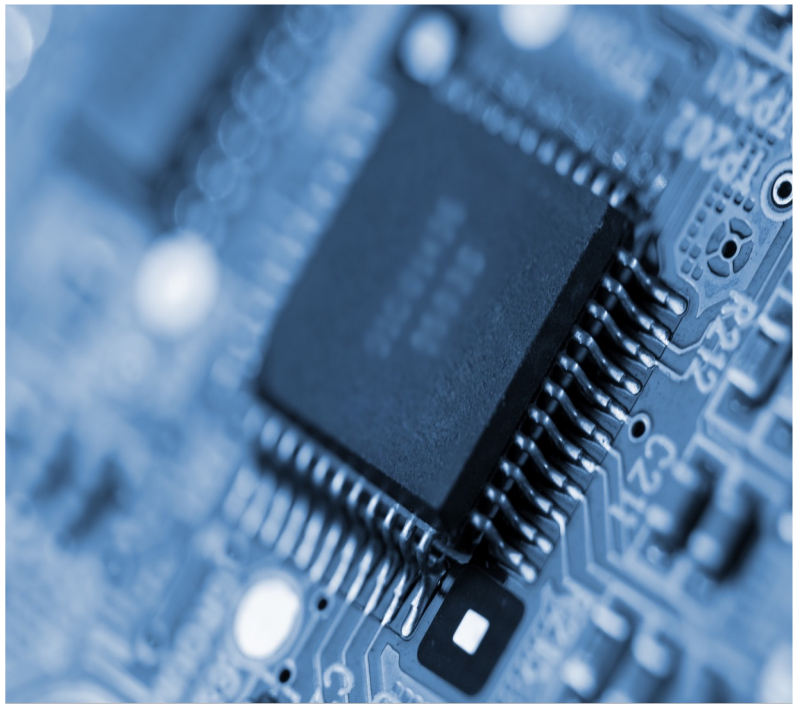
UNIVERSITY OF LEEDS

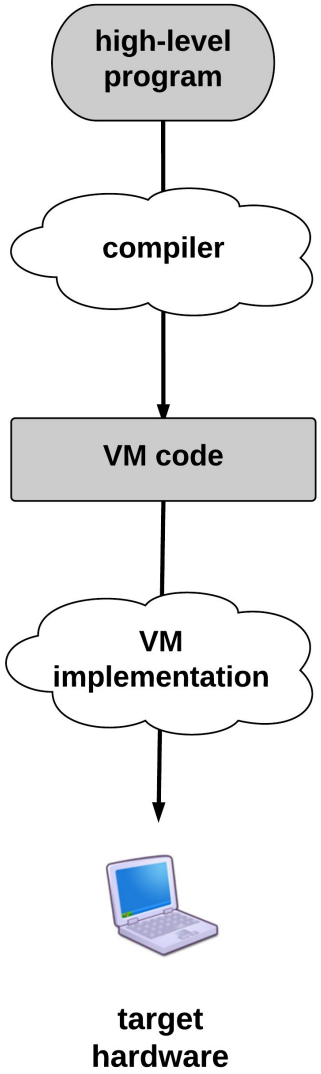
Computer Processors

Virtual Machines:
Stack Machine Model



This lecture is based on the excellent course *Nand to Tetris* by Noam Nisam and Shimon Schocken, and we reuse here many of the slides provided at www.nand2tetris.org

VM and 2-tier compilation



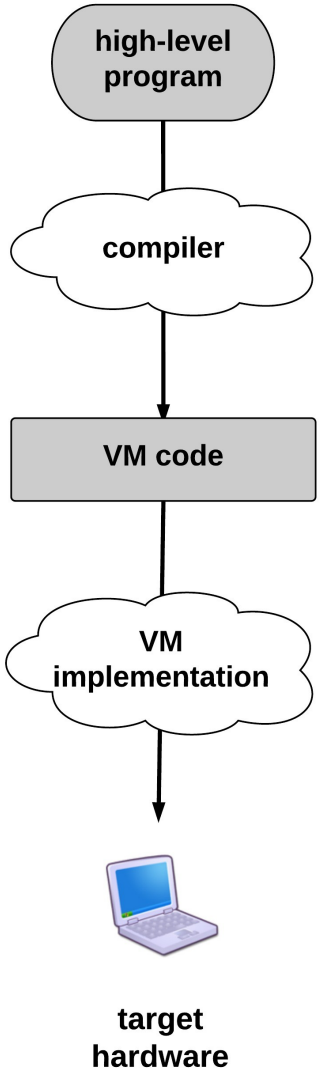
Two stages:

1. Translate high-level programs into an intermediate code (VM-code)

2. Translate VM-code into machine language

VM code is designed to run on a Virtual Machine (rather than on a real computer)
→ VM is an abstract computer that can be “realized” on other computer platforms.

VM and 2-tier compilation



Two stages:

1. Translate high-level programs into an intermediate code (VM-code)

2. Translate VM-code into machine language

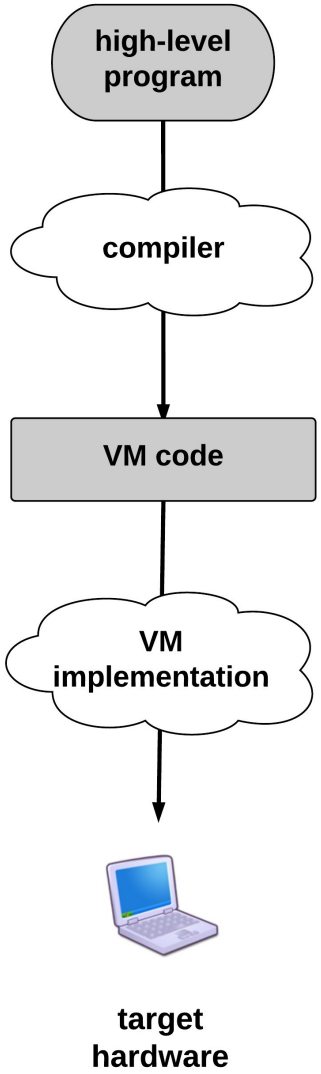
Advantages:

- VM may be implemented with relative ease on multiple target platforms

- VM-based software can run on many processors and operating systems without having to modify the original source code

The VM implementations can be realized in several ways, by software interpreters, by special-purpose hardware, or by ***translating the VM programs into the machine language of the target platform (this is what we are going to do).***

VM and 2-tier compilation



Two stages:

