The Hack VM I: Structure, arithmetic and logic
COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

Stacks (reminder)

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)

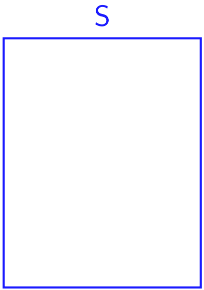
The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

S = create();

Stacks (reminder)

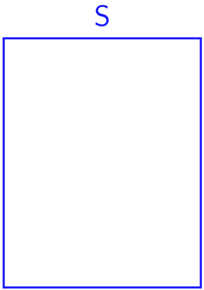


S = create();

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.



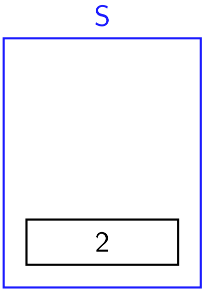
S = create();
S.push(2);

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



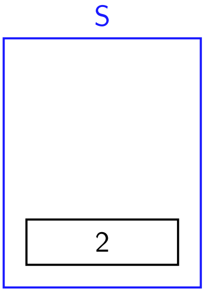
S = create();
S.push(2);

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



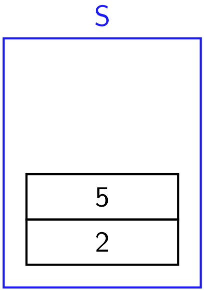
S = create();
S.push(2);
S.push(5);

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



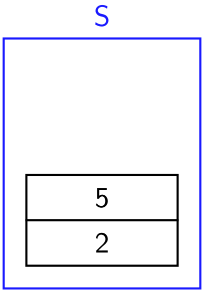
S = create();
S.push(2);
S.push(5);

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



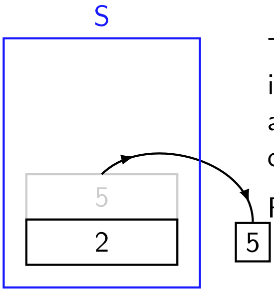
S = create();
S.push(2);
S.push(5);
S.pop();

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



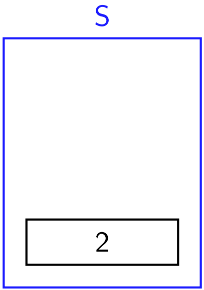
S = create();
S.push(2);
S.push(5);
S.pop();

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



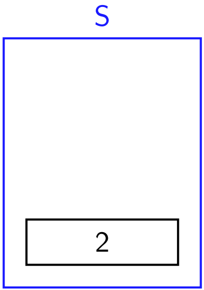
S = create();
S.push(2);
S.push(5);
S.pop();

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



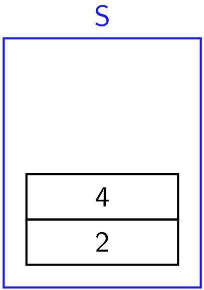
S = create();
S.push(2);
S.push(5);
S.pop();
S.push(4);

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



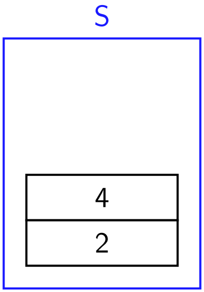
S = create();
S.push(2);
S.push(5);
S.pop();
S.push(4);

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



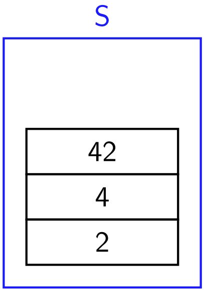

```
S = create();
S.push(2);
S.push(5);
S.pop();
S.push(4);
S.push(42);
```

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



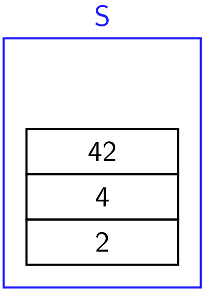
S = create();
S.push(2);
S.push(5);
S.pop();
S.push(4);
S.push(42);

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



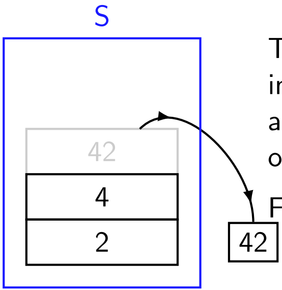
S = create();
S.push(2);
S.push(5);
S.pop();
S.push(4);
S.push(42);
S.pop();

