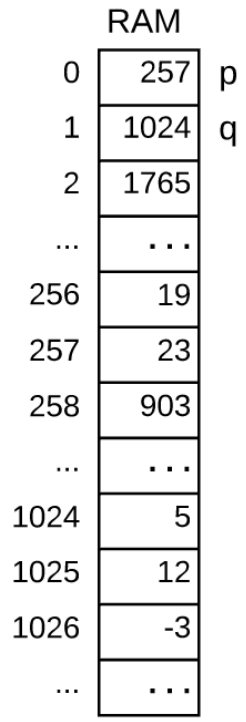
# Pointer manipulation

Pseudo assembly code

```
D = *p
```




## Notation:

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

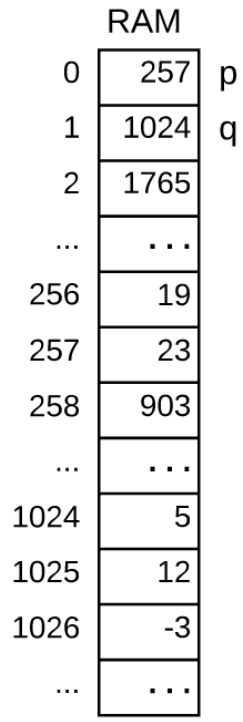
# Pointer manipulation

Pseudo assembly code

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

In Hack:

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




## Notation:

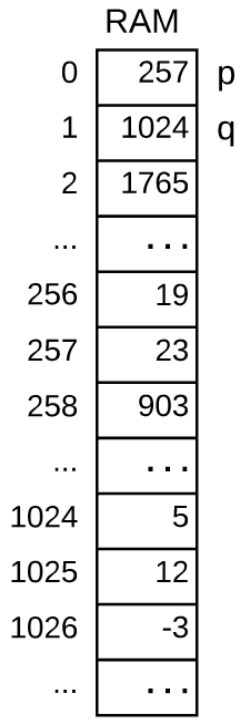
`*p` // the memory location that `p` points at

# Pointer manipulation

Pseudo assembly code

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

p--
D = *p
```




## Notation:

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

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

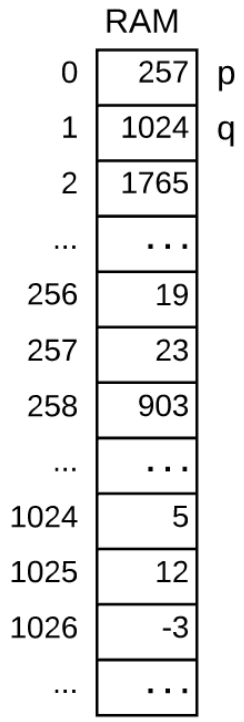
`x--` // decrement: `x = x - 1`

# Pointer manipulation

Pseudo assembly code

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

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




## Notation:

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

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

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

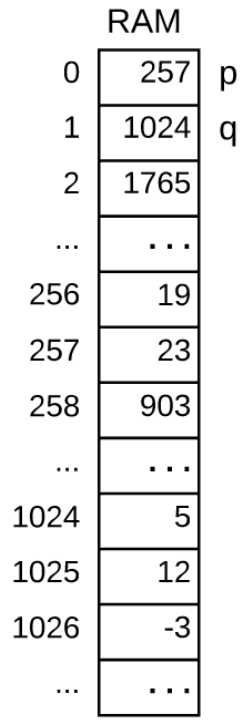
# Pointer manipulation

Pseudo assembly code

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

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

*q = 9
q++
```




## Notation:

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

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

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

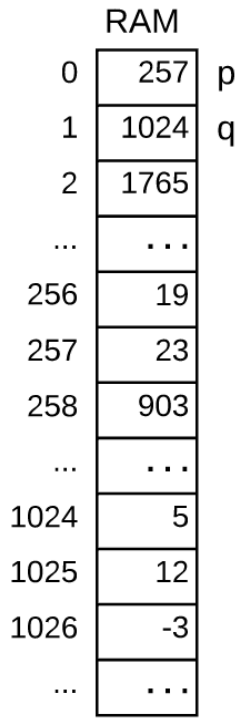
# Pointer manipulation

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:

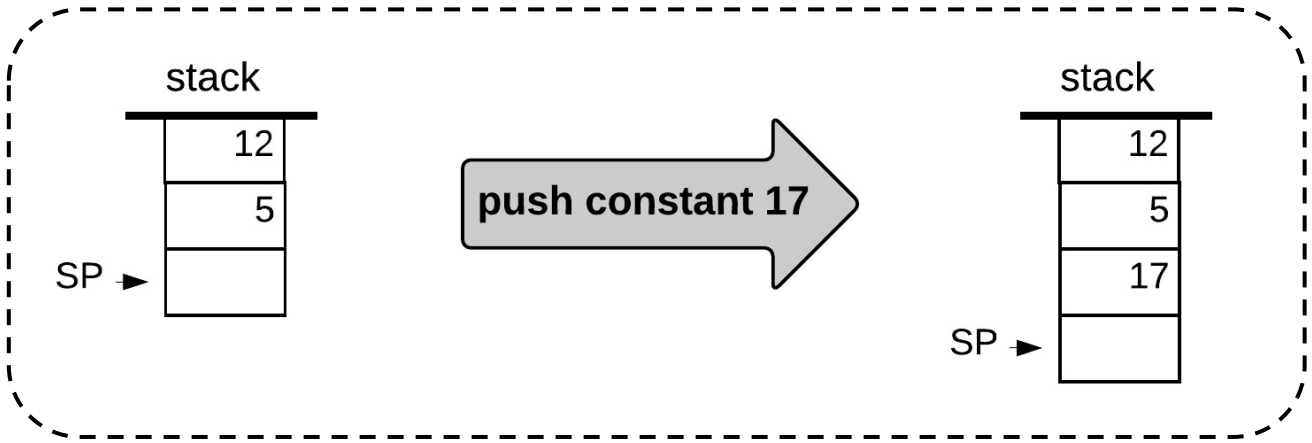
`*p` // the memory location that `p` points at

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

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

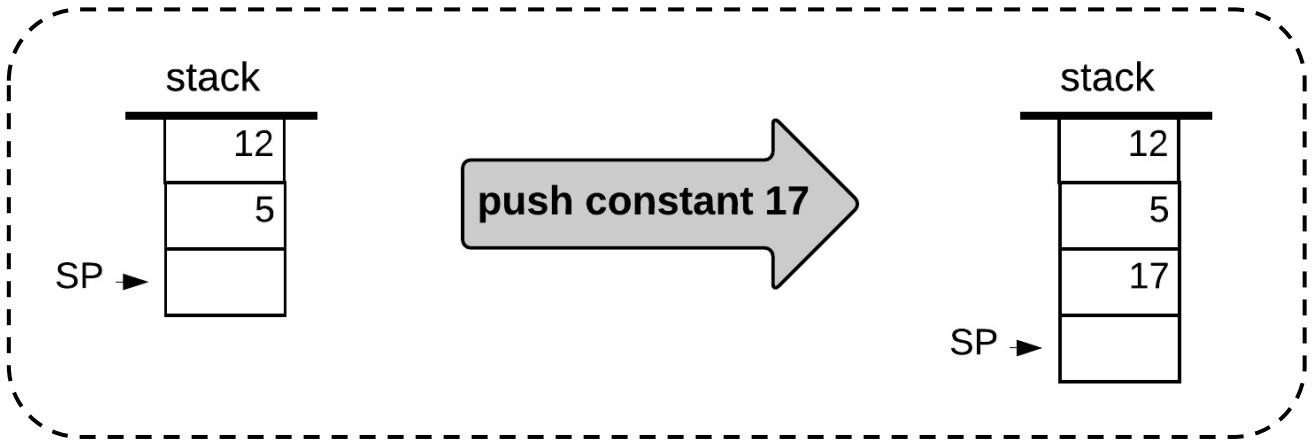
# Stack implementation

Abstraction:



# Stack implementation

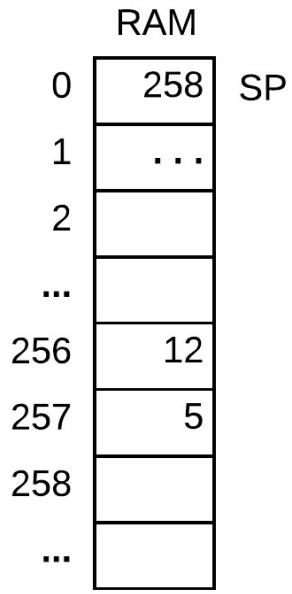
## Abstraction:




## Implementation:

Assumptions:

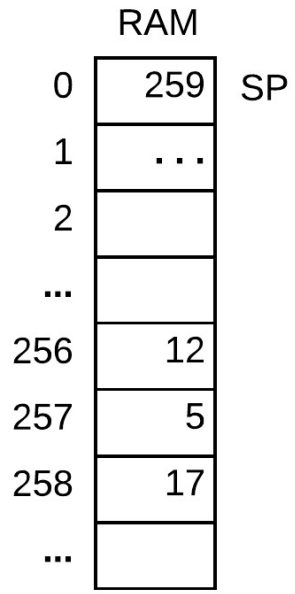
- `SP` stored in `RAM[0]`
- Stack base addr = `256`




Logic:

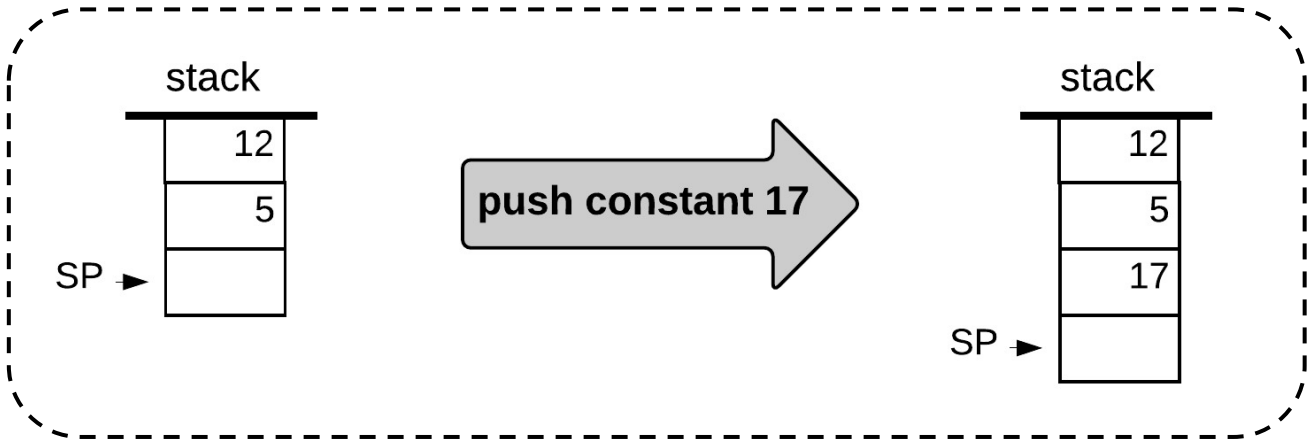
```
*SP = 17
SP++
```

Hack assembly:



# Stack implementation

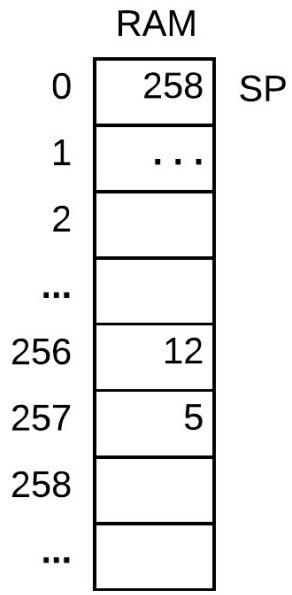
## Abstraction:




## Implementation:

Assumptions:

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

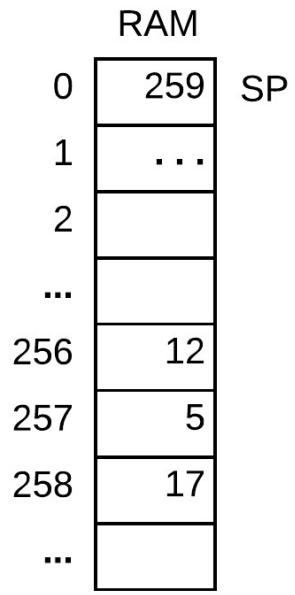



Logic:

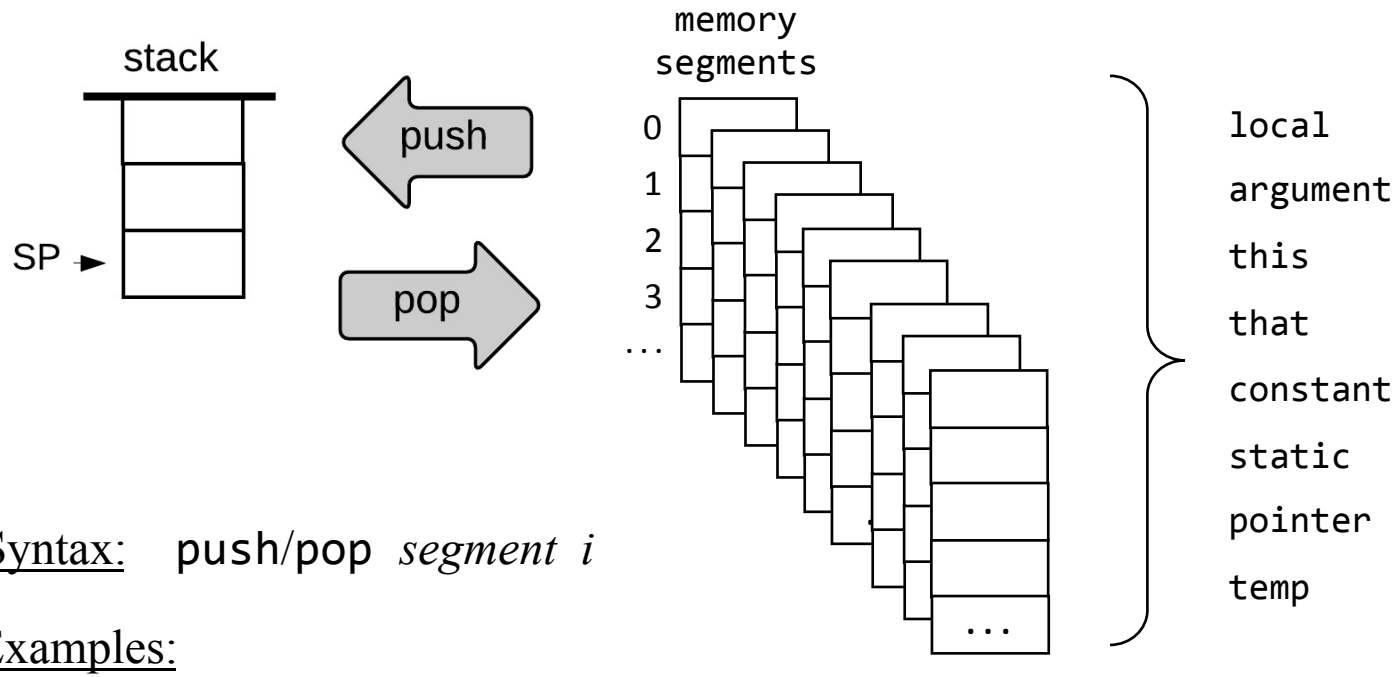
```
*SP = 17
SP++
```

Hack assembly:

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



# Memory segments (abstraction)

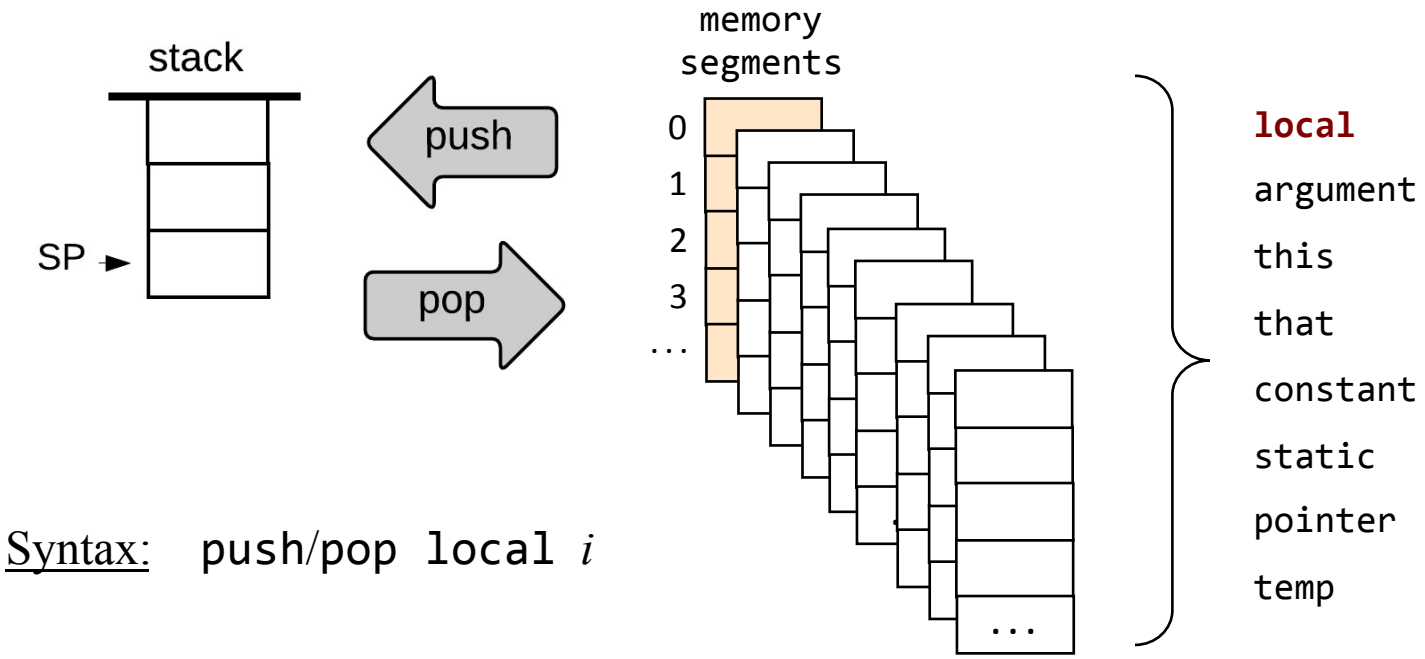



Syntax: `push/pop` *segment i*

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`

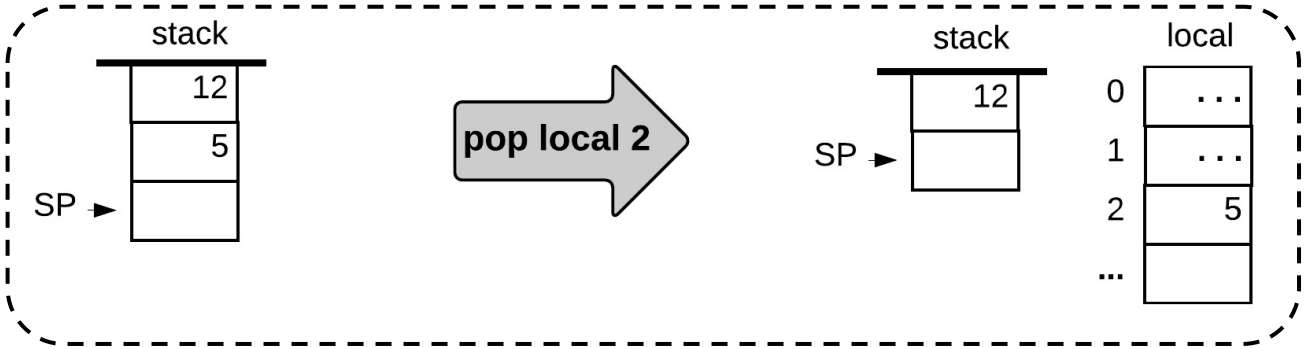
Why do we need a `local` segment?

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* (example)

