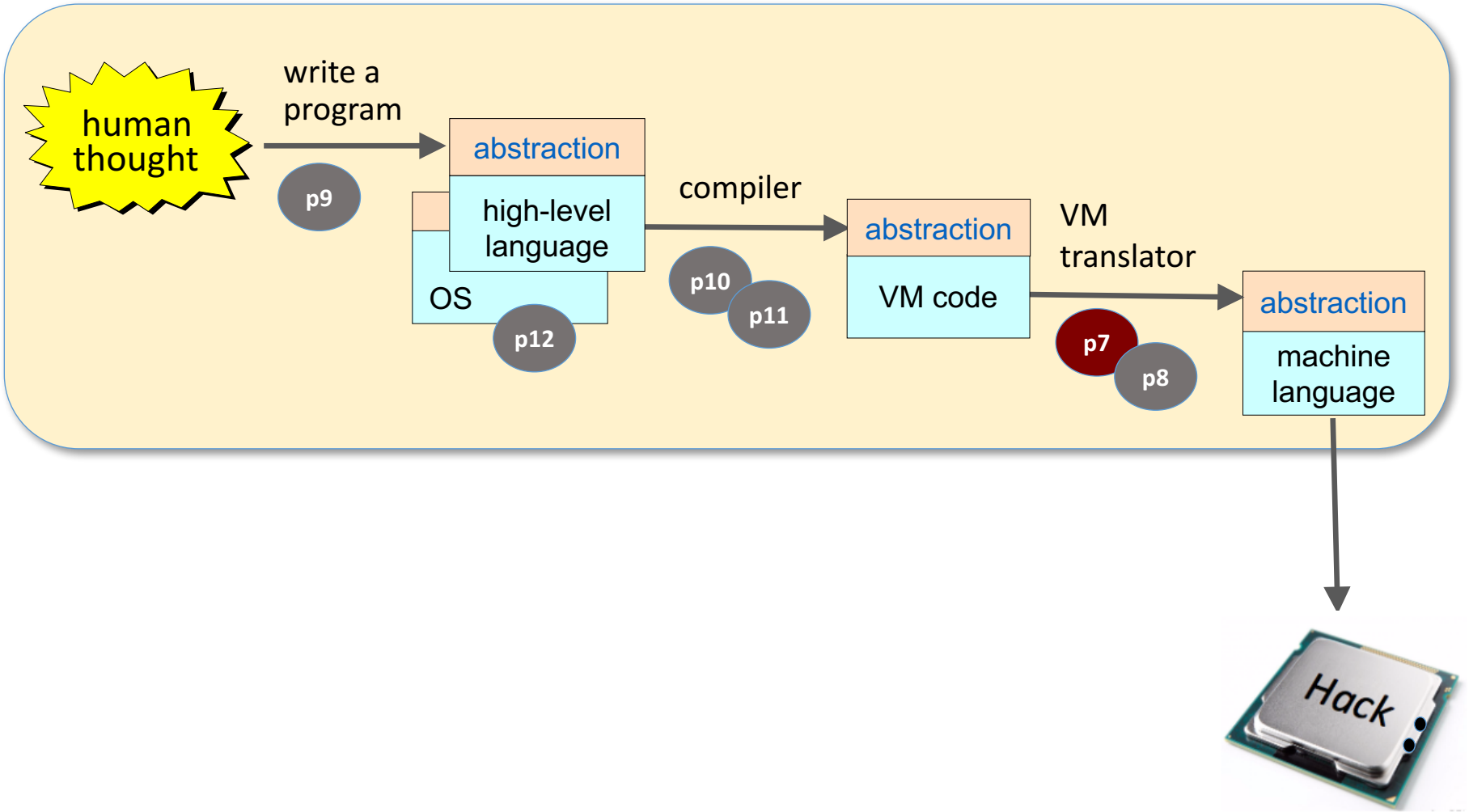
`return`

Project 8

Virtual machine: lecture plan

- ✓ The road ahead
- ✓ Program compilation overview
- ✓ VM abstraction:
  - ✓ the stack
  - ✓ memory segments
- ✓ VM implementation:
  - ✓ the stack
  - ✓ memory segments
- ✓ The VM emulator
- ✓ VM implementation on the Hack platform
- ✓ The VM translator: proposed implementation
- • Building the VM translator, Part I