1. Translate high-level programs into an intermediate code (VM-code)

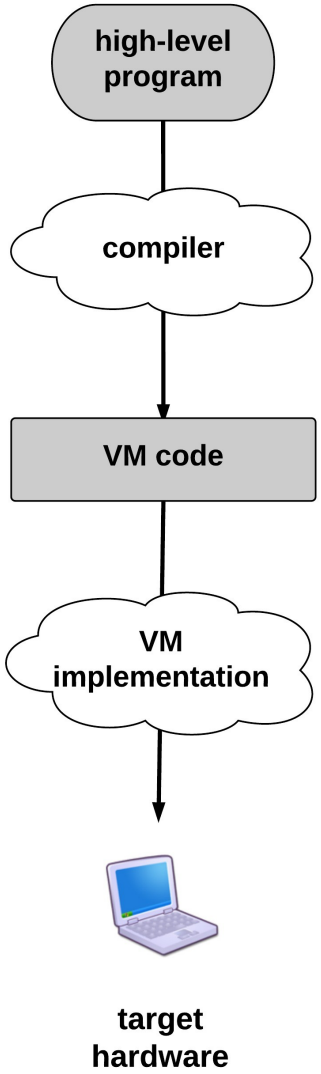
2. Translate VM-code into machine language (VM implementation)

Here, we use a typical VM architecture (= stack machine model) as also used in e.g. Java Virtual Machine (JVM)

To get ML-code from VM-code:

- write a program (*VM translator)* designed to translate VM code into Hack assembly code.

The stack machine model



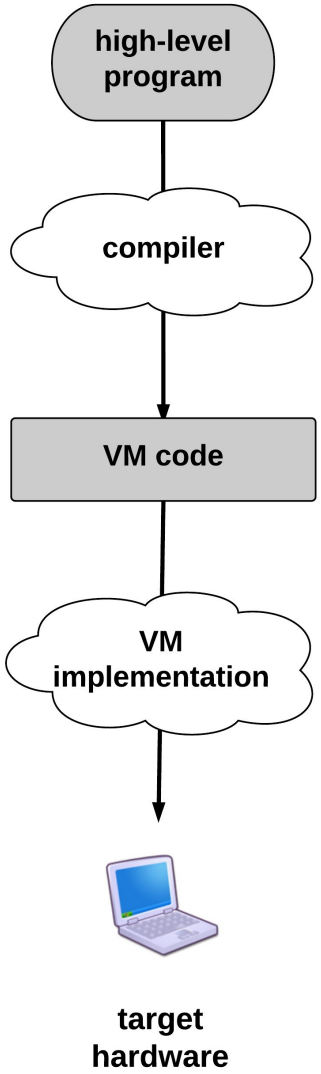
A virtual machine model typically has a language, in which one can write VM programs.

The VM language that we present here consists of four types of commands:

<u>Stack machine, manipulated by:</u>

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

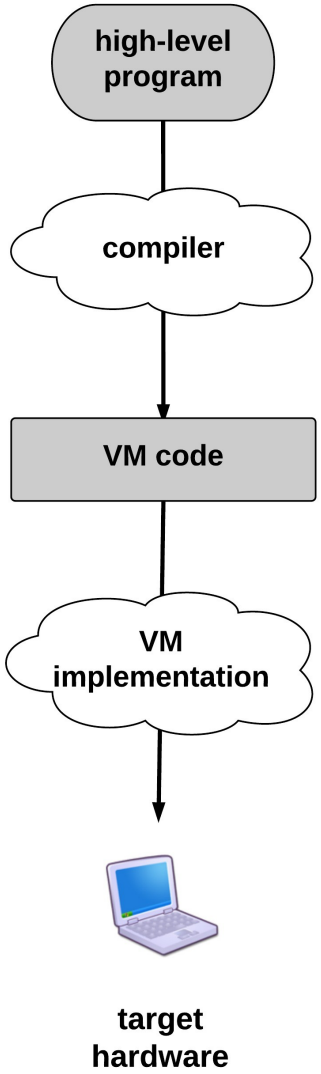
The stack machine model



<u>Stack machine, manipulated by:</u>

- Arithmetic / logical commands
  → Definition
  → Implementation
- Memory segment commands
  → Definition
  → Implementation
- Branching commands
  → Definition
  → Implementation
- Function commands
  → Definition
  → Implementation

The stack machine model



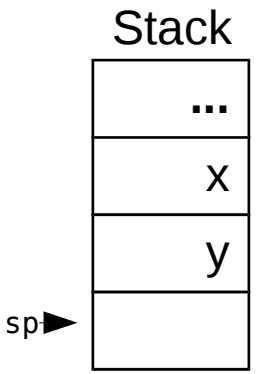
<u>Stack machine, manipulated by:</u>

- Arithmetic / logical commands
  → Definition
  → Implementation
- Memory segment commands
  → Definition
  → Implementation
- Branching commands
  → Definition
  → Implementation
- Function commands
  → Definition
  → Implementation

Arithmetic / Logical Commands

UNIVERSITY OF LEEDS

Command	Return value *(after popping the operand/s)*	Comment
add	$x + y$	Integer *(2's complement)*
sub	$x - y$	Integer *(2's complement)*
neg	$-y$	Integer *(2's complement)*
eq	true *if* $x = y$, *else* false	Boolean
gt	true *if* $x > y$, *else* false	Boolean
lt	true *if* $x < y$, *else* false	Boolean
and	x and y	Boolean *(bit-wise)*
or	x or y	Boolean *(bit-wise)*
not	not x	Boolean *(bit-wise)*



Return values are pushed to the stack

Arithmetic / Logical Commands

UNIVERSITY OF LEEDS

VM code

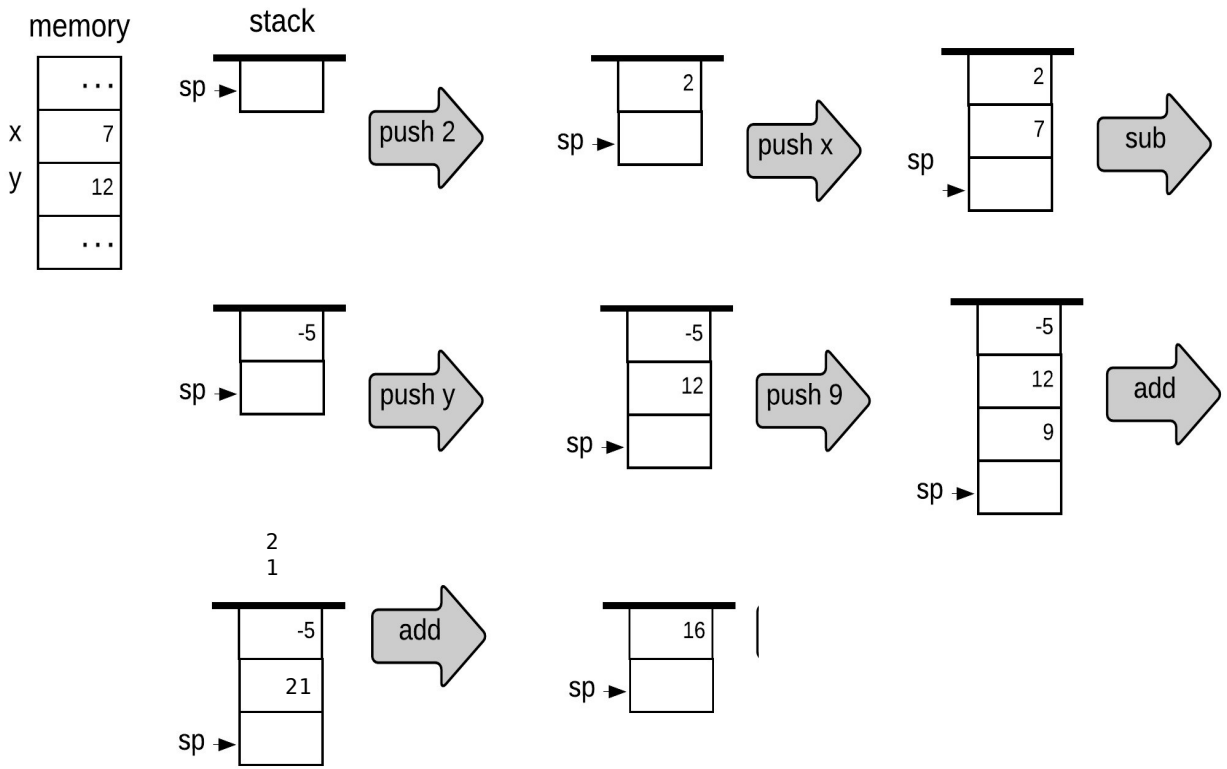
```
//(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
```