## Abstraction:

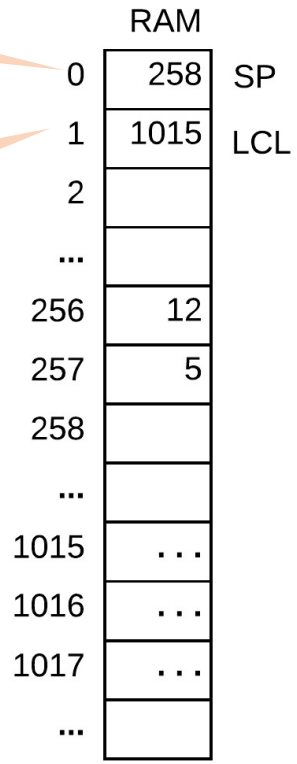



## Implementation:

stack pointer

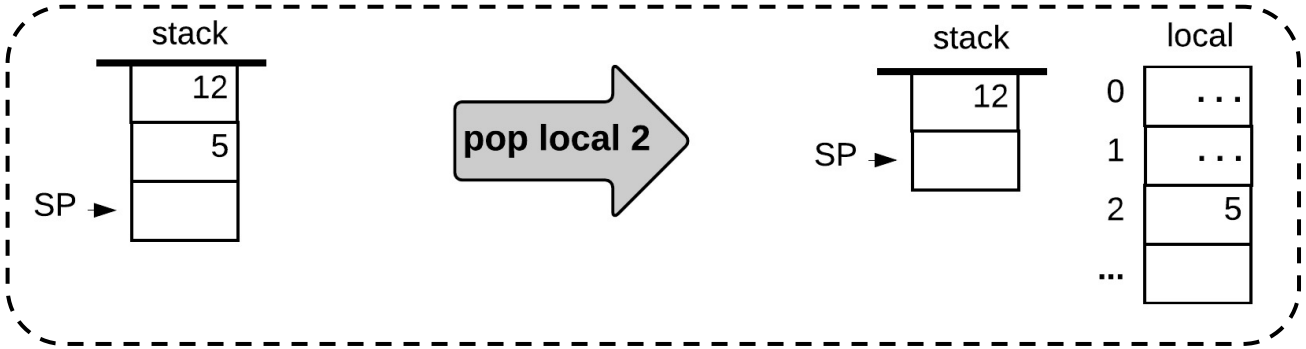
base address of the `local` segment

the `local` segment is stored some-where in the RAM



# Implementing `pop local` *i* (example)

## Abstraction:

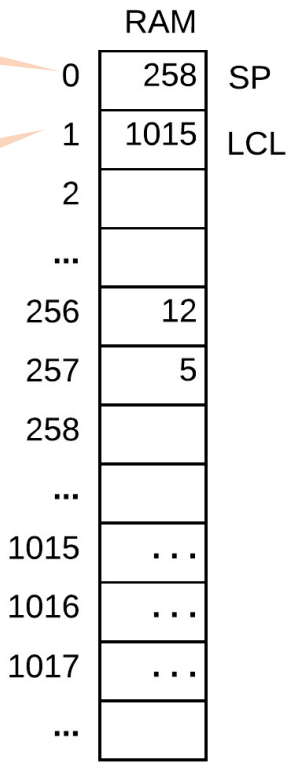



stack pointer

base address of the `local` segment

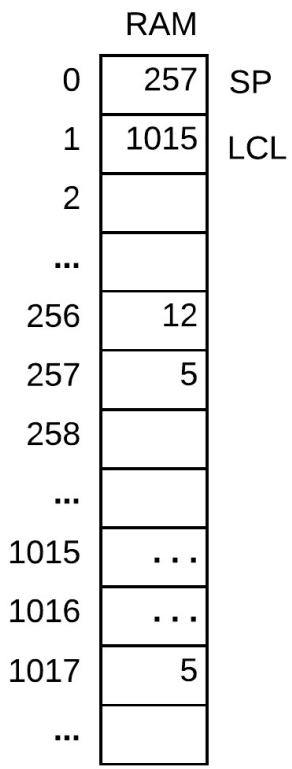
## Implementation:

the `local` segment is stored somewhere in the RAM




Implementation:

```
addr=LCL+2, SP--, *addr=*SP
```



# Implementing `pop local` *i*

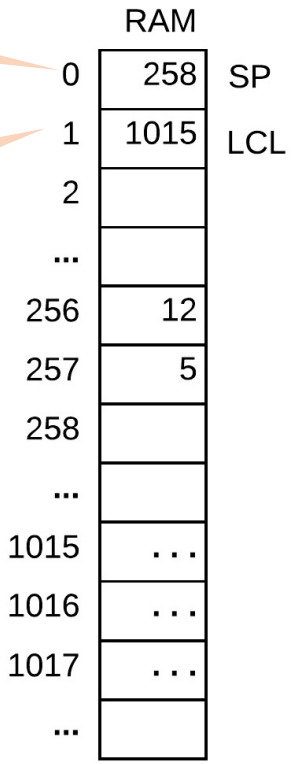
## Abstraction:


stack pointer

base address of the `local` segment

## Implementation:

the `local` segment is stored some-where in the RAM

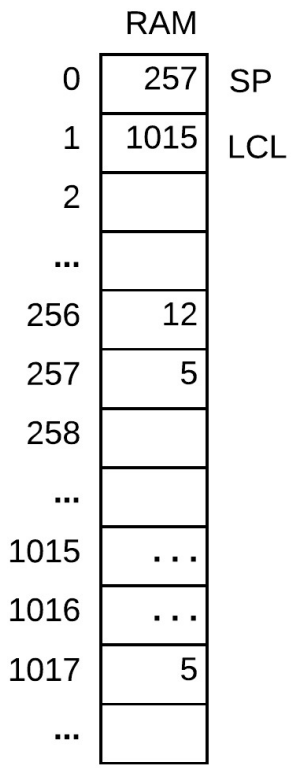



Implementation:

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

Hack assembly:

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
```

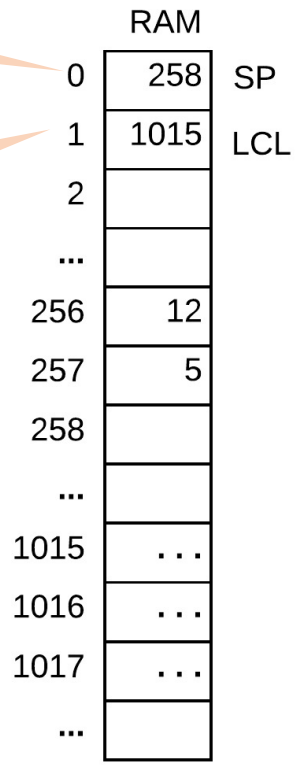
```
addr = LCL+ i, *SP = *addr, SP++
```

stack pointer

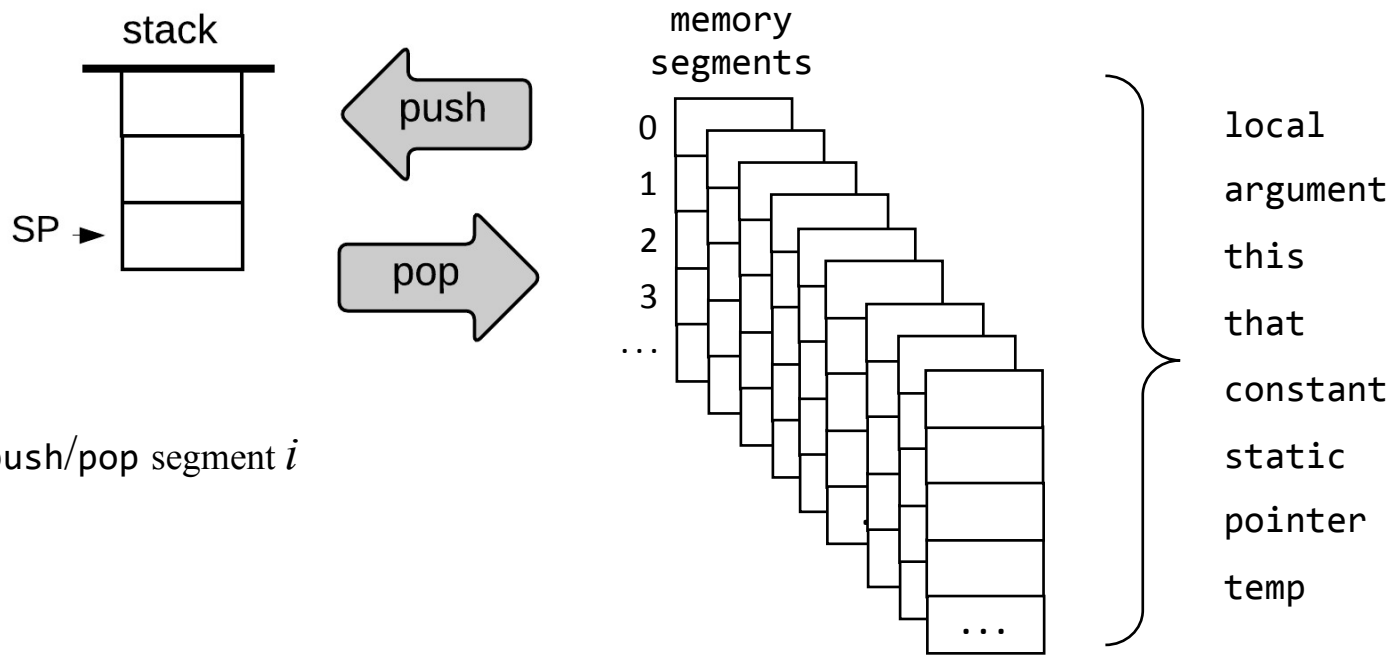
base address of the `local` segment

## Implementation:

the `local` segment is stored somewhere in the RAM



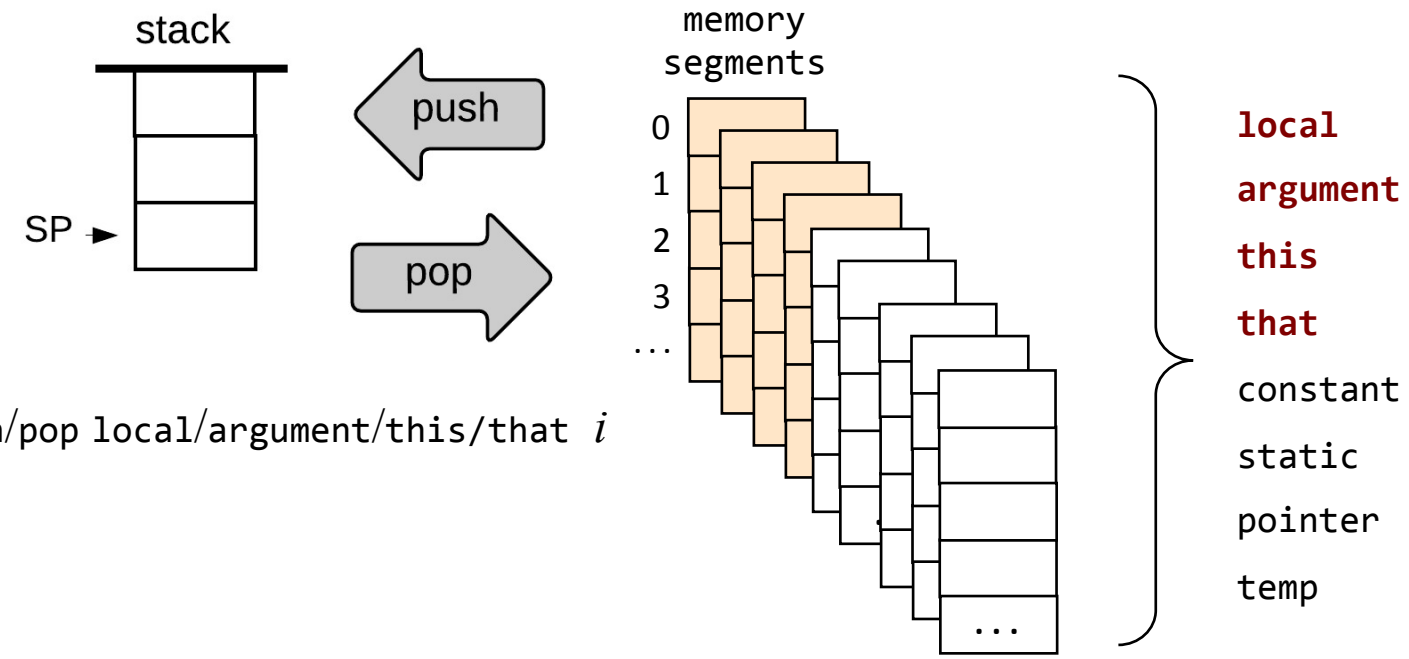
# Memory segments




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`

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

VM code:

`push segment i`

`pop segment i`

VM translator

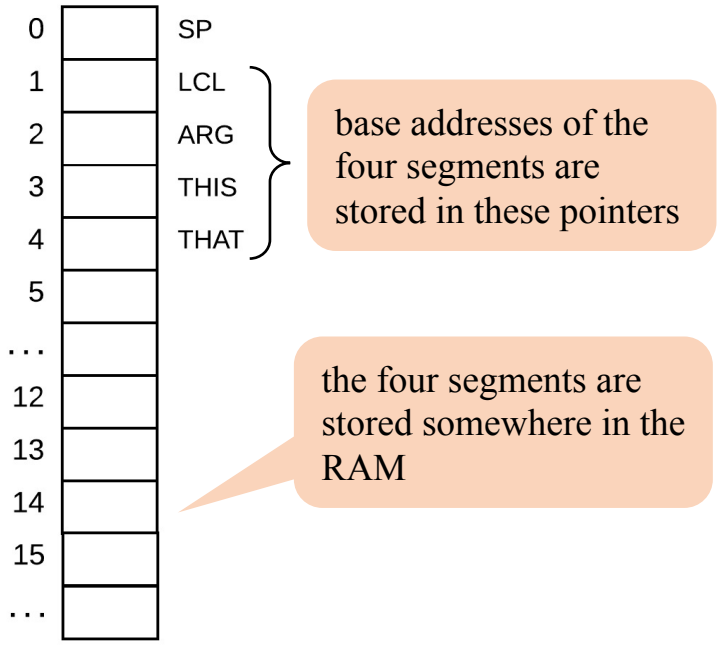
Assembly pseudo code:

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

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

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

Host RAM



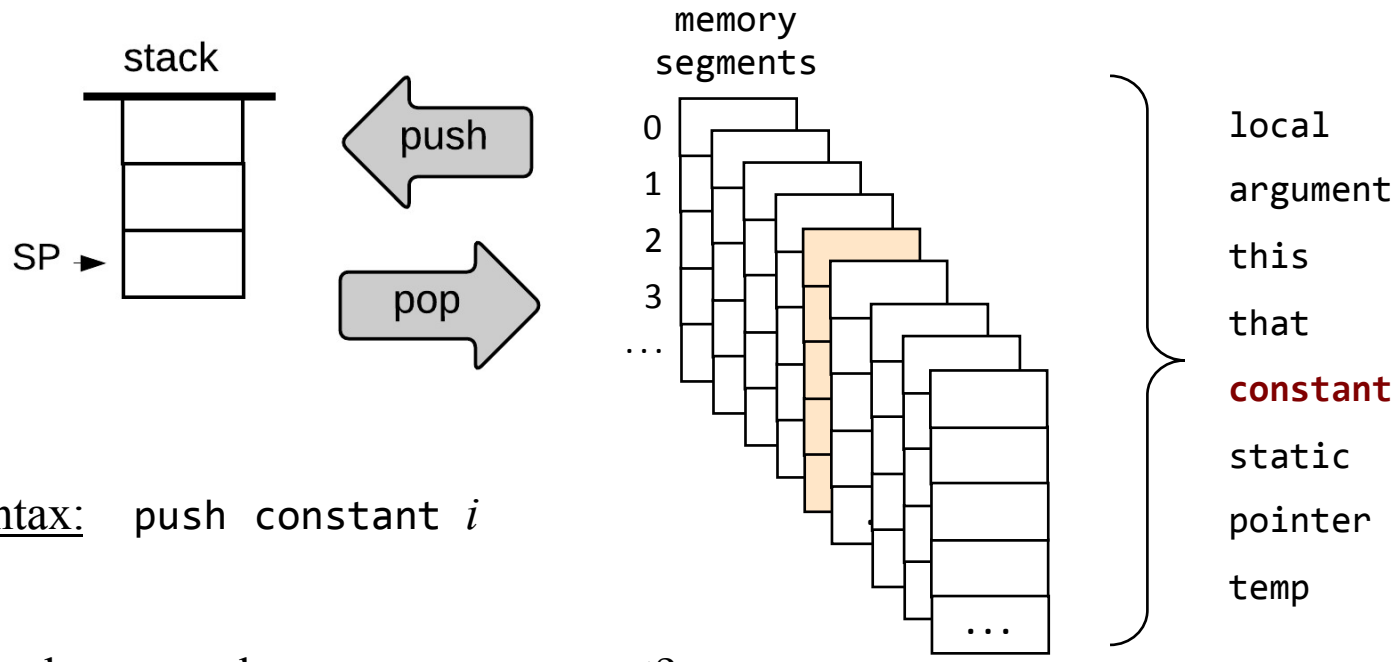

base addresses of the four segments are stored in these pointers

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

# 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:

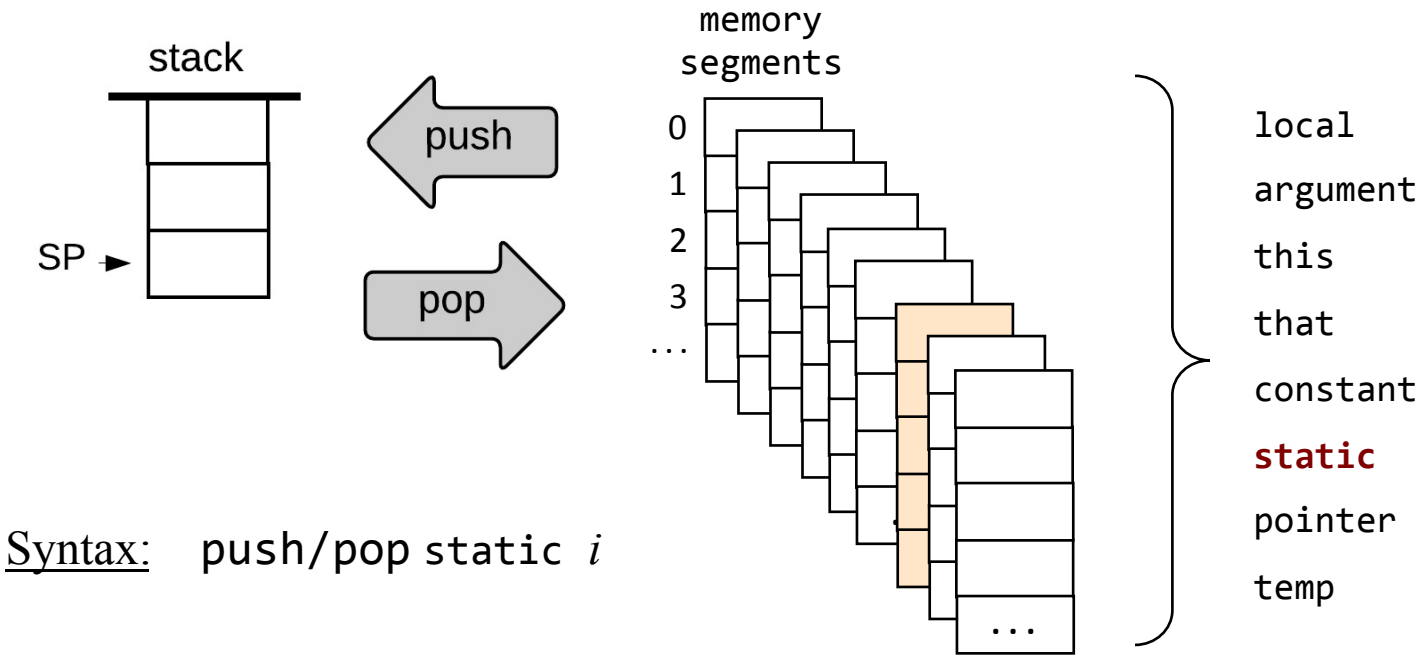
*SP = *i*, SP++

(no `pop constant` operation)

Implementation:

Supplies the specified constant

# 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
...
```