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



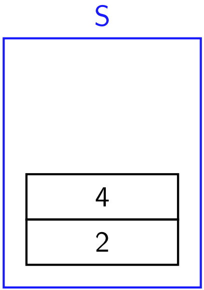
S = create();
S.push(2);
S.push(5);
S.pop();
S.push(4);
S.push(42);
S.pop();

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)



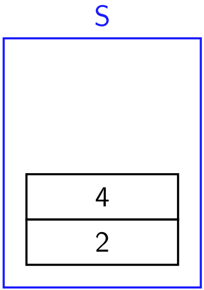
S = create();
S.push(2);
S.push(5);
S.pop();
S.push(4);
S.push(42);
S.pop();

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks (reminder)




```
S = create();
S.push(2);
S.push(5);
S.pop();
S.push(4);
S.push(42);
S.pop();
```

The Hack VM is an example of a **stack machine**, in which a stack takes the place of registers for arithmetic/logical operations and memory is used only for storage.

From COMSM1201, a stack supports operations:

- **create():** Creates a new stack.
- **push(x):** Adds x to top of the stack.
- **pop():** Removes the most recently-added piece of data from the stack and returns it.

Stacks are **LIFO**: "Last In, First Out".

(The JVM is also a stack machine, so there are good reasons to do this! We'll see these later.)

Virtual memory

In assembly, we use **physical memory** — every memory address is the exact logical signal sent to a physical latch on a physical chip, either ROM or RAM.

An intermediate representation should be portable, so instead we work with **virtual memory**. This acts like physical memory, storing one word at each address, but we don't worry about where each address is stored physically.

The (ISA-dependent) VM translator will then map virtual memory addresses to physical memory addresses during compilation.[1]

[1] In modern computers, virtual memory has a second, more important, role. Each process' assembly code assumes it's the only process running and has full access to any memory address. This is actually virtual memory. The operating system then uses dedicated machine code instructions to maintain a **page table** mapping each process' virtual memory space back to physical memory. This virtual memory may not even map to RAM — rarely-accessed data will be swapped to a **page file**.

Virtual memory

In assembly, we use **physical memory** — every memory address is the exact logical signal sent to a physical latch on a physical chip, either ROM or RAM.

An intermediate representation should be portable, so instead we work with **virtual memory**. This acts like physical memory, storing one word at each address, but we don't worry about where each address is stored physically.

The (ISA-dependent) VM translator will then map virtual memory addresses to physical memory addresses during compilation.[1]

The Hack VM has 8(!) separate virtual memory banks, which our VM translator will map to different **segments** (continuous blocks) of the underlying RAM.

For now, we only care about two:

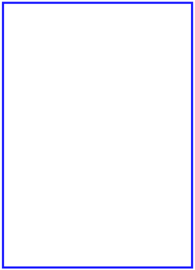
- **`local`** is general-purpose storage for local variables.
- **`constant`** holds the constant *i* at each 15-bit address *i*. This "memory" is read-only and doesn't correspond to any physical ROM or RAM.

---

[1] In modern computers, virtual memory has a second, more important, role. Each process' assembly code assumes it's the only process running and has full access to any memory address. This is actually virtual memory. The operating system then uses dedicated machine code instructions to maintain a **page table** mapping each process' virtual memory space back to physical memory. This virtual memory may not even map to RAM — rarely-accessed data will be swapped to a **page file**.