Arithmetic / Logical Commands

UNIVERSITY OF LEEDS

VM code

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

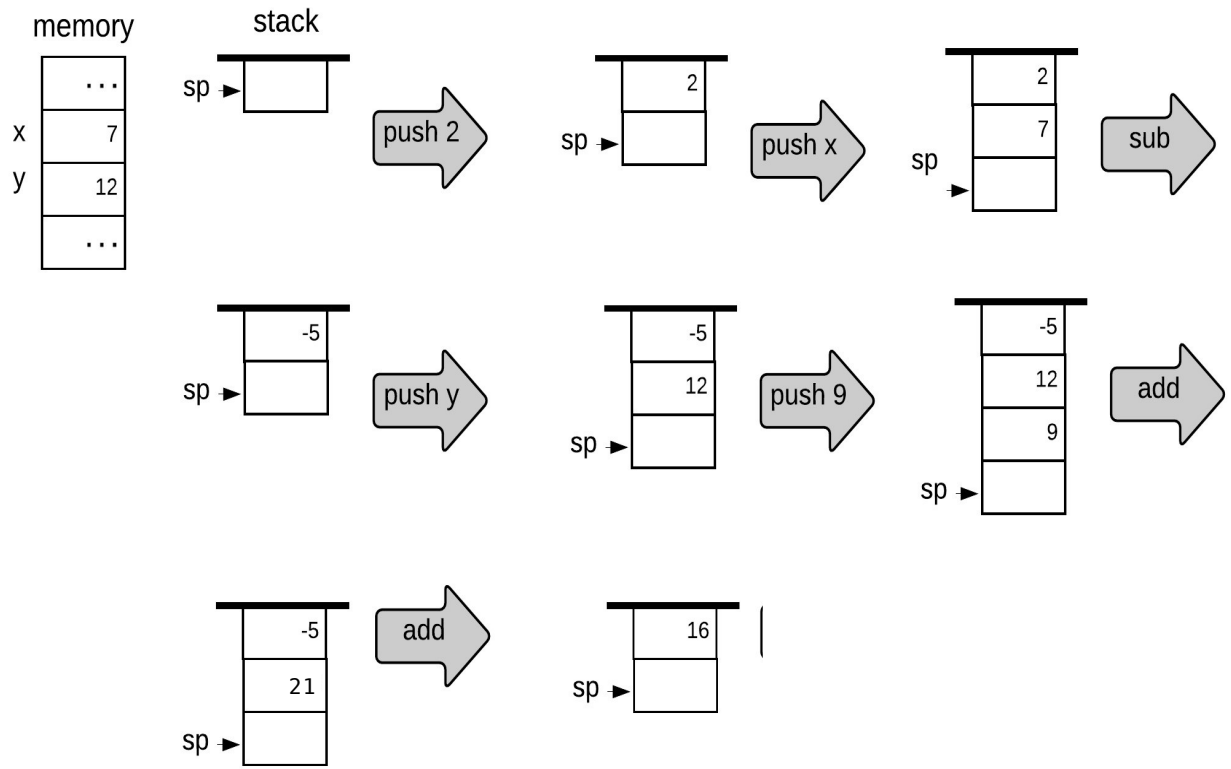


Arithmetic / Logical Commands

UNIVERSITY OF LEEDS

VM code

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



(2-x)+(y+9) is “translated to” 2 x – y 9 + +

Arithmetic / Logical Commands

UNIVERSITY OF LEEDS

VM code

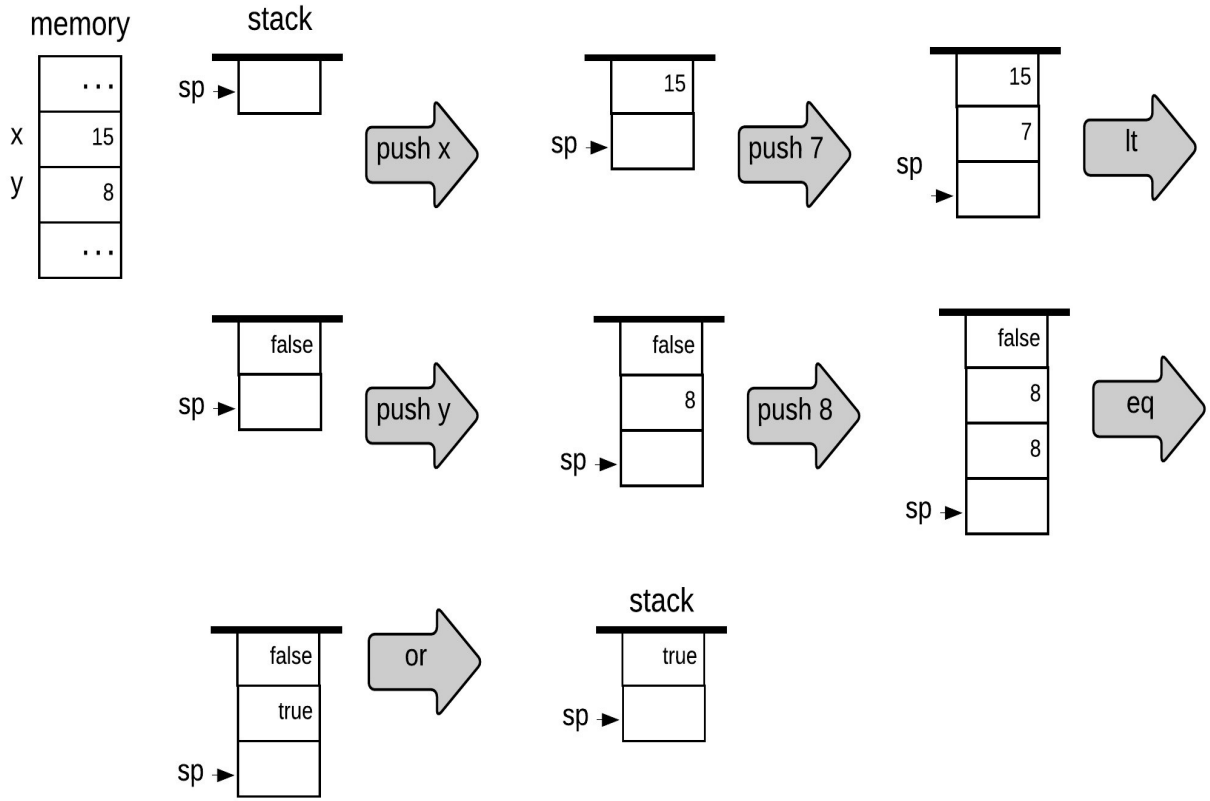
```
// (x<7) or (y==8)
push x
push 7
lt
push y
push 8
eq
or
```