The big picture



The VM language

VM language:

Arithmetic / Logical commands:

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

Memory access commands:

`pop` *segment i*

`push` *segment i*

Branching commands:

`label` *label*

`goto` *label*

`if-goto` *label*

Function commands:

`function` *functionName nVars*

`call` *functionName nArgs*

`return`

Project 7

Objective: build a basic VM translator that handles a subset of the VM language: stack arithmetic and memory access (`push`/`pop`) commands

VM language:

Arithmetic / Logical commands:

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

Memory access commands:

`pop` *segment i*

`push` *segment i*

Branching commands:

`label` *label*

`goto` *label*

`if-goto` *label*

Function commands:

`function` *functionName nVars*

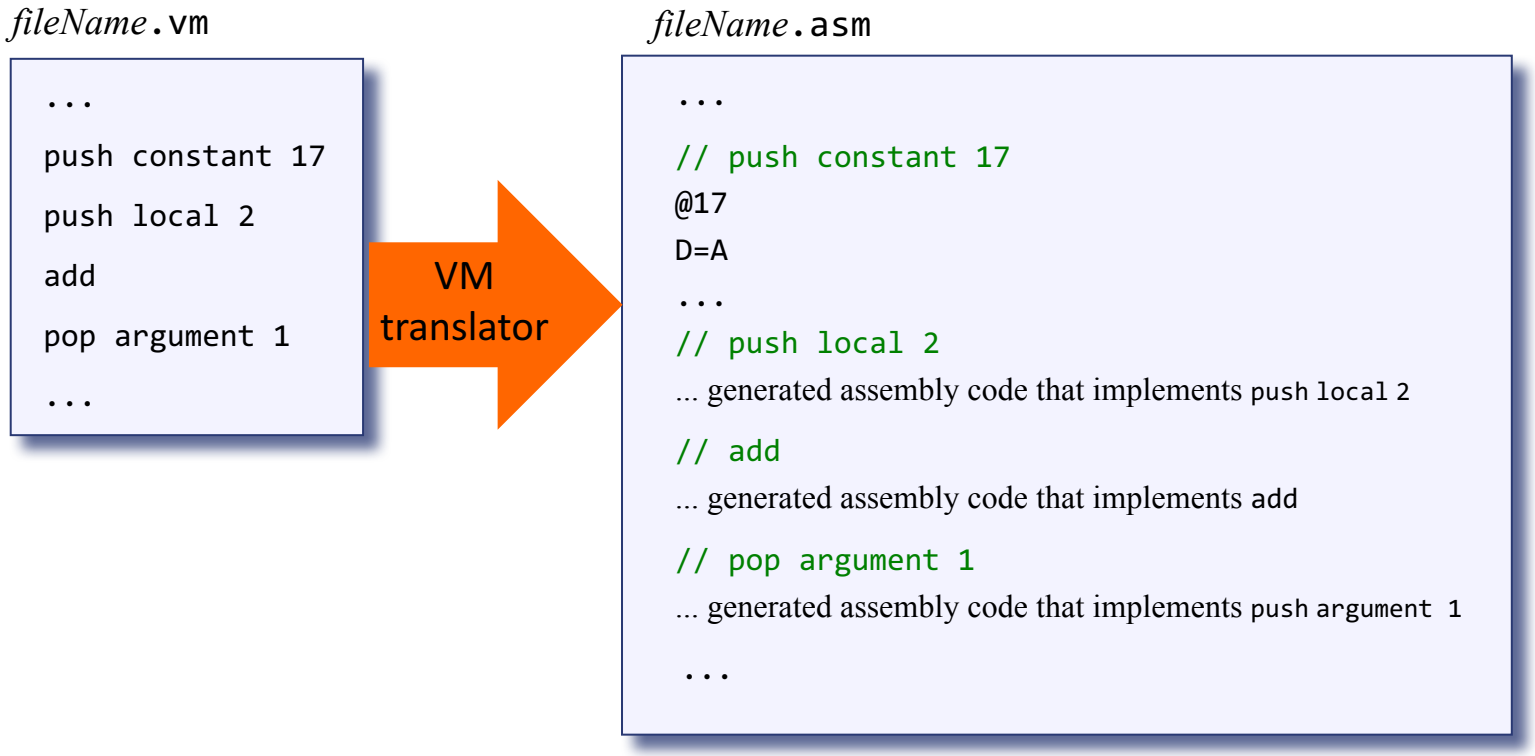
`call` *functionName nArgs*

`return`

Project 7

Project 7

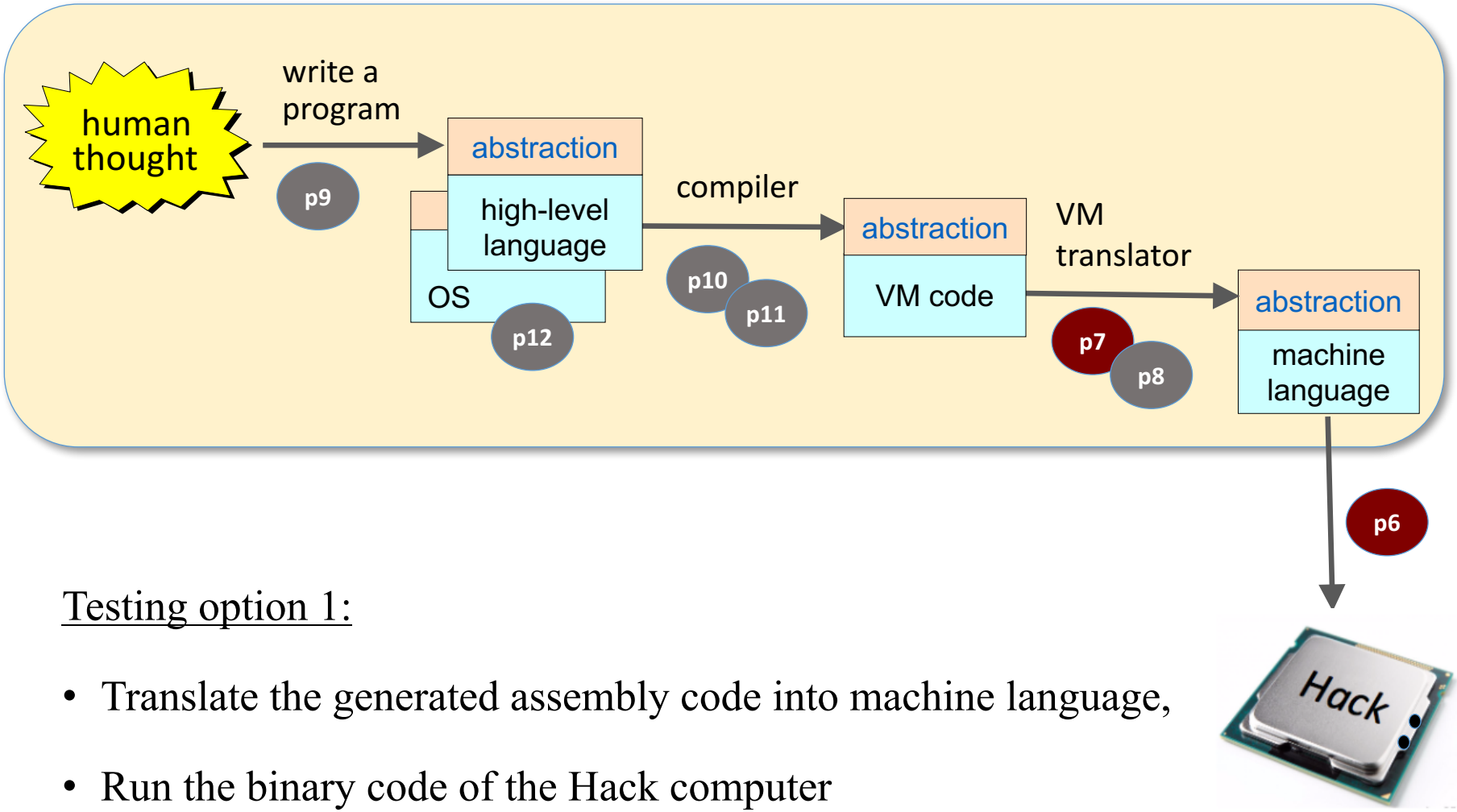
Objective: build a basic VM translator that handles a subset of the VM language: stack arithmetic and memory access (`push`/`pop`) commands