Syntax: Pushing and popping

S



`local`

⋮	
33	451
34	1138
35	0
36	0
37	13
⋮	

There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

Syntax: Pushing and popping

S

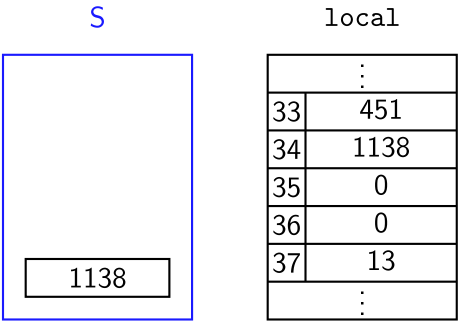
local

⋮	
33	451
34	1138
35	0
36	0
37	13
⋮	

`push local 34`

There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

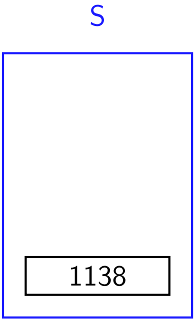


`push local 34`

There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

Syntax: Pushing and popping



local

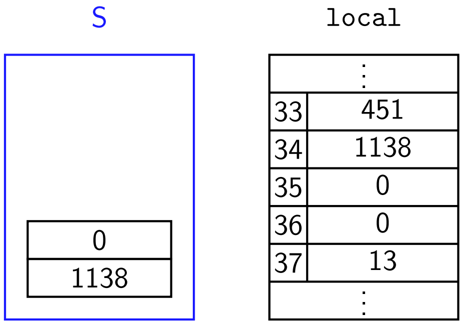
⋮	
33	451
34	1138
35	0
36	0
37	13
⋮	

```
push local 34
push local 36
```

There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

Syntax: Pushing and popping

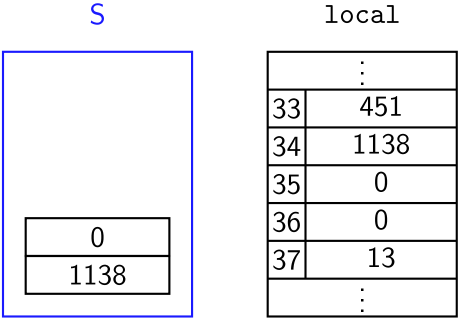



```
push local 34
push local 36
```

There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

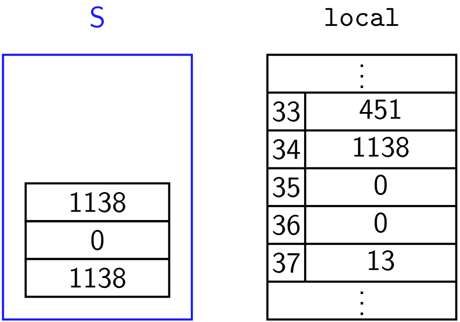
Syntax: Pushing and popping




```
push local 34
push local 36
push local 34
```

There's only one stack, so we don't create() it.

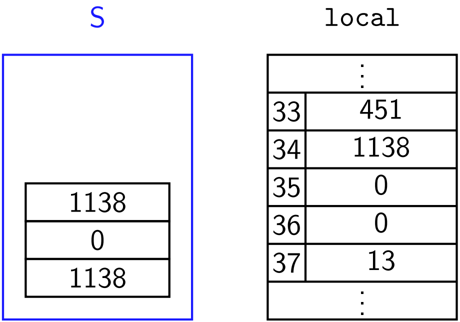
We push with the command `push [memory] [address]`.




```
push local 34
push local 36
push local 34
```

There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

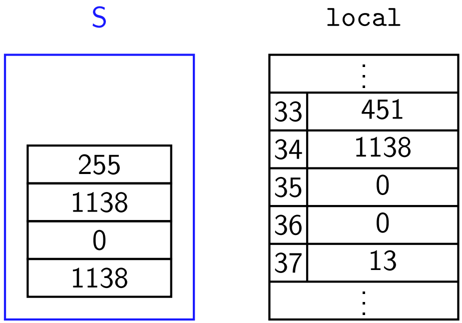



```
push local 34
push local 36
push local 34
push constant 255
```

There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

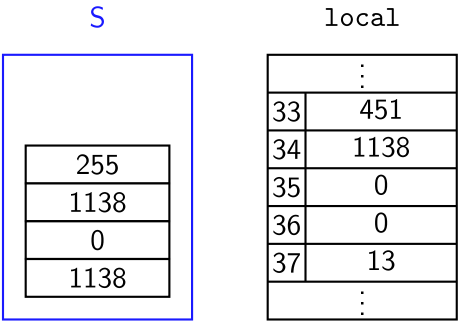
Syntax: Pushing and popping




```
push local 34
push local 36
push local 34
push constant 255
```

There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.




```
push local 34
push local 36
push local 34
push constant 255
```

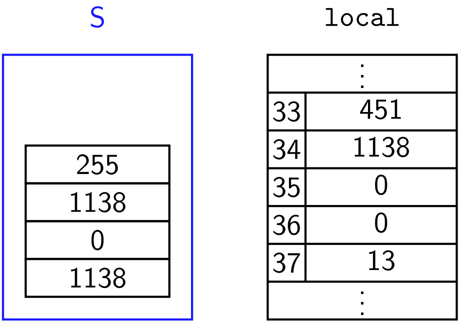
There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

We pop and store the result with the command `pop [memory] [address]`. The value we pop is then stored at that memory address.