Arithmetic / Logical Commands

UNIVERSITY OF LEEDS

VM code

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

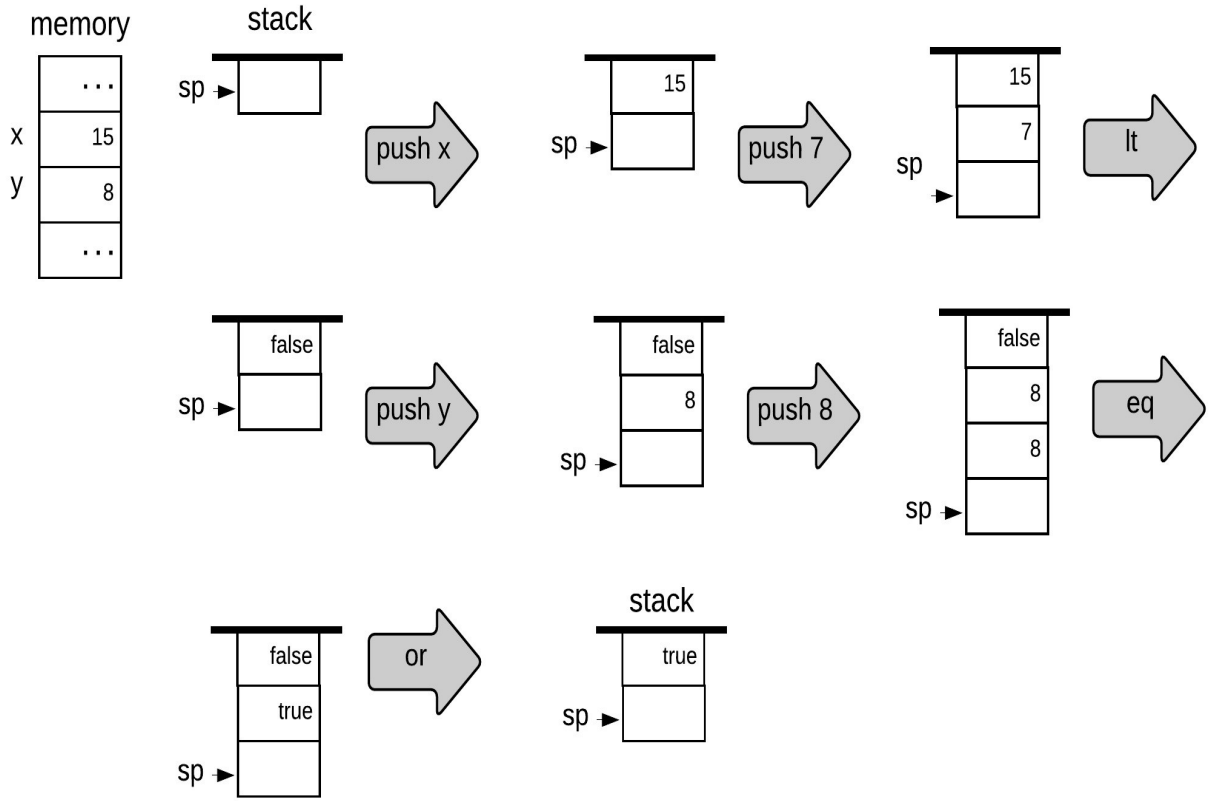


Arithmetic / Logical Commands

UNIVERSITY OF LEEDS

VM code

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

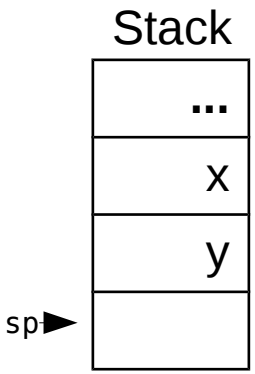


`(x<7)or(y==8)` is “translated to” `x 7 < y 8 eq or`

Arithmetic / Logical Commands

UNIVERSITY OF LEEDS

Command	Return value *(after popping the operand/s)*	Comment
add	$x + y$	Integer *(2's complement)*
sub	$x - y$	Integer *(2's complement)*
neg	$-y$	Integer *(2's complement)*
eq	$x == y$	boolean
gt	$x > y$	boolean
lt	$x < y$	Boolean
and	x `and` y	boolean
or	x `or` y	boolean
not	`not` x	boolean

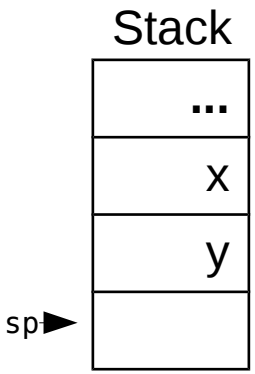


<u>Observation:</u> Any arithmetic or logical expression can be expressed and evaluated by applying some sequence of the above operations on a stack.

