University of Nottingham
UK | CHINA | MALAYSIA

Virtual Machine (Part 2)

Dr. Wooi Ping Cheah

Outlines

- Introduction to virtual machine
- VM abstraction
- VM implementation
  - Stack
  - Memory segment commands
  - Branching commands
  - Function commands

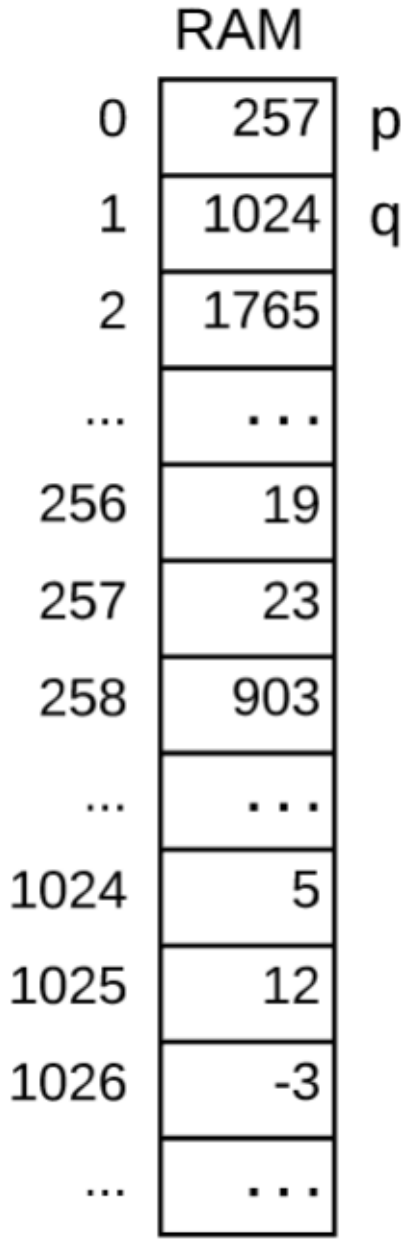
Pointer manipulation

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
```

In Hack:

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



Notation:

*p // the memory content that p points at

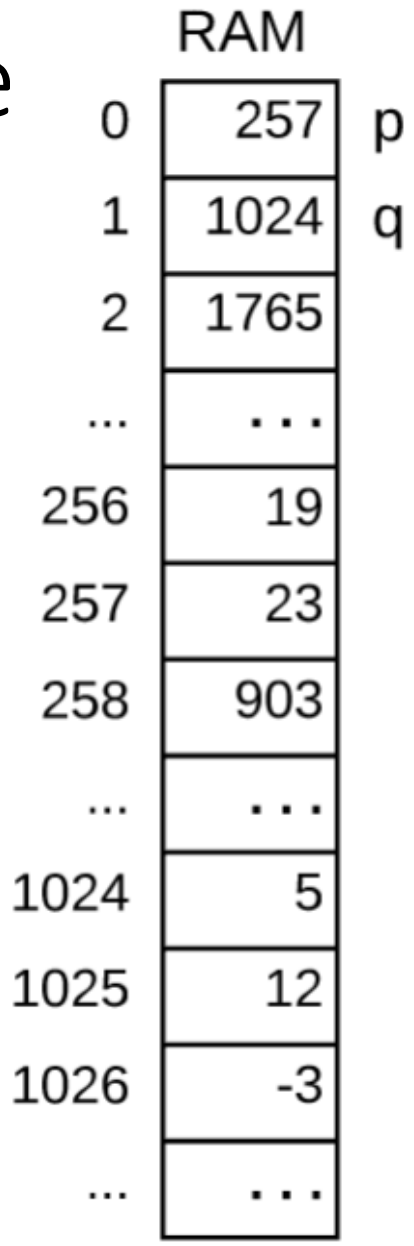
x-- // decrement: x = x - 1

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

Pointer manipulation - exercise

Given the initial memory status shown on the right, find:

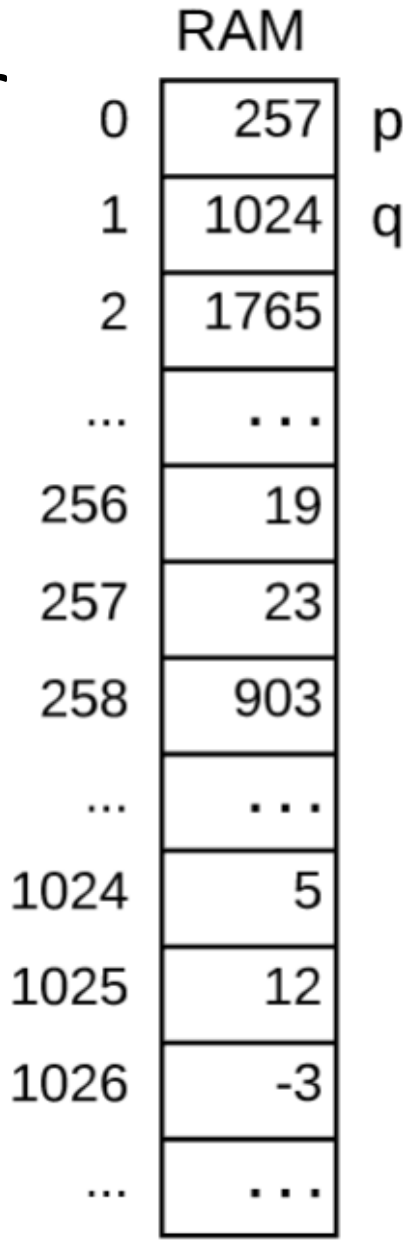
```
p++ // What is RAM[0]?

D = *p //What is D?

*q = D // What is *q?

q++    //What is RAM[1]?