To test the translation:

Run the generated code on the target platform

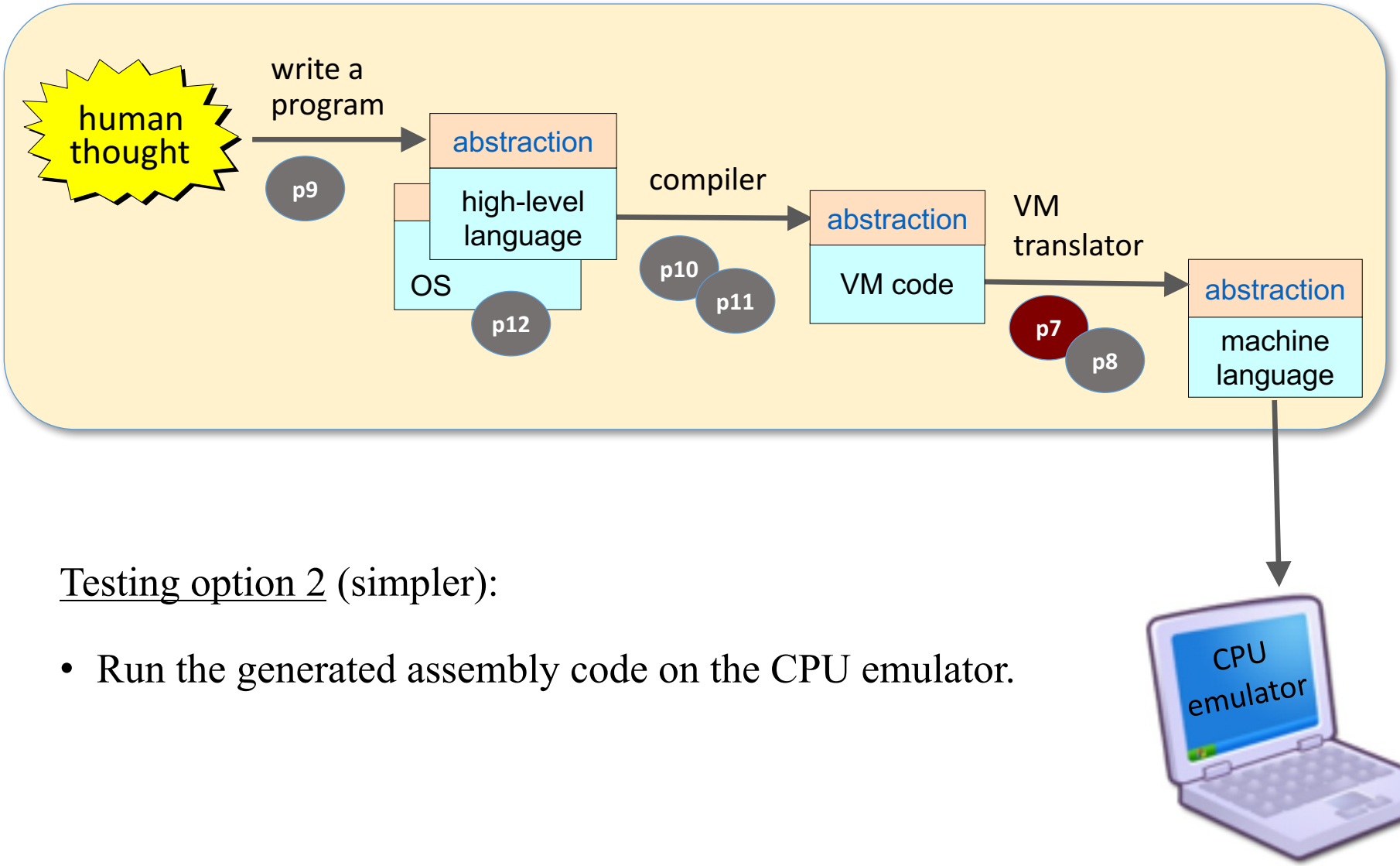
Project 7: testing



Testing option 1:

- Translate the generated assembly code into machine language,
- Run the binary code of the Hack computer

Project 7: testing



Testing option 2 (simpler):

- Run the generated assembly code on the CPU emulator.

Development Plan

Objective: build a basic VM translator that handles the VM language *stack arithmetic* and *memory access* (`push`/`pop`) commands

Contract

- Write a VM-to-Hack translator, conforming to the *Standard VM-on-Hack Mapping*
- Use your VM translator to translate and test the supplied `.vm` programs, yielding corresponding `.asm` programs
- When executed on the supplied CPU emulator, the generated `.asm` programs should deliver the same results mandated by the supplied test scripts and compare files.

Test programs

- `SimpleAdd`
- `StackTest`
- `BasicTest`
  - `BasicTest.vm`
  - `BasicTest.tst`
  - `BasicTest.cmp`
  - `BasicTestVME.tst`
- `PointerTest`
- `StaticTest`

`BasicTest.vm` (example)

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

`BasicTest.asm`

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

Development Plan

Objective: build a basic VM translator that handles the VM language *stack arithmetic* and *memory access* (`push`/`pop`) commands

For each test *xxx*`.vm` program:

0. (optional) load *xxx*`VME.tst` into the VM emulator; run the test script and inspect the program's operation
1. use your 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 *xxx*`.tst` into the CPU emulator
4. Run the test script, inspect the results
5. If there's a problem, fix your translator and go to stage 1.

Test programs

- `SimpleAdd`
- `StackTest`
- `BasicTest`
  - `BasicTest.vm`
  - `BasicTest.tst`
  - `BasicTest.cmp`
  - `BasicTestVME.tst`
- `PointerTest`
- `StaticTest`

`BasicTest.vm` (example)

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

`BasicTest.asm`

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

Tools and resources

Objective: build a basic VM translator that handles the VM language
stack arithmetic and *memory access* (`push`/`pop`) commands

Tools and resources:

- Test programs and compare files: `nand2tetris/projects/07`
- Experimenting with the test VM programs: the supplied *VM emulator*
- Translating the test 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*

Some testing issues

Objective: build a **basic** VM translator that will be extended in project 8

BasicTest.vm

```
function Foo.bar
...
push constant 510
pop temp 6
push local 0
push that 5
add
push argument 1
sub
push this 6
...
return
```

BasicTest.asm

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

There's no need to delve into the test's code

BasicTest.tst (for CPU emulator)

```
load BasicTest.asm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
output-list RAM[256] RAM[300] ...

set RAM[0] 256,   // base of stack
set RAM[1] 300,   // base of local
set RAM[2] 400,   // base of argument
set RAM[3] 3000,  // base of this
set RAM[4] 3010,  // base of that

repeat 600 {
  ticktock;
}
output;
```

The stack and segment base addresses are handled by the supplied test scripts, using fixed addresses

(In project 8 the mappings will be handled dynamically, by the VM implementation)

Missing elements (in project 7)

- `function` / `return` "envelope"
- Mapping of the stack and the memory segments on the host RAM

Recap

Recap

Objective: build a basic VM translator that handles the VM language *stack arithmetic* and *memory access* (`push`/`pop`) commands

VM language:

Arithmetic / Logical commands:

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

Memory access commands:

`push` *segment i*

`pop` *segment i*

Project 7

Branching commands:

`label` *label*

`goto` *label*

`if-goto` *label*

Function commands:

`function` *functionName nVars*

`call` *functionName nArgs*

`return`

Project 8

Perspective

(A subset of historical notes and additional issues)

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