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`

# Implementing push/pop static *i*

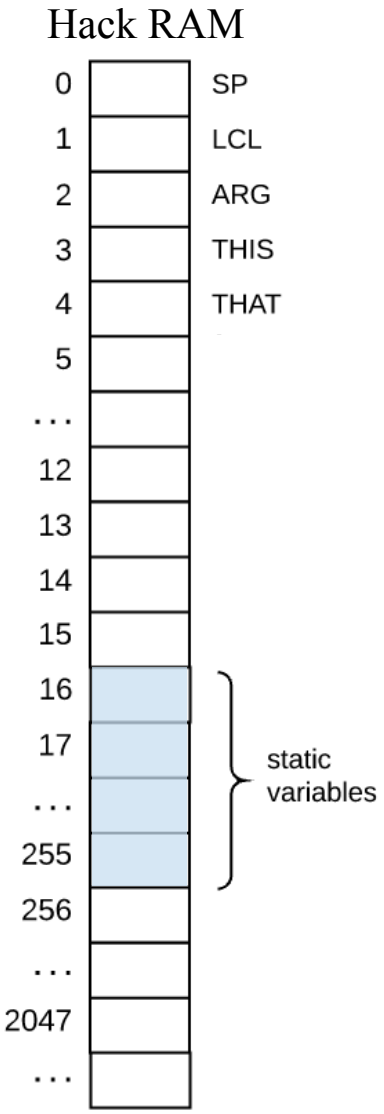
VM code:

```
// 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
...
```