(Note that pop by itself is not valid syntax.)

Syntax: Pushing and popping




```
push local 34
push local 36
push local 34
push constant 255
pop local 35
```

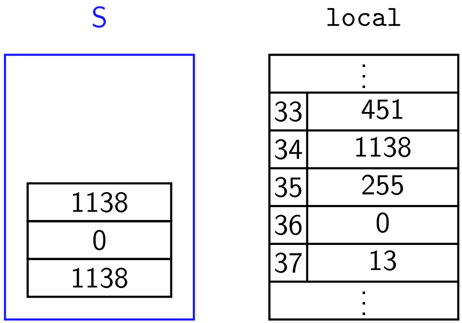
There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

We pop and store the result with the command `pop [memory] [address]`. The value we pop is then stored at that memory address.

(Note that `pop` by itself is not valid syntax.)

Syntax: Pushing and popping



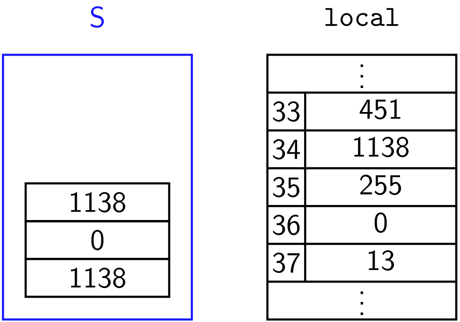

```
push local 34
push local 36
push local 34
push constant 255
pop local 35
```

There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

We pop and store the result with the command `pop [memory] [address]`. The value we pop is then stored at that memory address.

(Note that `pop` by itself is not valid syntax.)




```
push local 34
push local 36
push local 34
push constant 255
pop local 35
pop local 33
```

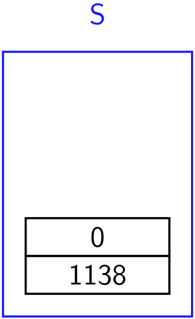
There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

We pop and store the result with the command `pop [memory] [address]`. The value we pop is then stored at that memory address.

(Note that `pop` by itself is not valid syntax.)

Syntax: Pushing and popping



local

	⋮
33	1138
34	1138
35	255
36	0
37	13
	⋮

```
push local 34
push local 36
push local 34
push constant 255
pop local 35
pop local 33
```

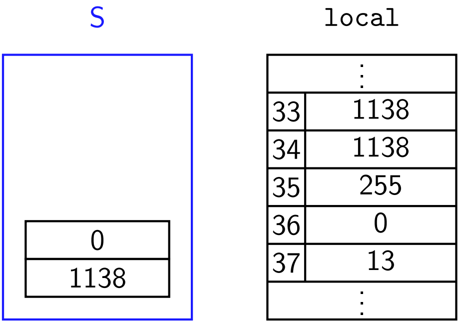
There's only one stack, so we don't create() it.

We push with the command `push [memory] [address]`.

We pop and store the result with the command `pop [memory] [address]`. The value we pop is then stored at that memory address.

(Note that `pop` by itself is not valid syntax.)

Syntax: Pushing and popping




```
push local 34
push local 36
push local 34
push constant 255
pop local 35
pop local 33
```

Note that pushing and popping are our *only* form of memory management. For example, to copy the value of `local 4` into `local 250`, we would write:

```
push local 4
pop local 250
```

Stack operations in Hack VM

S

`local`

⋮	
48	1138
49	1138
50	255
51	0
52	13
⋮	

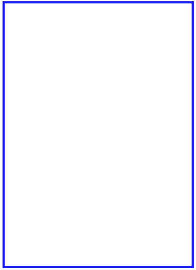
All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM

S



`local`

⋮	
48	1138
49	1138
50	255
51	0
52	13
⋮	

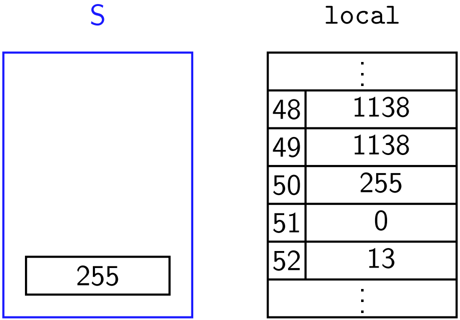
```
push local 50
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



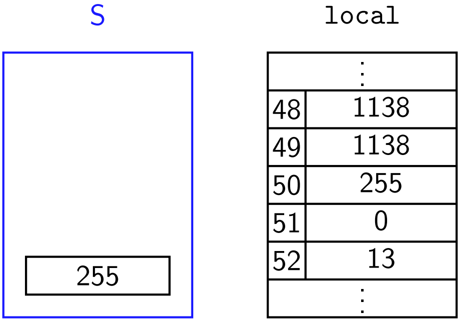

```
push local 50
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



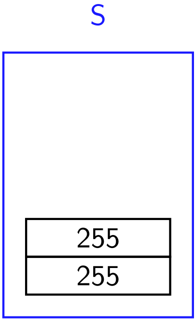

```
push local 50
push local 50
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



`local`

	⋮
48	1138
49	1138
50	255
51	0
52	13
	⋮

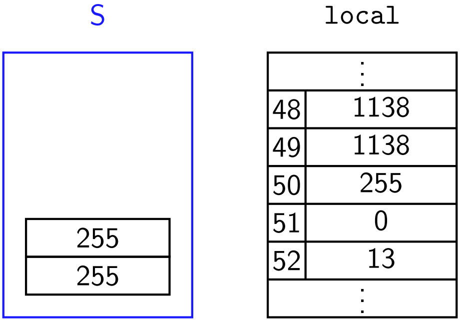
```
push local 50
push local 50
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



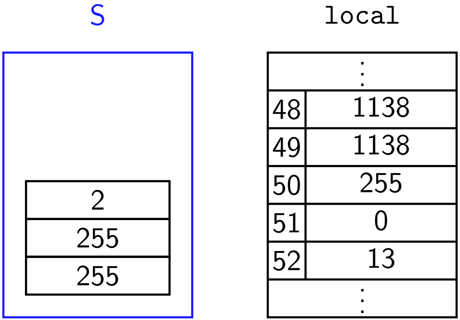

```
push local 50
push local 50
push constant 2
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



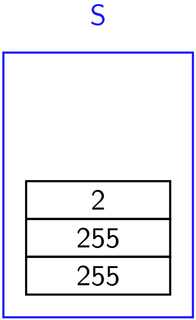

```
push local 50
push local 50
push constant 2
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



`local`

	⋮
48	1138
49	1138
50	255
51	0
52	13
	⋮

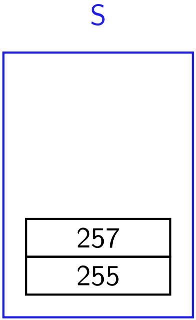
```
push local 50
push local 50
push constant 2
add
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



`local`

	⋮
48	1138
49	1138
50	255
51	0
52	13
	⋮

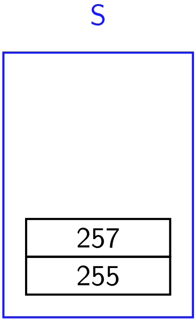
```
push local 50
push local 50
push constant 2
add
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



local

	⋮
48	1138
49	1138
50	255
51	0
52	13
	⋮

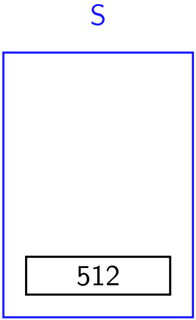
```
push local 50
push local 50
push constant 2
add
add
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



`local`

⋮	
48	1138
49	1138
50	255
51	0
52	13
⋮	

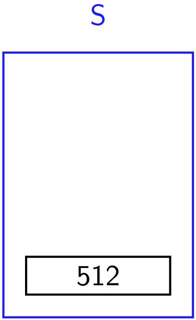
```
push local 50
push local 50
push constant 2
add
add
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



local

⋮	
48	1138
49	1138
50	255
51	0
52	13
⋮	

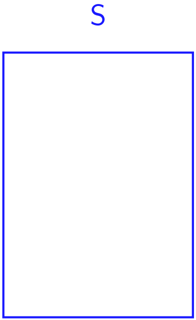
```
push local 50
push local 50
push constant 2
add
add
pop local 50
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM



`local`

	⋮
48	1138
49	1138
50	512
51	0
52	13
	⋮