Arithmetic / Logical Commands

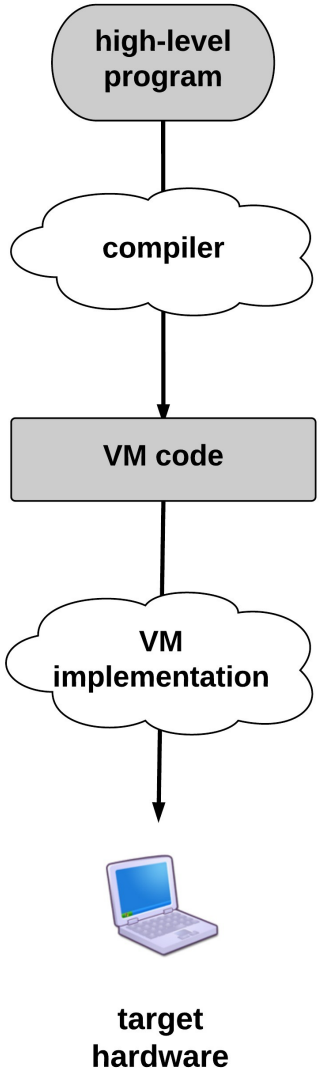
UNIVERSITY OF LEEDS

Command	Return value *(after popping the operand/s)*	Comment
add	$x + y$	Integer *(2's complement)*
sub	$x - y$	Integer *(2's complement)*
neg	$-y$	Integer *(2's complement)*
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



The VM represents true as “-1” and false as “0” !

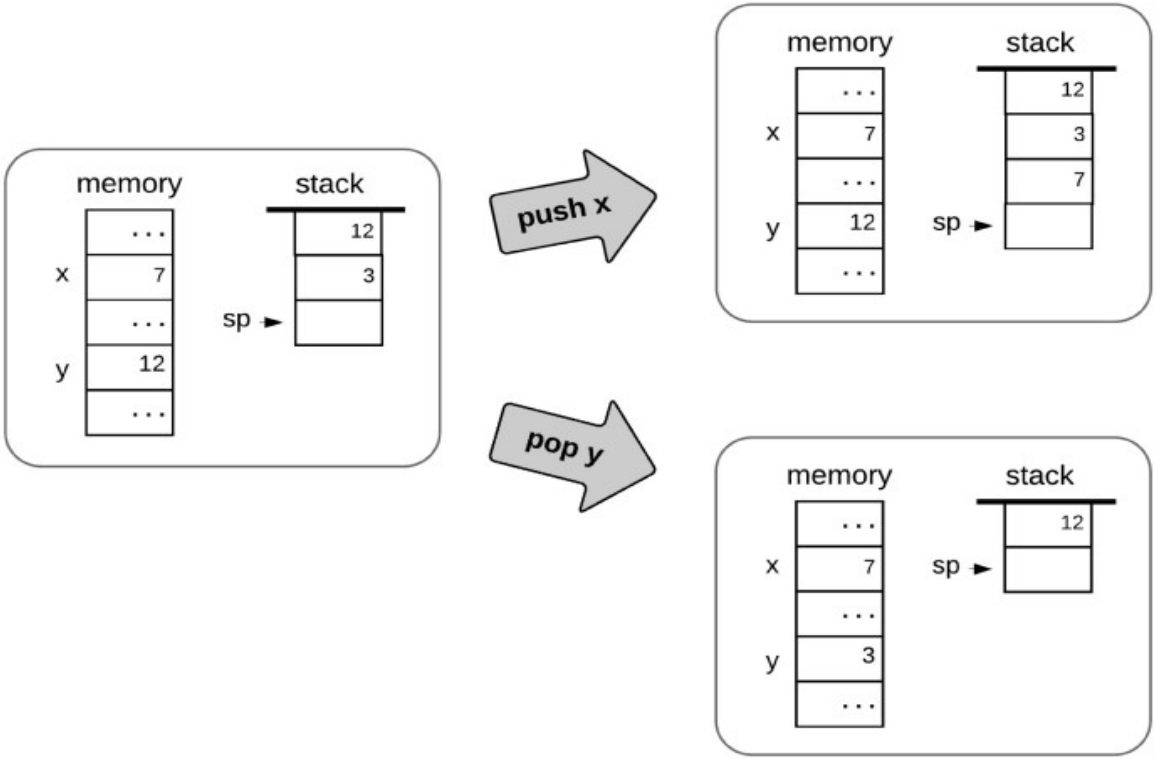
The stack machine model



Stack machine, manipulated by:

- Arithmetic / logical commands
  → Definition ✓
  → Implementation
- Memory segment commands
  → Definition
  → Implementation
- Branching commands
  → Definition
  → Implementation
- Function commands
  → Definition
  → Implementation

Memory Commands – An overview



Basic Idea:

- `push x`: add the element at memory location x to the stack's top
- `pop y`: remove the top element from the stack and store it at memory location y

Memory Commands – An overview

UNIVERSITY OF LEEDS

High-level code

```
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

```
...
...
...
...
...
...
...
...
...
...
```

Memory Commands – An overview

UNIVERSITY OF LEEDS

High-level code

```
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
...
...
```

Memory Commands – An overview

UNIVERSITY OF LEEDS

High-level code

```
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
...
...
```

Variable kinds

- Argument variables
- Local variables
- Static variables

(More kinds later)

Memory Commands – An overview

UNIVERSITY OF LEEDS

High-level code