*p = *q //What is *p?
```



Pointer manipulation - answer

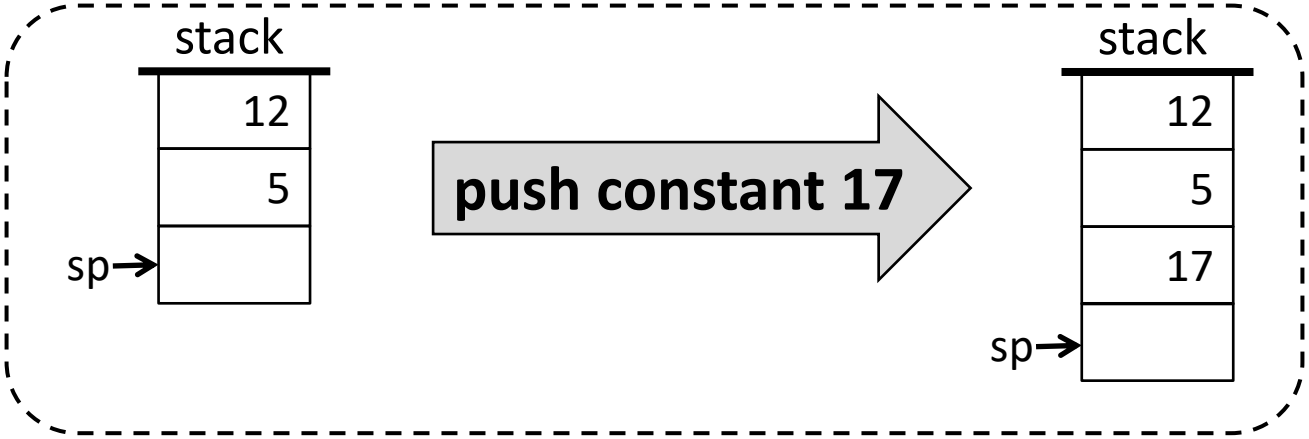
Given the initial memory status shown on the right, find:

```
p++ // What is RAM[0]? 258

D = *p //What is D? 903

*q = D // What is *q? 903

q++    //What is RAM[1]? 1025

*p = *q //What is *p? 12
```



Stack implementation

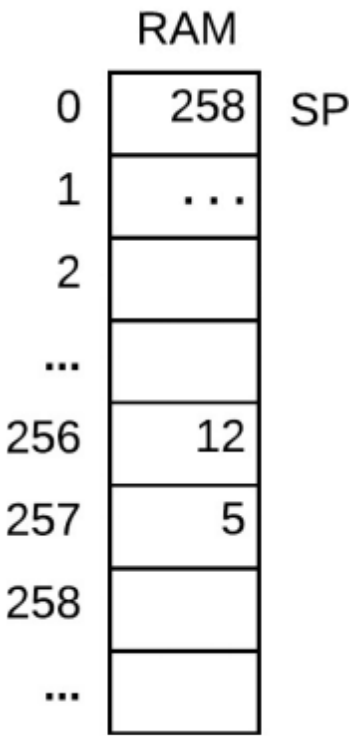
Abstraction:



Implementation:

Assumptions:

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

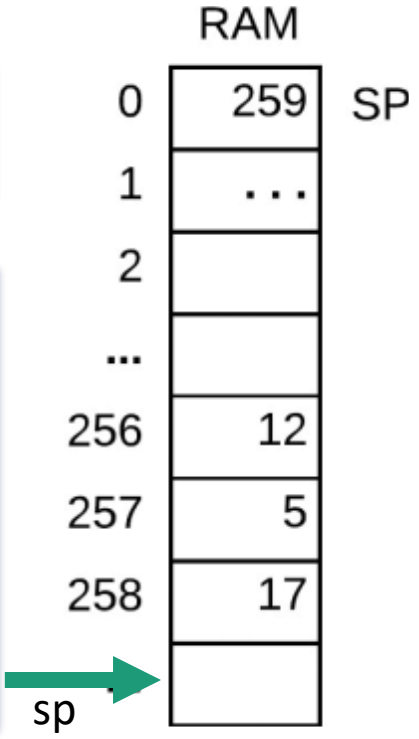


Logic:

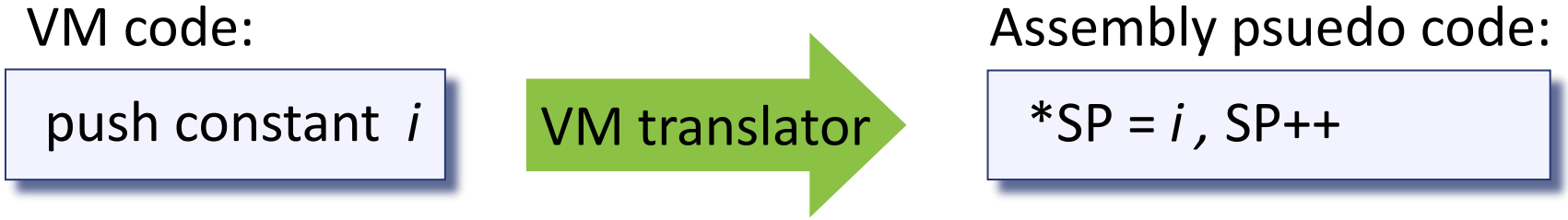
```
*SP = 17
SP++
```

Hack assembly:

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



Stack implementation



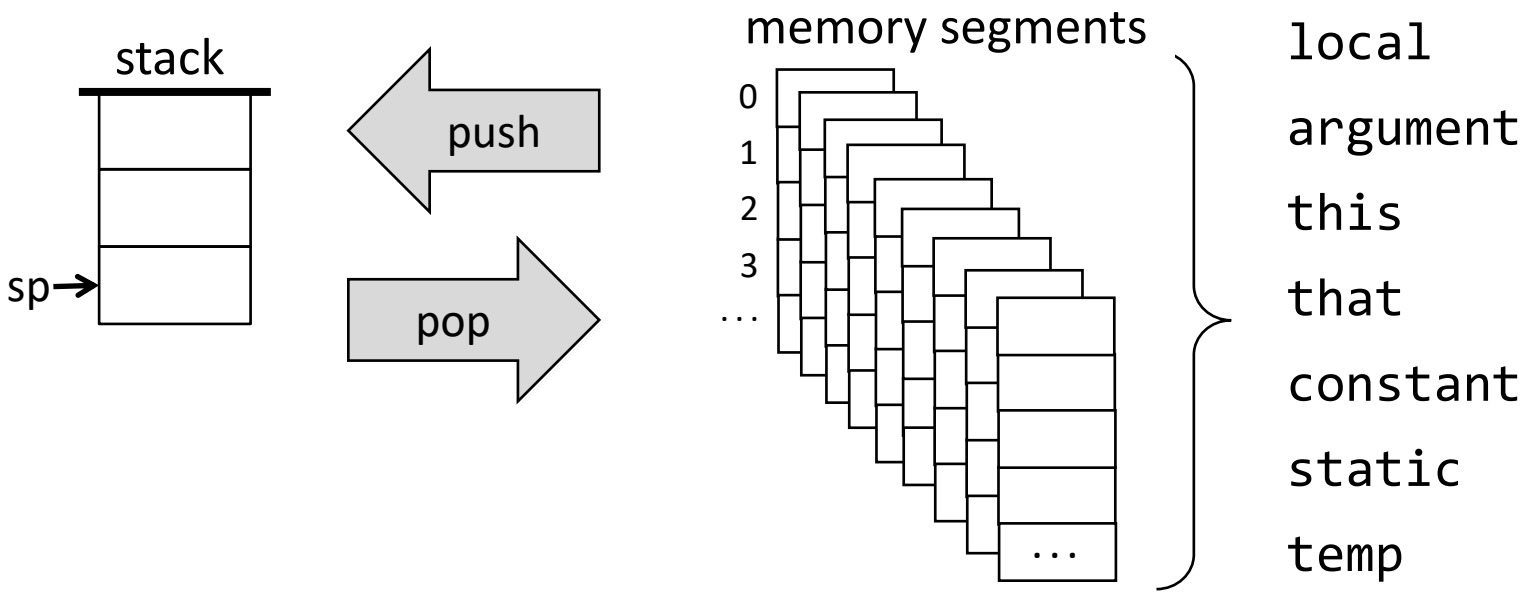
VM Translator

- 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 **realize** the stack and the memory segments on the host platform.

Outlines

- Introduction to virtual machine
- VM abstraction
- VM implementation
  - Stack
  - Memory segment commands
  - Branching commands
  - Function commands

Memory segments (abstraction)

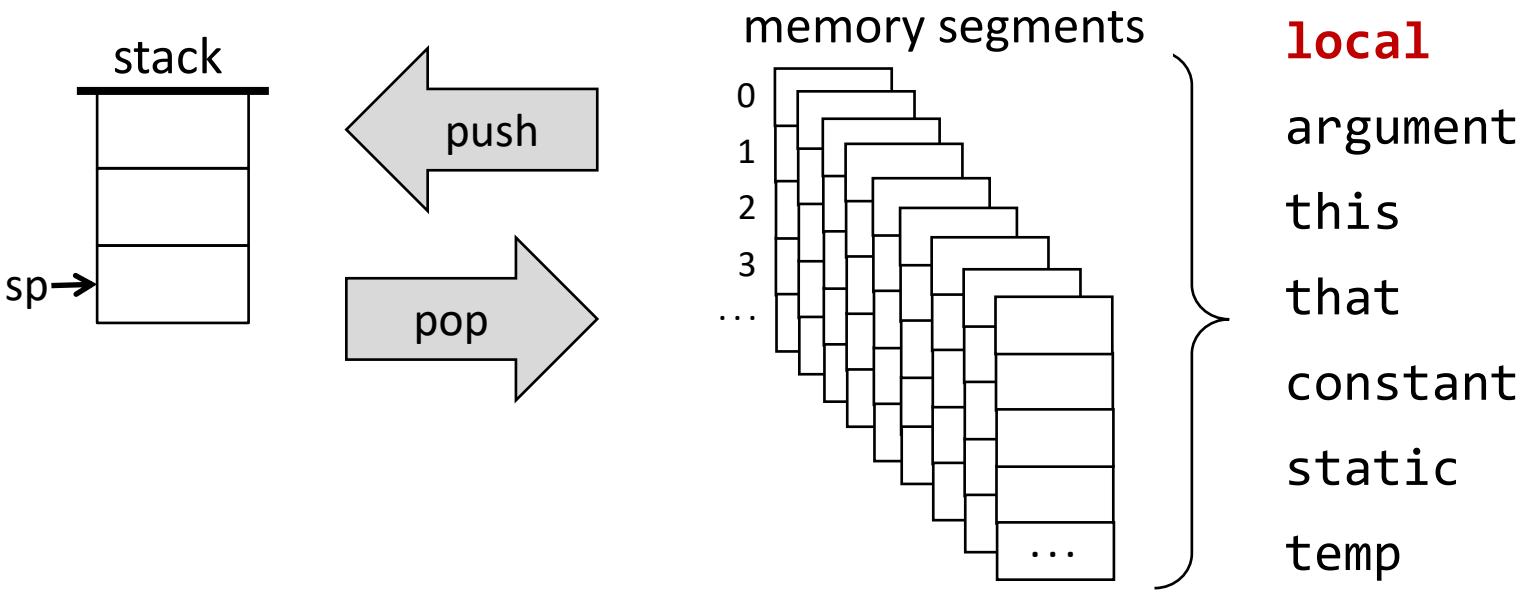


Syntax: push / pop segment i

Examples:

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

Implement `push/pop local` *i*



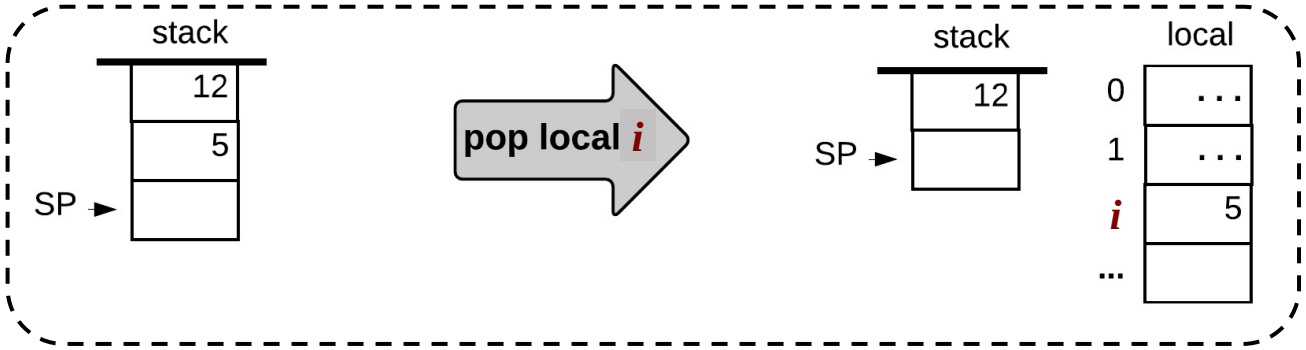
Syntax: push/pop local *i*

Why do we need a local segment?

- High-level code on ***local variables*** are translated into VM operations on the entries of the segment ***local***.

Implement pop local *i*

Abstraction:

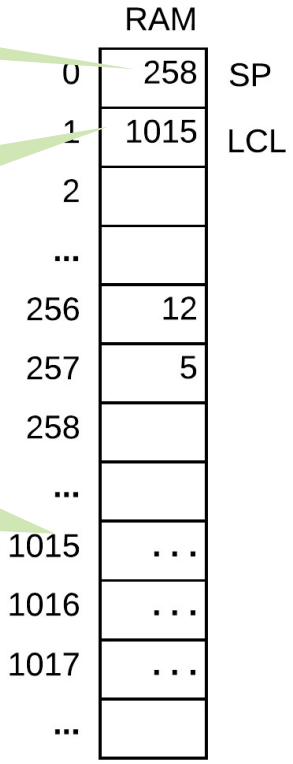


stack pointer

base address of the local segment

Implementation:

the local segment is stored some-where in the RAM

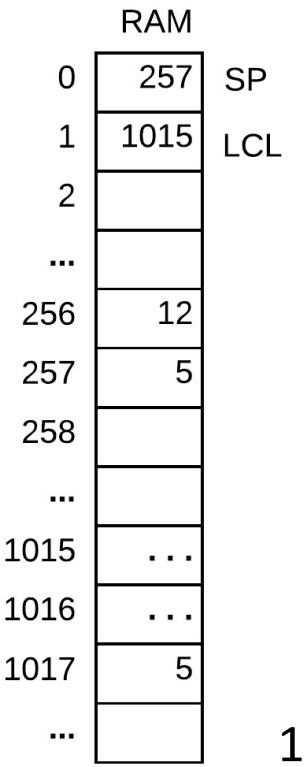


Implementation:

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

Hack assembly:

On next slide!



Implement `pop local` *i*

Abstraction

pop local ***i***

Implementation:

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

***i* is a constant here!!!**

Hack assembly:

```
@i      // addr=LCL+i
D=A
@LCL
D=D+M
@addr
M=D
@SP     // SP--
M=M-1
@SP     // D=*SP
A=M
D=M
@addr   // *addr=D
A=M
M=D
```

Implement `push/pop local` *i*

VM code:

`pop local` *i*

`push local` *i*

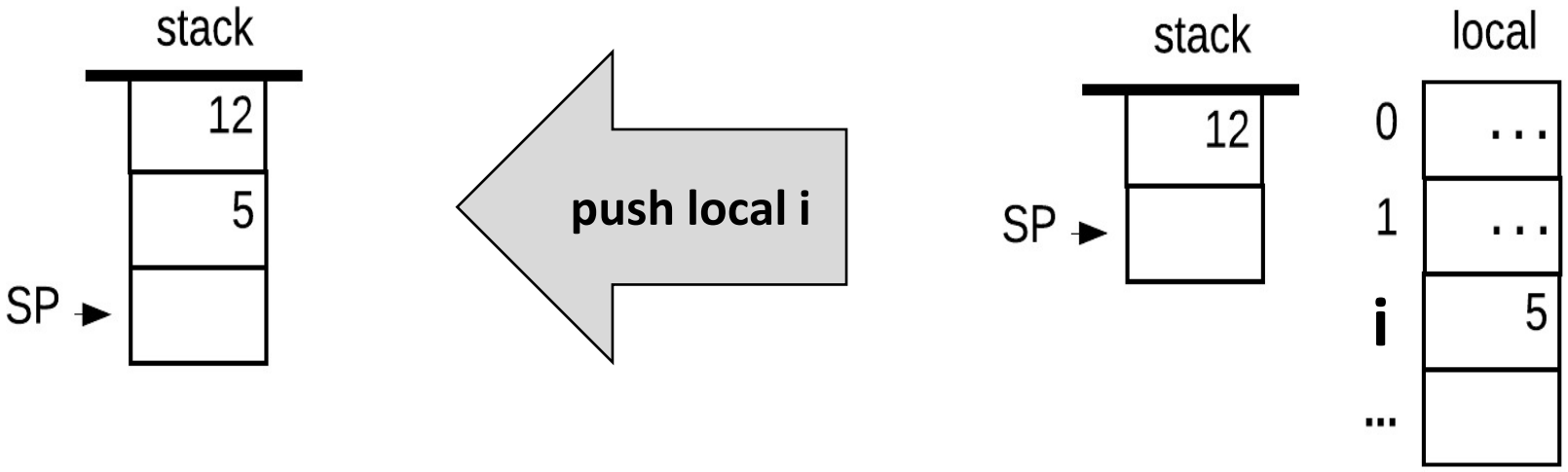
VM Translator

Assembly pseudo code:

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

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

Abstraction:



Implement `push/pop local i`

VM code:

```
pop local i
```

```
push local i
```

VM Translator

Assembly pseudo code:

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

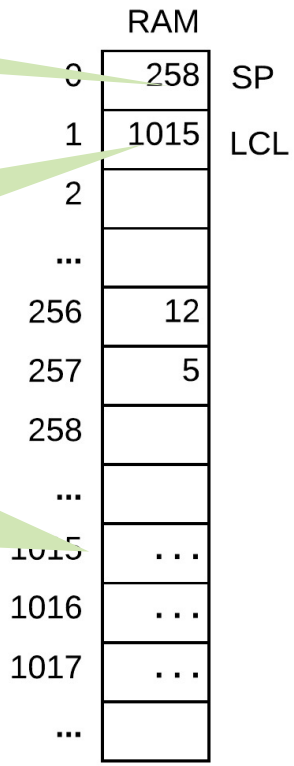
```
addr = LCL+ i, *SP = *addr, SP++
```

Stack pointer

Base address of the local segment

Implementation:

The local segment is stored some-where in the RAM

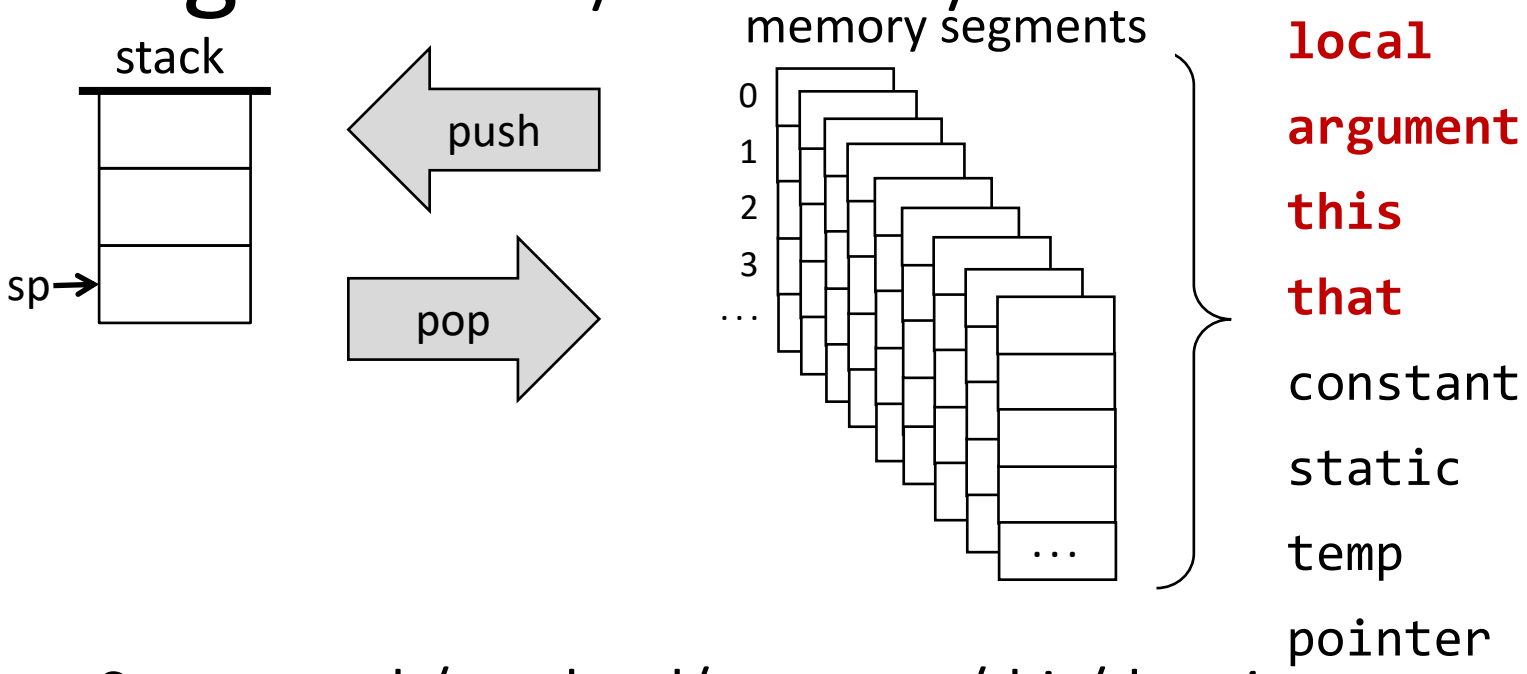


Hack assembly:

```
// implement
// push local i
// addr=LCL+i
@i
D=A
@LCL
D=D+M
@addr
M=D
```

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

Implement `push / pop local / argument / this / that` *i*



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

	High-level language	VM code
local	*local* variable	*local i*
argument	*argument* in a function call	*argument i*
this	*field* variables of the current object	*this i*
that	*array entries*	*that i*

Implement `push / pop local / argument / this / that` *i*

VM code:

`push` *segment i*

`pop` *segment i*

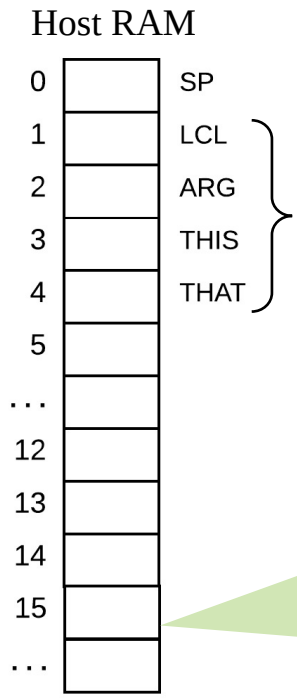
VM translator

Assembly pseudo code:

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

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

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



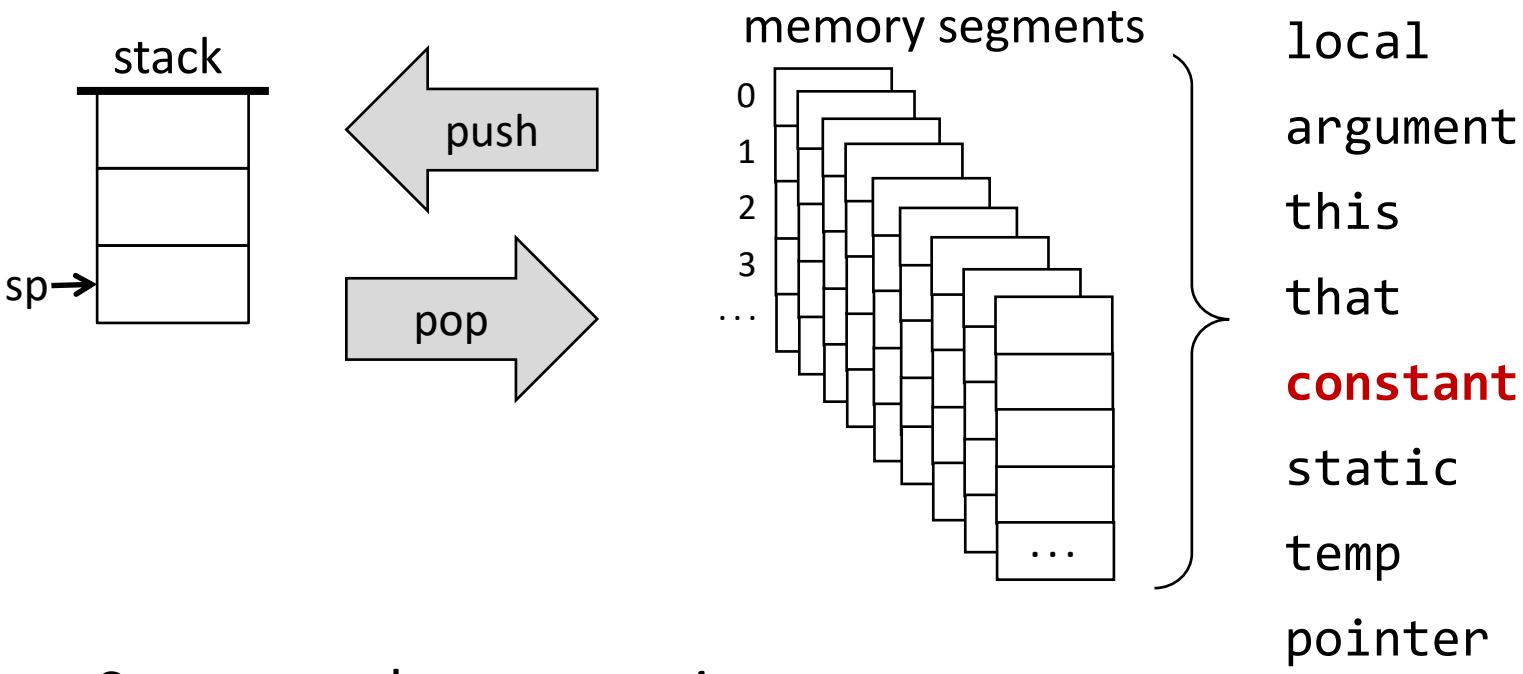
base addresses of the four segments are stored in these pointers

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*

implemented precisely the same way.

Implement `push constant` *i*



Syntax: push constant *i*

Why do we need a constant segment?

- High-level code on the *constant i* are translated into VM operations on the segment entry constant *i.*

Implement `push constant` *i*

VM code:

```
push constant i
```

Assembly psuedo code:

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

(no pop constant operation)

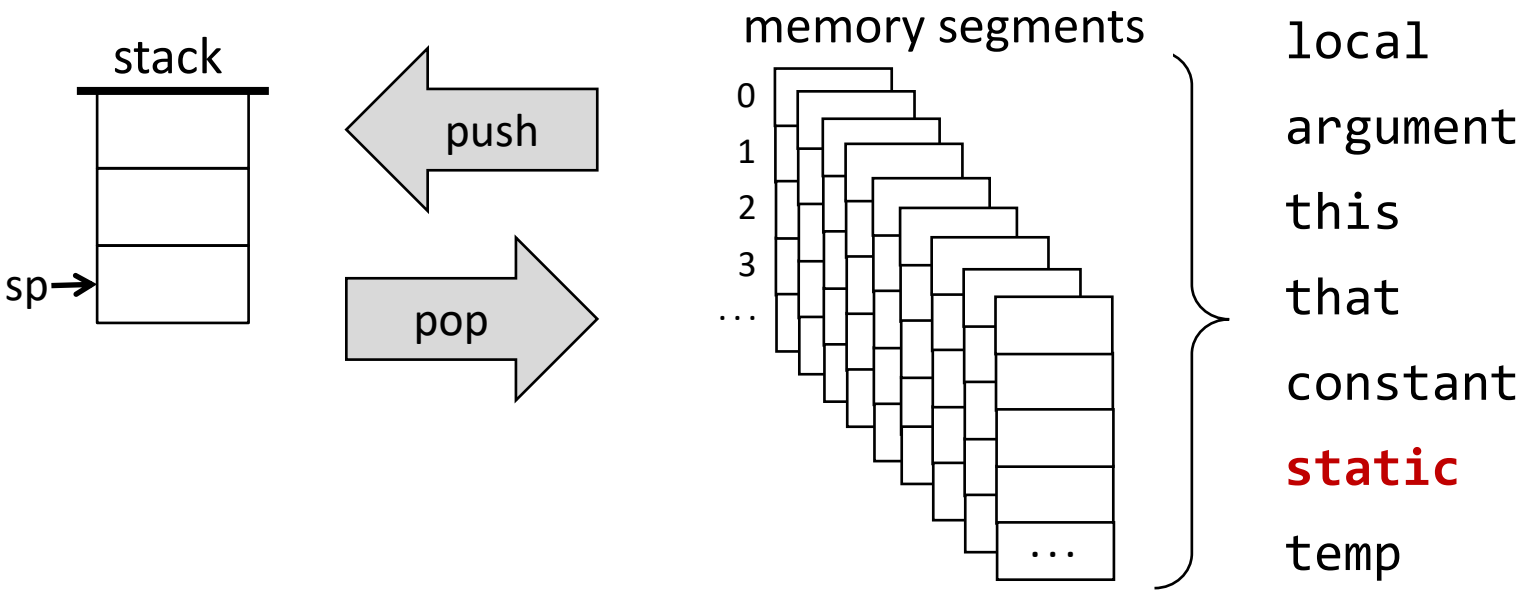
Hack assembly:

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

Implementation:

Supplies the specified constant.

Implementing `push/pop static i`



Syntax: `push/pop static i`

Why do we need a static segment?

- High-level operations on *static* variables are translates into VM operations on entried of the segment *static*.
- Static variables can be used as "global" variables, or to store constant values.

Implement `push/pop static i`

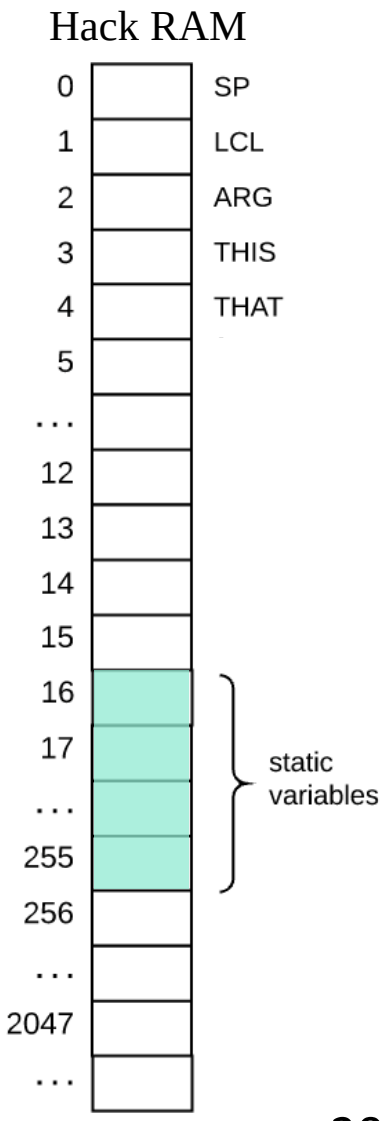
VM code:

```
// 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
...
```



<u>The challenge:</u>

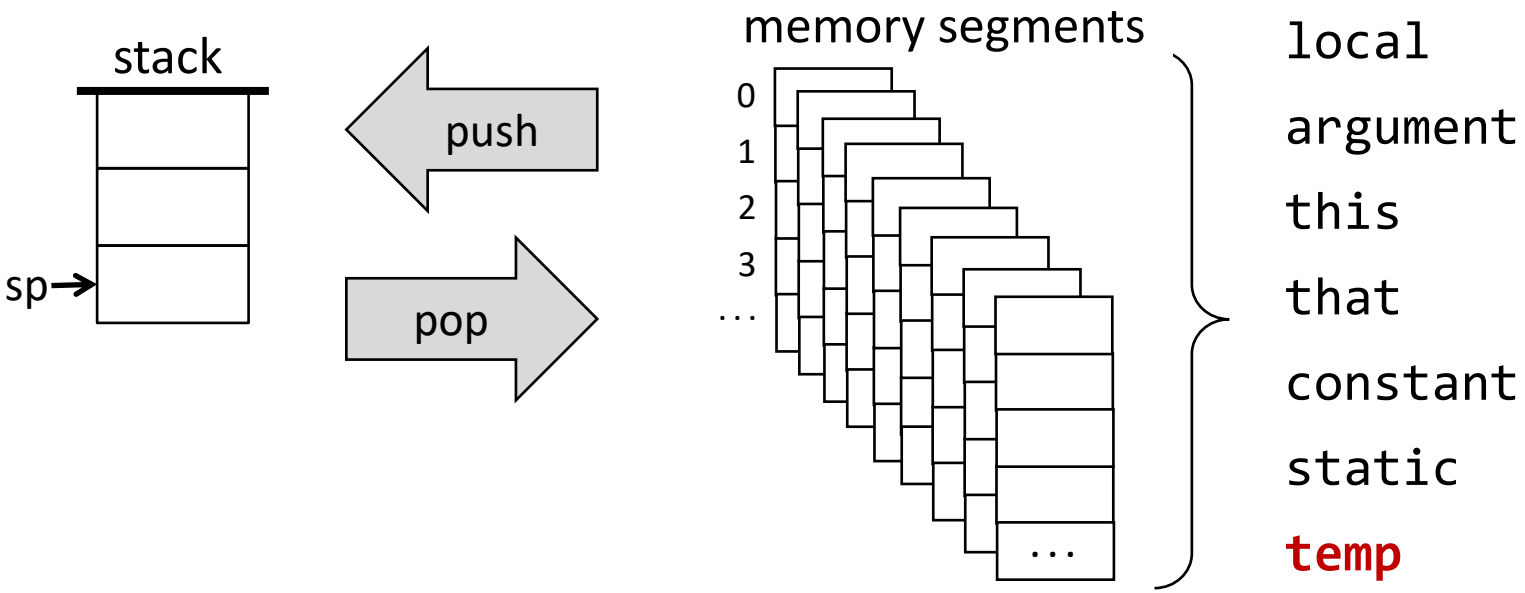
Static variables should be seen by all the methods in a program.

<u>Solution:</u>

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.

Implement `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*.

Implement `push/pop temp` *i*

VM code:

`push temp` *i*

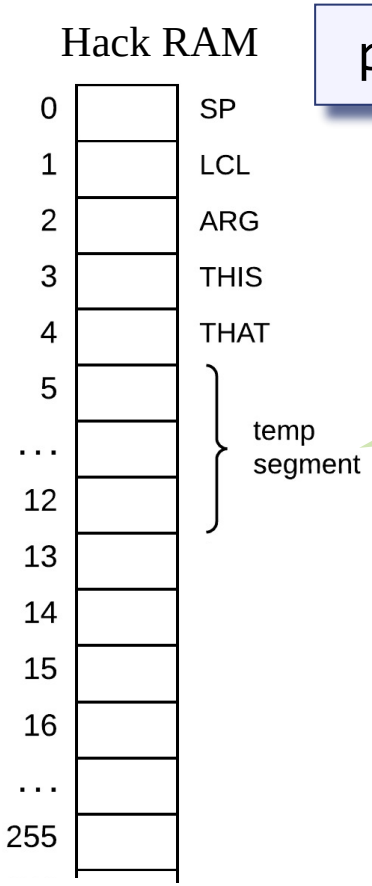
`pop temp` *i*

VM Translator

Assembly psuedo code:

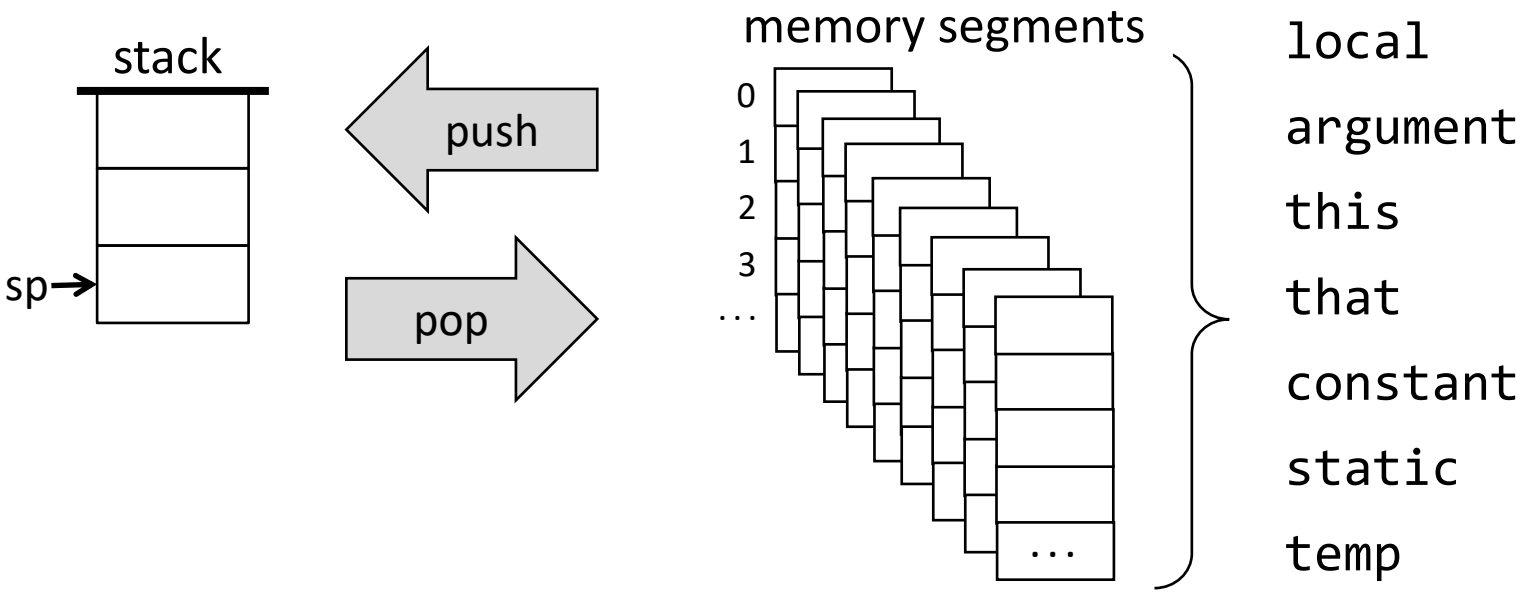
`addr = 5 + i, *SP = *addr, SP++`

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



A fixed, **8-place** memory segment, stored in RAM locations 5 to 12

Implement `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 writing the compiler.

Implement `push/pop pointer 0/1`

VM code:

```
push pointer 0/1
```

```
pop pointer 0/1
```

VM Translator

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.
// THIS and THAT: Built-in symbols.
```

Outlines

- Introduction to virtual machine
- VM abstraction
- VM implementation
  - Stack
  - Memory segment commands
  - Branching commands
  - Function commands

Branching

- goto *label*
  - jump to execute the command just after *label*
- if-goto *label*
  - *cond* = pop
  - if *cond* jump to execute the command just after *label*
- label *label*
  - label declaration command
- Implementation (VM translation):
  - The assembly language has similar branching commands.