```
push local 50
push local 50
push constant 2
add
add
pop local 50
```

All arithmetic operations in Hack VM happen via the stack. For example, to add two numbers, we use the `add` command (which has no arguments).

`add` pops the top two values of the stack, adds them together, and then pushes the result back onto the stack.

Other operations use exactly the same approach and syntax (see next slide). We can't add two values directly from `local` — we must push them to the stack first.

Stack operations in Hack VM

S

`local`

	⋮
48	1138
49	1138
50	512
51	0
52	13
	⋮

Logical comparisons work the same way. For example, the `eq` command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

Stack operations in Hack VM

S

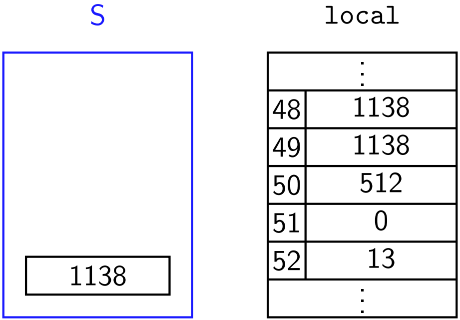
local

	⋮
48	1138
49	1138
50	512
51	0
52	13
	⋮

```
push local 48
```

Logical comparisons work the same way. For example, the `eq` command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

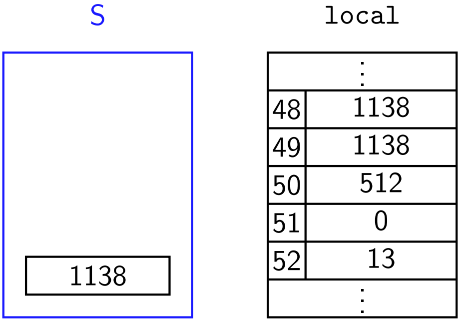


`push local 48`

Logical comparisons work the same way. For example, the `eq` command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

Stack operations in Hack VM

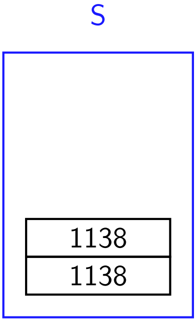



```
push local 48
push local 49
```

Logical comparisons work the same way. For example, the eq command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

Stack operations in Hack VM



`local`

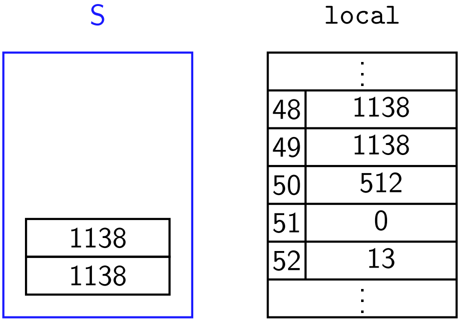
	⋮
48	1138
49	1138
50	512
51	0
52	13
	⋮

```
push local 48
push local 49
```

Logical comparisons work the same way. For example, the `eq` command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

Stack operations in Hack VM

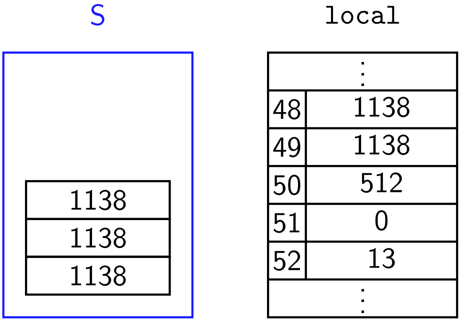



```
push local 48
push local 49
push constant 1138
```

Logical comparisons work the same way. For example, the eq command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

Stack operations in Hack VM

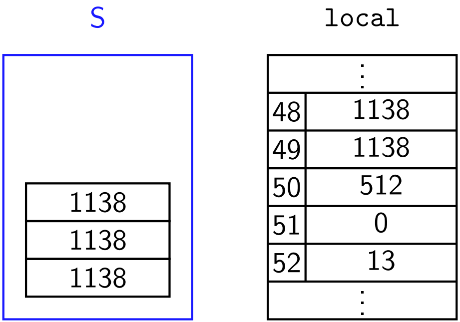