```
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
...
...
```

Problem: Different kind of variables have different roles and must be treated differently

→ We must be able to keep track the particular "kind of variable" in our VM abstraction (otherwise things may go wrong in the execution of the program)

Solution: Use more than one memory segment

Variable kinds

- Argument variables
- Local variables
- Static variables

(More kinds later)

Memory Commands – An overview

UNIVERSITY OF LEEDS

High-level code

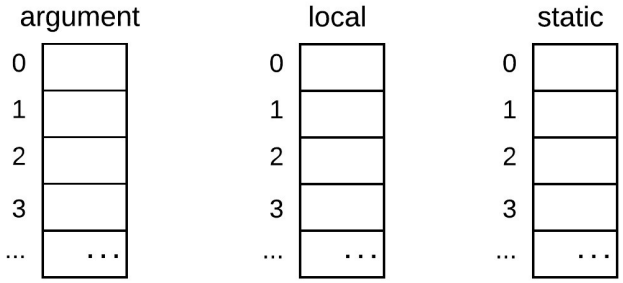
```
class Foo {
   static int s1, s2;
   function int bar (int x, int y) {
      var int a, b, c;
      ...
      let c = s1 + y;
      ...
   }
}
```

compile

Compiled VM code

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

Virtual memory segments:



Variable kinds

- Argument variables
- Local variables
- Static variables

(More kinds later)

Memory Commands – An overview

UNIVERSITY OF LEEDS

High-level code

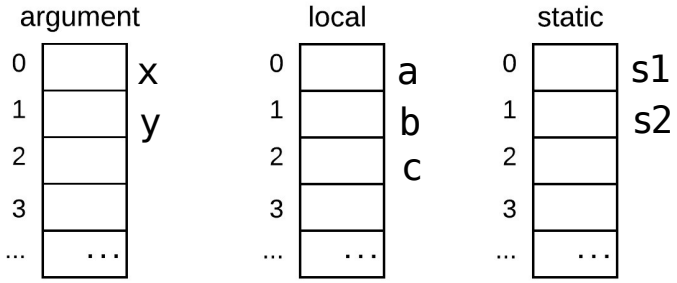
```
class Foo {
   static int s1, s2;
   function int bar (int x, int y) {
      var int a, b, c;
      ...
      let c = s1 + y;
      ...
   }
}
```

compile

Compiled VM code

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

Virtual memory segments:



Variable kinds

- Argument variables
- Local variables
- Static variables

(More kinds later)

Memory Commands – An overview

UNIVERSITY OF LEEDS

High-level code

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

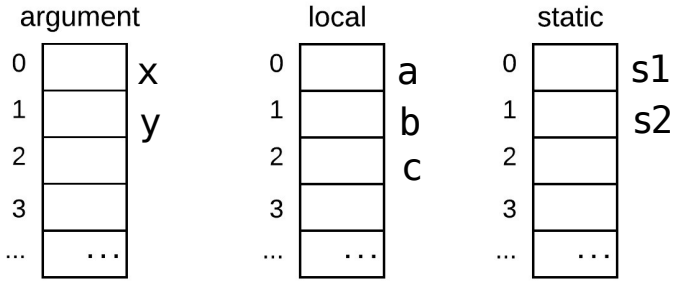
compile

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 Commands – An overview

UNIVERSITY OF LEEDS

High-level code

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