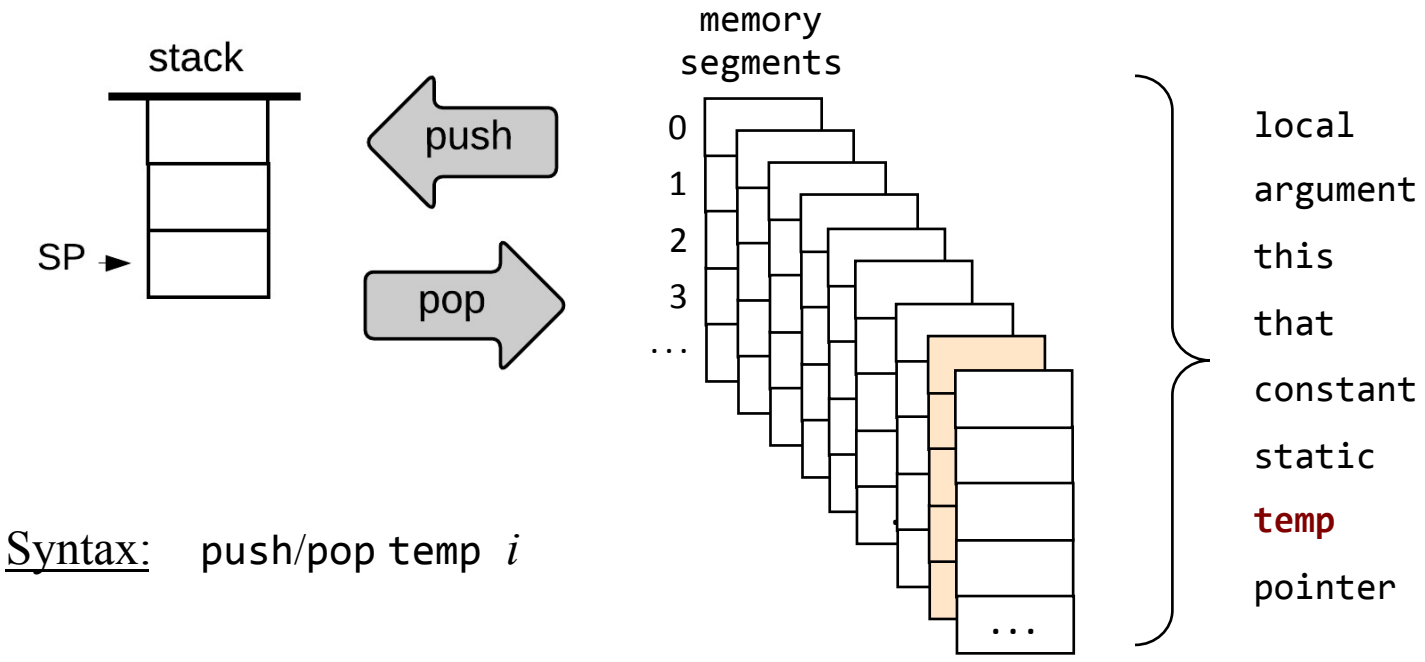



The challenge:
static variables should be seen by all the methods in a program

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.

# 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`.

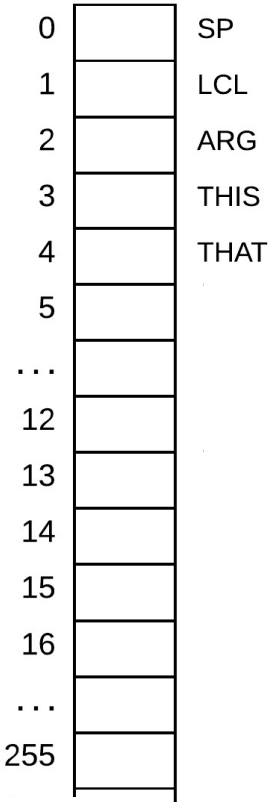
# Implementing `push/pop temp` *i*

VM code:

```
push temp i
```

```
pop temp i
```

Hack RAM



# Implementing `push/pop temp i`

VM code:

`push temp i`

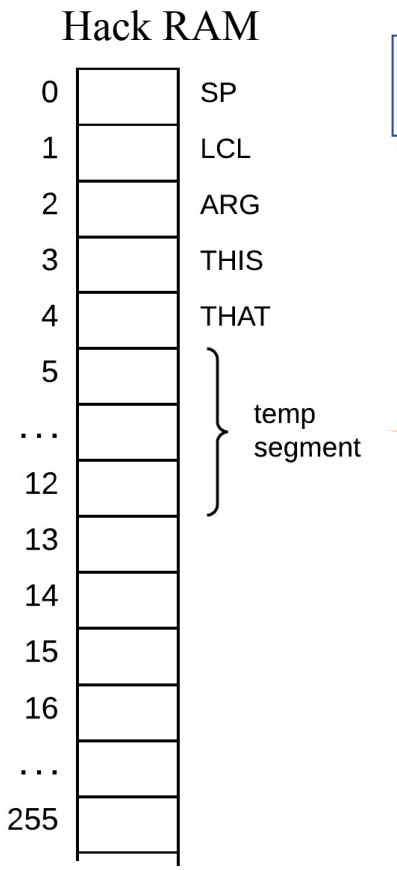
`pop temp i`

VM Translator

Assembly psuedo code:

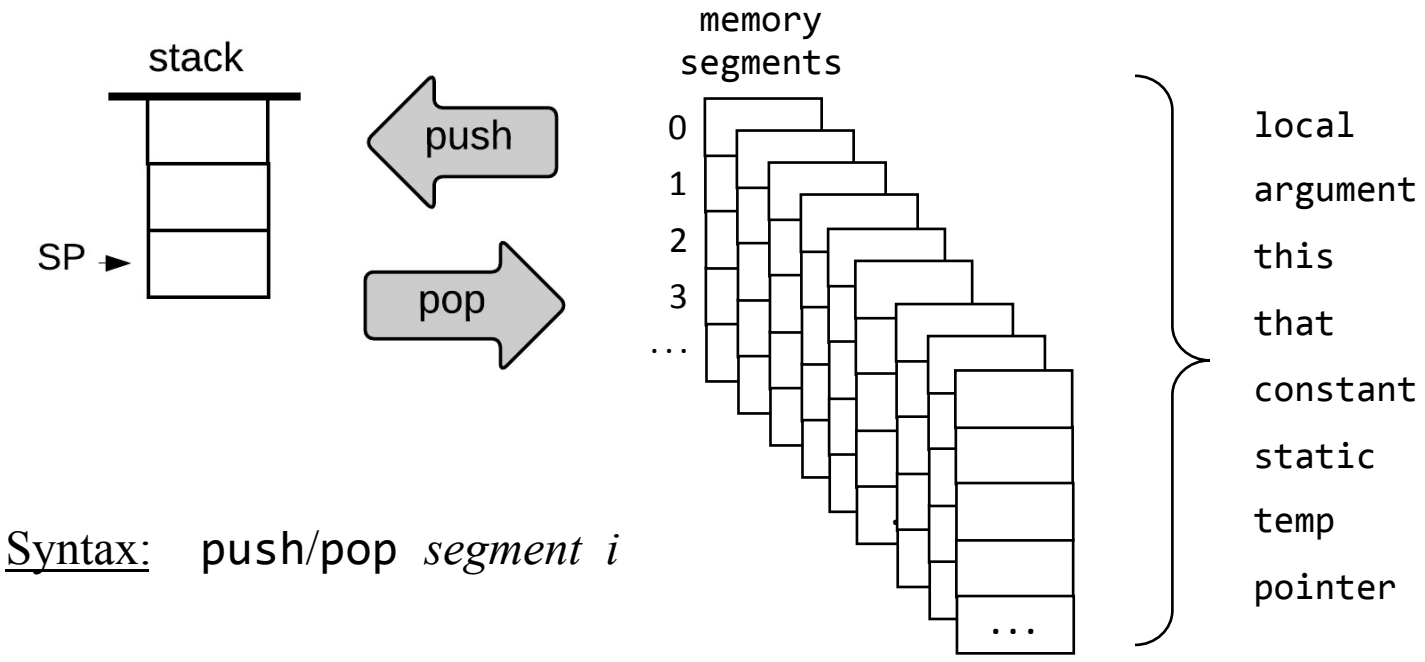
`addr = 5 + i, *SP = *addr, SP++`

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



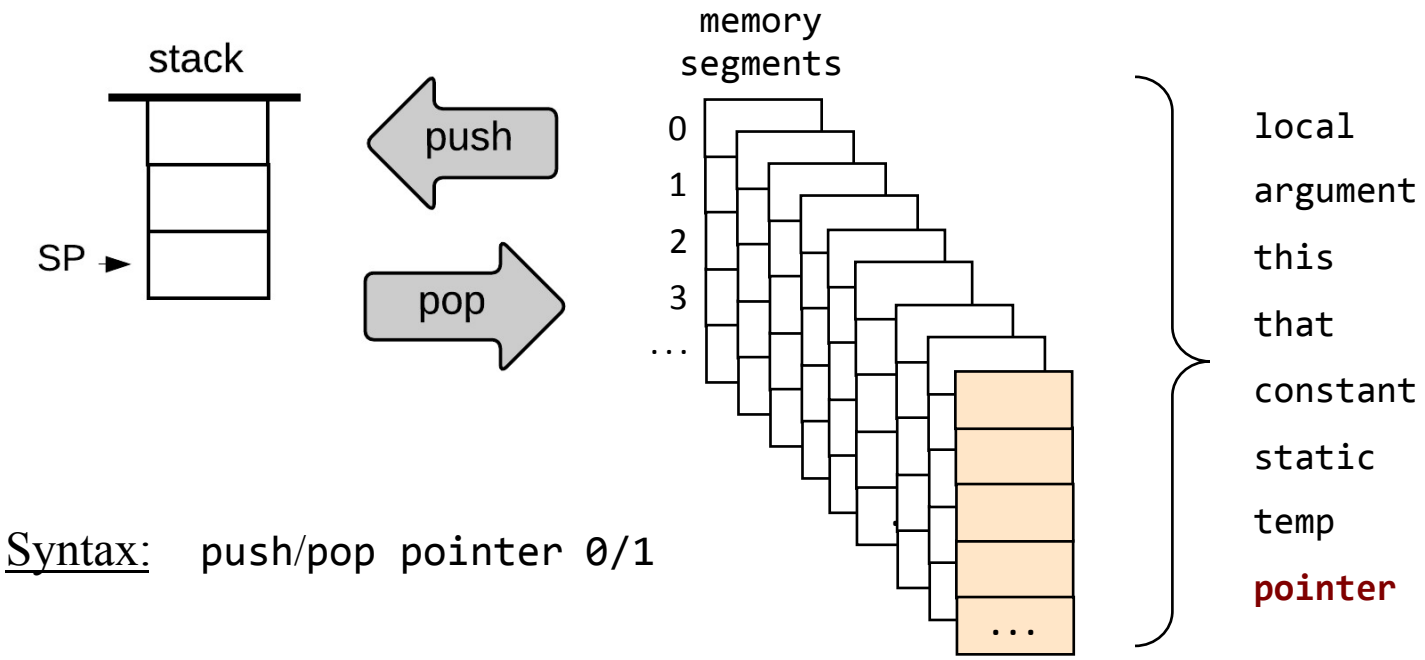

A fixed, 8-place memory segment, stored in RAM locations 5 to 12

# Memory segments




Syntax: `push/pop` *segment i*

# Implementing `push/pop pointer 0/1`

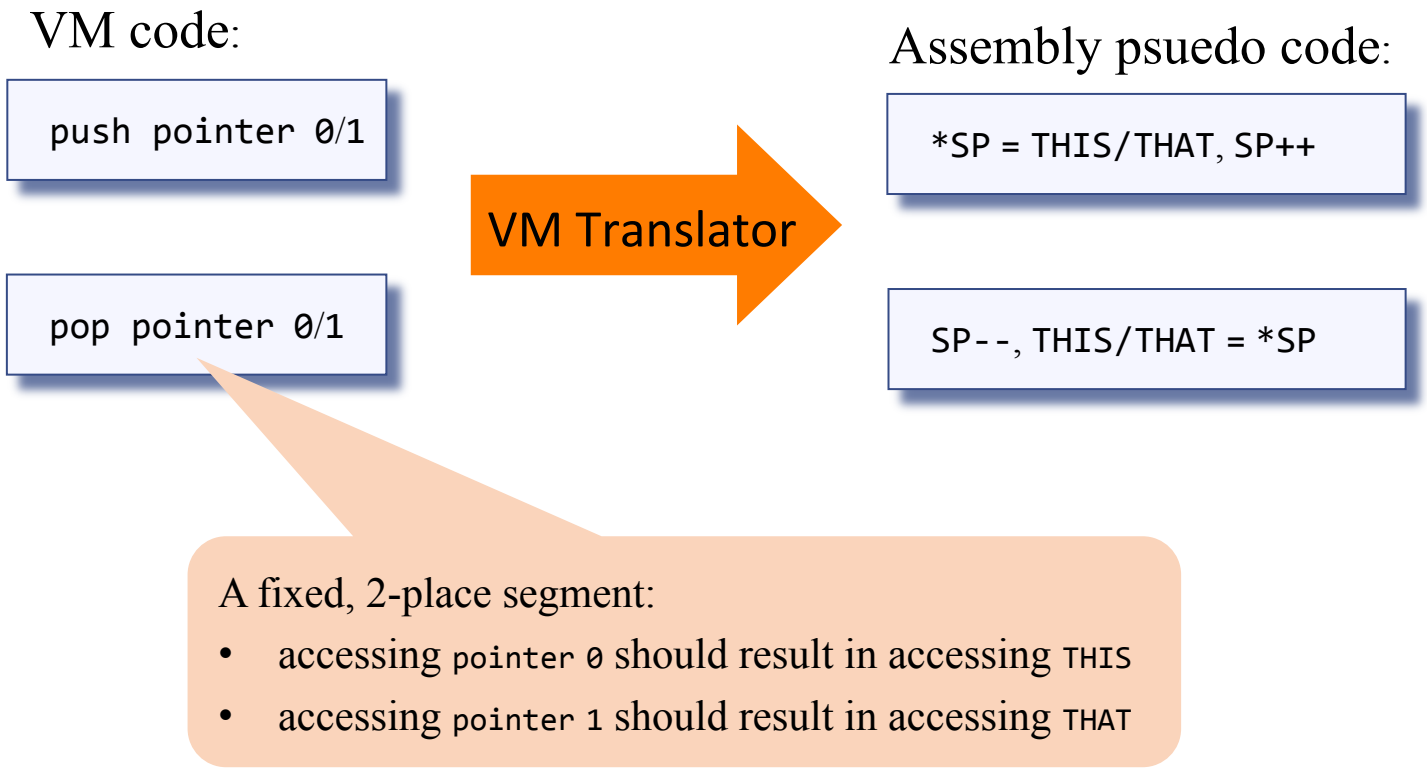



Syntax: `push/pop pointer 0/1`

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`




Implementation:

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