```
push local 48
push local 49
push constant 1138
```

Logical comparisons work the same way. For example, the eq command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

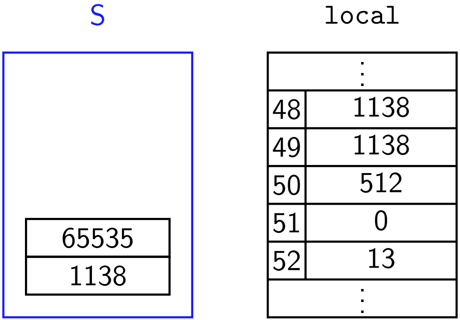
Stack operations in Hack VM




```
push local 48
push local 49
push constant 1138
eq
```

Logical comparisons work the same way. For example, the eq command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

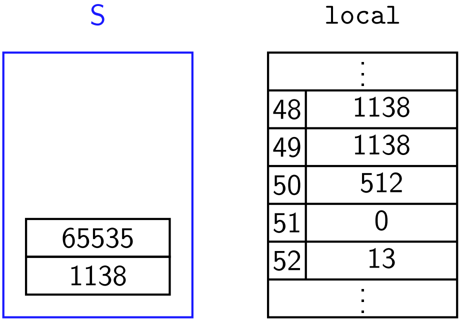



```
push local 48
push local 49
push constant 1138
eq
```

Logical comparisons work the same way. For example, the eq command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

Stack operations in Hack VM

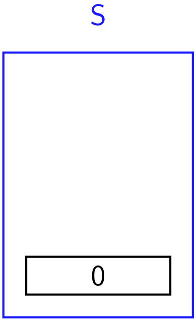



```
push local 48
push local 49
push constant 1138
eq
eq
```

Logical comparisons work the same way. For example, the eq command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

Stack operations in Hack VM



`local`

	⋮
48	1138
49	1138
50	512
51	0
52	13
	⋮

```
push local 48
push local 49
push constant 1138
eq
eq
```

Logical comparisons work the same way. For example, the eq command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

Stack operations in Hack VM

S

0

local

	⋮
48	1138
49	1138
50	512
51	0
52	13
	⋮

```
push local 48
push local 49
push constant 1138
eq
eq
pop local 50
```

Logical comparisons work the same way. For example, the eq command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

Stack operations in Hack VM

S

`local`

⋮	
48	1138
49	1138
50	0
51	0
52	13
⋮	

```
push local 48
push local 49
push constant 1138
eq
eq
pop local 50
```

Logical comparisons work the same way. For example, the eq command pops the top two values of the stack, checks whether they're equal, then pushes the result back onto the stack.

The Hack VM represents a result of `true` by 0xFFFF, and `false` as 0x0000. (We'll talk more about how the Hack VM uses the results of comparisons later.)

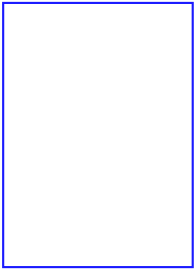
Quick reference

Command	Pops	Computes	Comment
`add`	2 values	$x + y$	Integer addition
`sub`	2 values	$x - y$	Integer subtraction
`neg`	1 value	$-y$	Arithmetic negation
`and`	2 values	$x \& y$	Bitwise AND
`or`	2 values	$x \mid y$	Bitwise OR
`not`	1 value	$\neg y$	Bitwise NOT
`eq`	2 values	$x == y$	Test equality
`gt`	2 values	$x > y$	Test greater than
`lt`	2 values	$x < y$	Test less than

- For operations that pop two values, y is the first value popped and x is the second value. E.g. `push constant 3`, `push constant 1`, `sub` will end with 2 on top of the stack rather than -2.
- All arithmetic uses twos complement, so e.g. $-x = \neg x + 1$.
- All logic writes `true` as 0xFFFF and `false` as 0x0000.
(This means bitwise operations double as logical operations!)
- You will have this table as a reference in the exam.

Example: Arithmetic

S



local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

S

local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

"Test `local 3` > 42"

"Test `local 3` $<$ `local 5` $- 100$"

`and`

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)

Example: Arithmetic

S

local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

"Test `local 3` > 42"

"Test `local 3` $<$ `local 5` $- 100$"

`and`

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.

Example: Arithmetic

S

local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
```

"Test `local 3` $<$ `local 5` $- 100$"

`and`

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.