compile

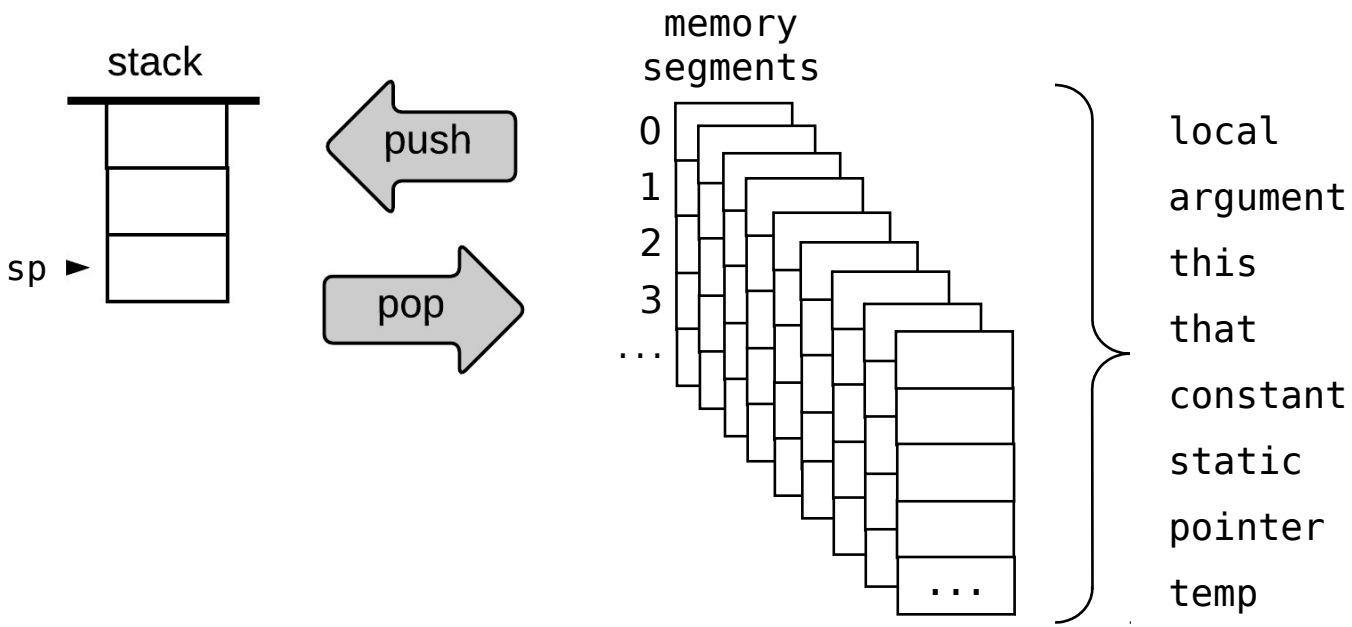
Compiled VM code

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

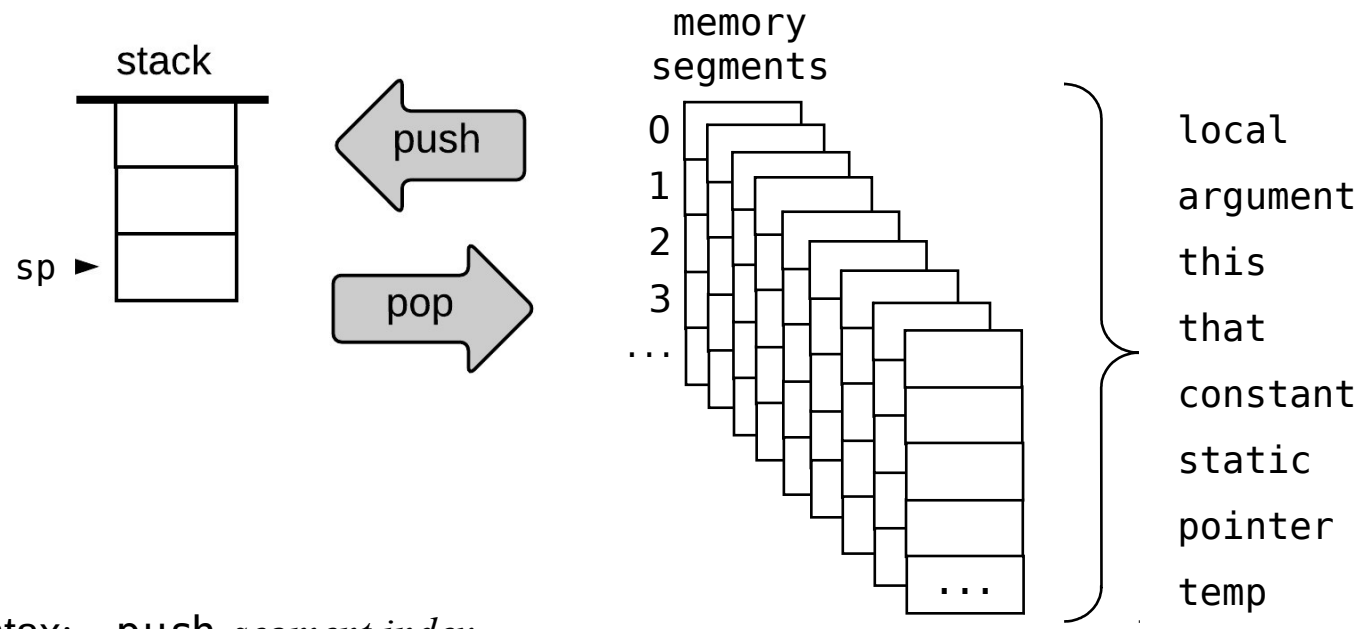
Following compilation, all the symbolic references are replaced with references to virtual memory segments

Variable names are lost and not recognized by our VM abstraction, but this is not important, since we preserved the "role semantics" of these variables!

Memory Commands and Segments



Memory Commands and Segments

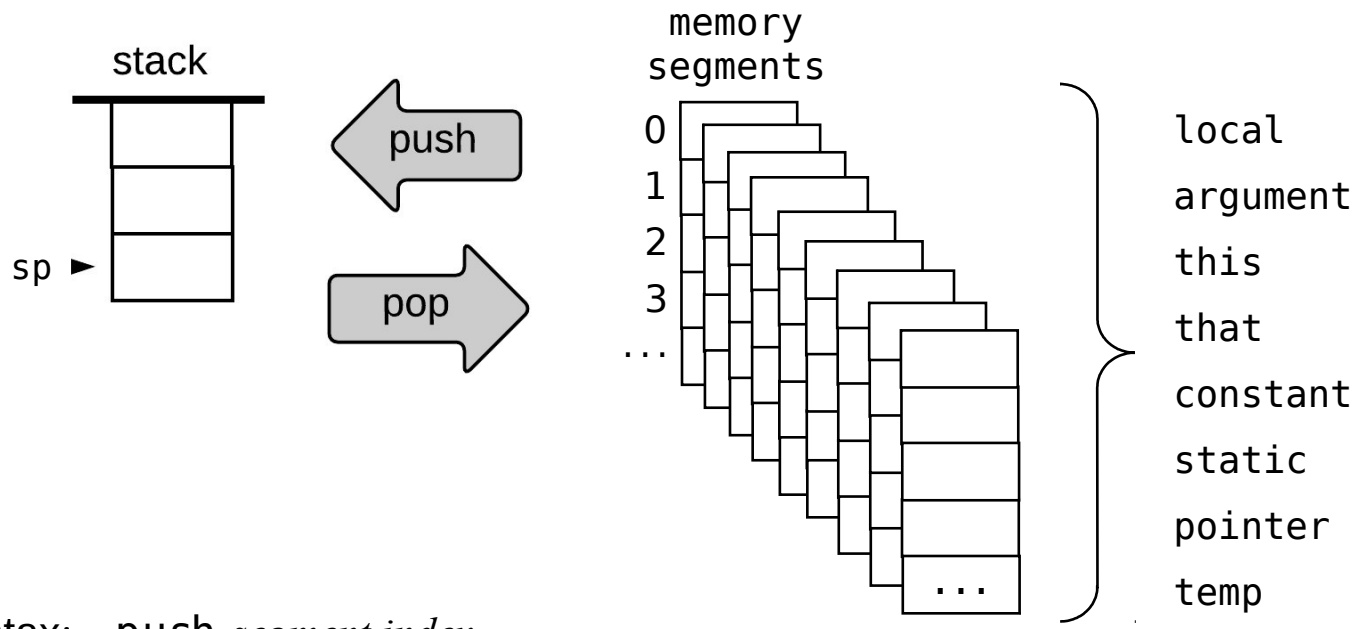


<u>Syntax:</u> `push` *segment index*

Push the value of *segment[index]* onto the stack, where *segment* is `argument, local, static, this, that, pointer, temp, constant` and *index* is a non-negative integer.

Memory Commands and Segments

UNIVERSITY OF LEEDS



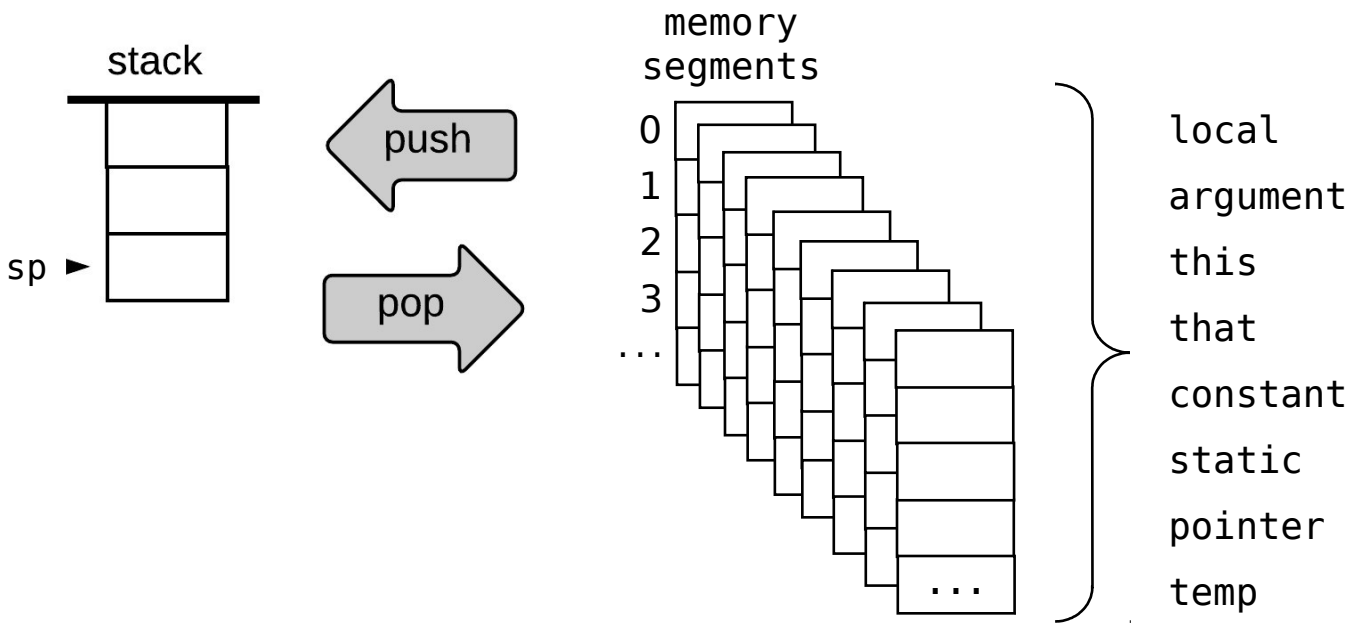
<u>Syntax:</u> `push` *segment index*

Push the value of *segment[index]* onto the stack, where *segment* is
`argument, local, static, this, that, pointer, temp, constant`
and *index* is a non-negative integer.

<u>Syntax:</u> `pop` *segment index*

Pop the top stack value and store it in *segment[index],* where *segment* is
`argument, local, static, this, that, pointer, temp`
and *index* is a non-negative integer.
(one cannot "pop" into a constant – for good reasons ;))

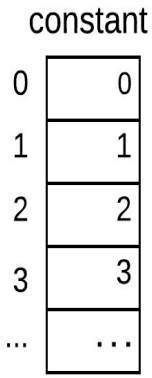
Memory Commands and Segments



REMARK*:*

`constant` is already initialized as:

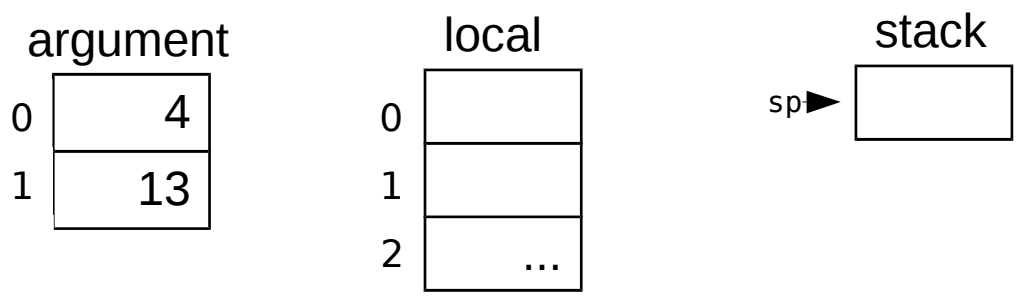
(= Pseudo-segment that holds all the constants in the range 0 . . 32767)



Memory Commands and Segments

UNIVERSITY OF LEEDS

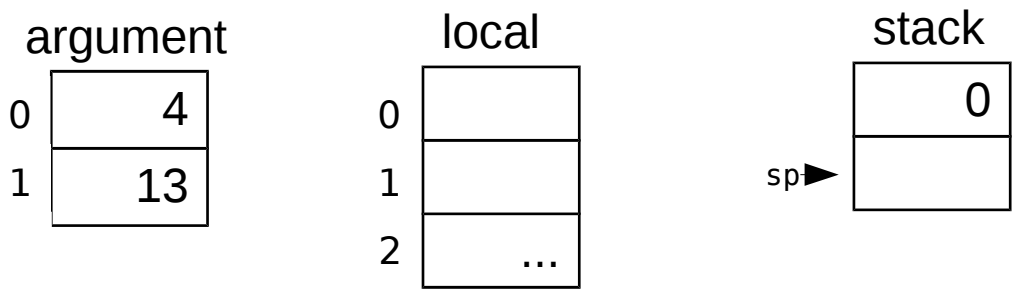
```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```