Outlines

- Introduction to virtual machine
- VM abstraction
- VM implementation
  - Stack
  - Memory segment commands
  - Branching commands
  - Function commands

Functions in VM language: implementation

```
// Computes 3 +8* 5
0 function main 0
1   push constant 3
2   push constant 8
3   push constant 5
4   call mult 2
5   add
6   return
```

caller

```
// Computes the product of two given arguments
0 function mult 2
1   push constant 0
2   pop local 0
3   push constant 1
4   pop local 1
5 label LOOP
6   push local 1
7   push argument 1
    //... computes the product into local 0
19 label END
20   push local 0
21   return
```

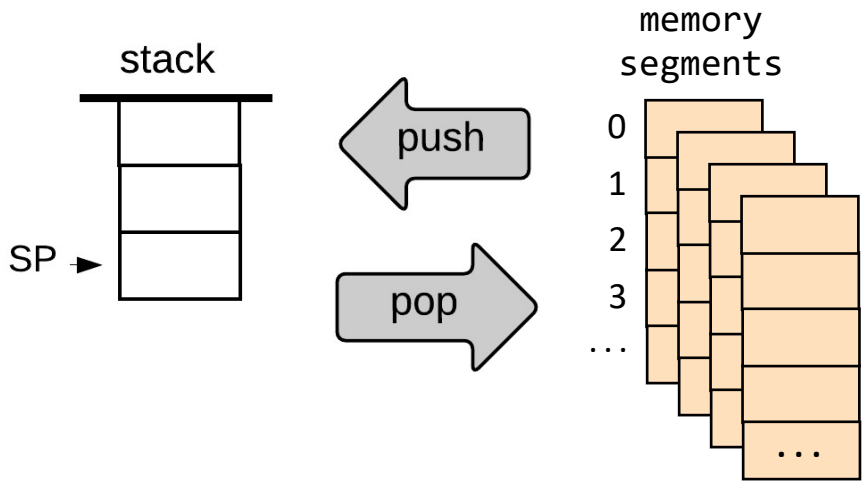
callee

Implementation

We can write low-level code to

- Handle the VM command call,
- Handle the VM command function,
- Handle the VM command return.

The function's state



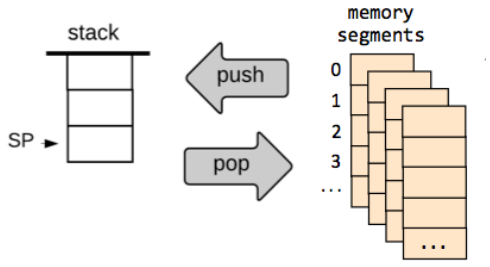
<u>During run-time:</u>

- Each function uses a **working stack** + **memory segments**
- The working stack and some of the segments should be:
  - **Created** when the function starts running,
  - **Maintained** as long as the function is executing,
  - **Recycled** when the function returns.

The function's state

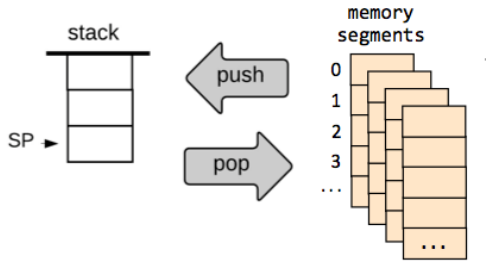
```
function main 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   return
```

caller




```
function mult 2
   push constant 0
   pop local 0
   ...
label LOOP
   push local 1
   //... computes the product
label END
   push local 0
   return
```

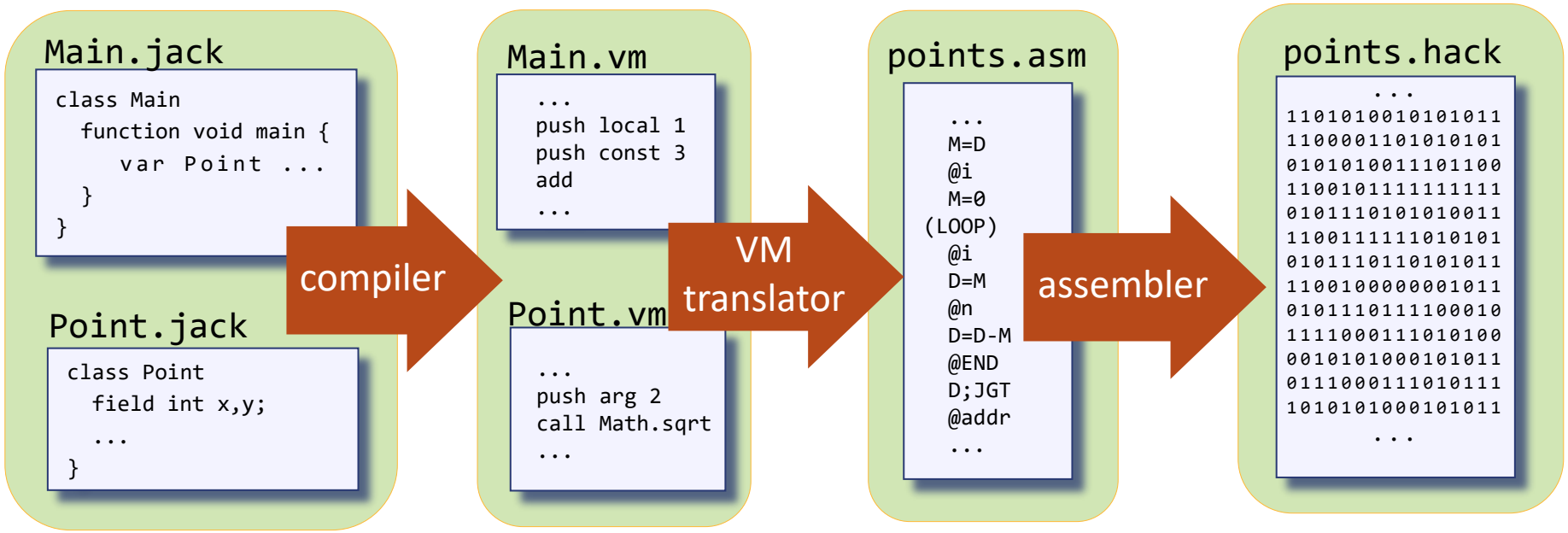
callee



Challenge:

- Maintain the states of all the functions up the calling chain.
- Can be done by using a single ***global*** *stack*.

Big picture


Main.jack
class Main
function void main {
var Point ...
}
}
Point.jack
class Point
field int x,y;
...
}
compiler
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
points.hack
(4,6)
5
Hack

Summary

- VM bridges the high-level programming language and the machine code.
- VM is implemented using stack
  - Arithmetic/logic operation
  - memeory segment
  - branching
  - function

Acknowlegement

- This set of lecture notes are based on the lecture notes provided by Noam Nisam / Shimon Schocken.
- You may find more information on: www.nand2tetris.org.