Example: Arithmetic

S

local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
```

"Test `local 3` $<$ `local 5` $- 100$"

```
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

S

`local`

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
```
"Calculate `local 5` $- 100$"

```
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic

S

local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic

S

local

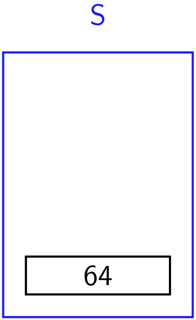
0	0
1	0
2	0
3	64
4	0
5	256
	⋮

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic



local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic

S

64

`local`

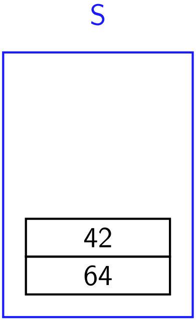
0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic



local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic

S

42
64

local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

S

65535

local

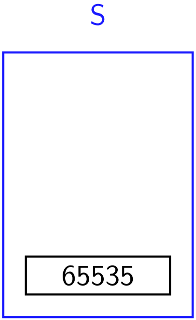
0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic



local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic

S

64
65535

local

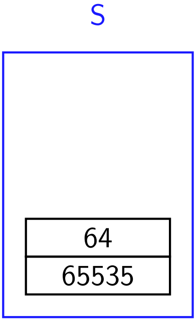
0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic



`local`

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic

S

256
64
65535

local

0	0
1	0
2	0
3	64
4	0
5	256
	⋮

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` $>$ 42) `and` (`local 3` $<$ `local 5` $-$ 100):

- Test `local 3` $>$ 42 and `local 3` $<$ `local 5` $-$ 100 separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` $>$ 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $-$ 100, first push `local 3`. Then calculate `local 5` $-$ 100 and leave it on the stack. Then `lt`.

Example: Arithmetic

S

256
64
65535

local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic

S

100
256
64
65535

local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic

S

100
256
64
65535

local

0	0
1	0
2	0
3	64
4	0
5	256
	⋮

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic

S
156
64
65535

local	
0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

S

156
64
65535

local

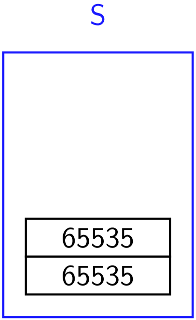
0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` $>$ 42) `and` (`local 3` $<$ `local 5` $-$ 100):

- Test `local 3` $>$ 42 and `local 3` $<$ `local 5` $-$ 100 separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` $>$ 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $-$ 100, first push `local 3`. Then calculate `local 5` $-$ 100 and leave it on the stack. Then `lt`.

Example: Arithmetic



local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic

S

65535
65535

local

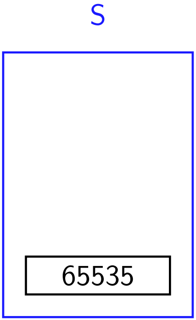
0	0
1	0
2	0
3	64
4	0
5	256
	⋮

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.

Example: Arithmetic



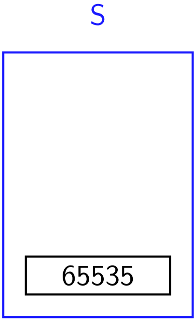
local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

For example, to test (`local 3` > 42) `and` (`local 3` $<$ `local 5` $- 100$):

- Test `local 3` > 42 and `local 3` $<$ `local 5` $- 100$ separately, then `and` them. (Remember `and`, `or` and `not` are *both* bitwise *and* logical operations!)
- To test `local 3` > 42, push `local 3` and 42, then `gt` them.
- To test `local 3` $<$ `local 5` $- 100$, first push `local 3`. Then calculate `local 5` $- 100$ and leave it on the stack. Then `lt`.



local

0	0
1	0
2	0
3	64
4	0
5	256
⋮	

```
push local 3
push constant 42
gt
push local 3
push local 5
push constant 100
sub
lt
and
```

In the workshop this week you'll see how to convert a general arithmetic expression into "stack order", a.k.a. "Reverse Polish Notation", using trees and grammars. (This is non-examinable!)

For now, it's enough to see how it's possible.

[See video for a demonstration on the VM emulator.]