Memory Commands and Segments

UNIVERSITY OF LEEDS

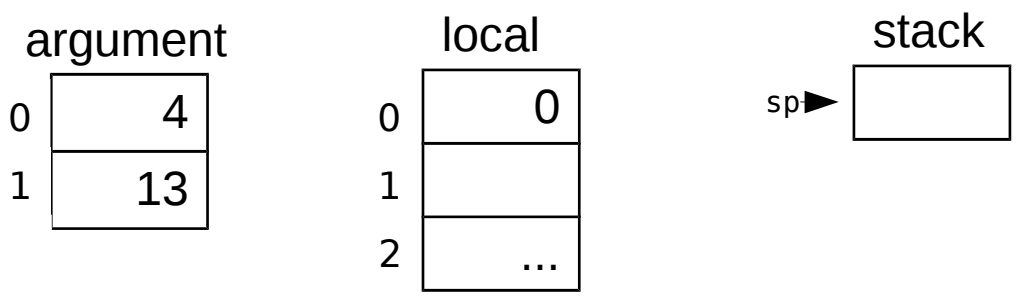
```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```



Memory Commands and Segments

UNIVERSITY OF LEEDS

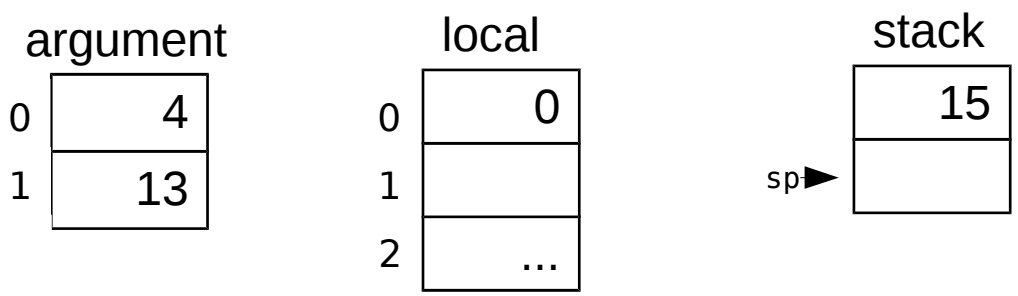
```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```



Memory Commands and Segments

UNIVERSITY OF LEEDS

```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```



Memory Commands and Segments

UNIVERSITY OF LEEDS

```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```

argument

0	4
1	13

local

0	0
1	15
2	...

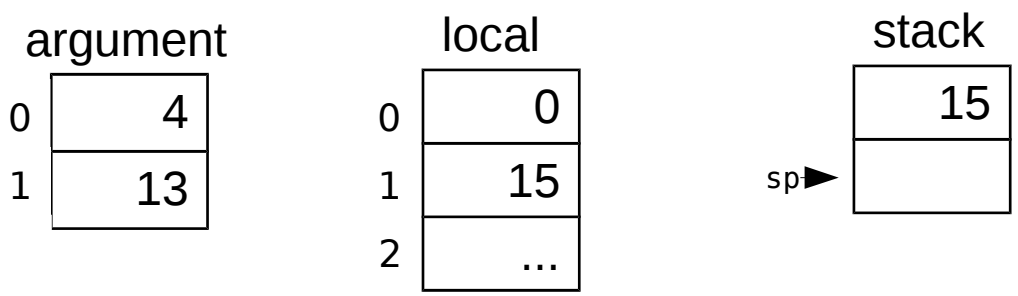
stack

sp

Memory Commands and Segments

UNIVERSITY OF LEEDS

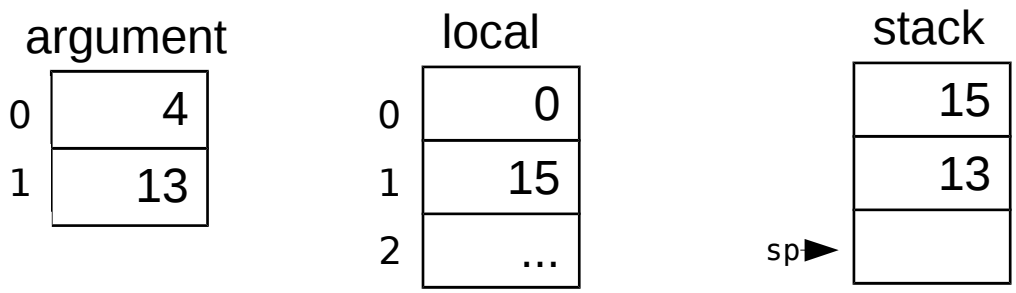
```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```



Memory Commands and Segments

UNIVERSITY OF LEEDS

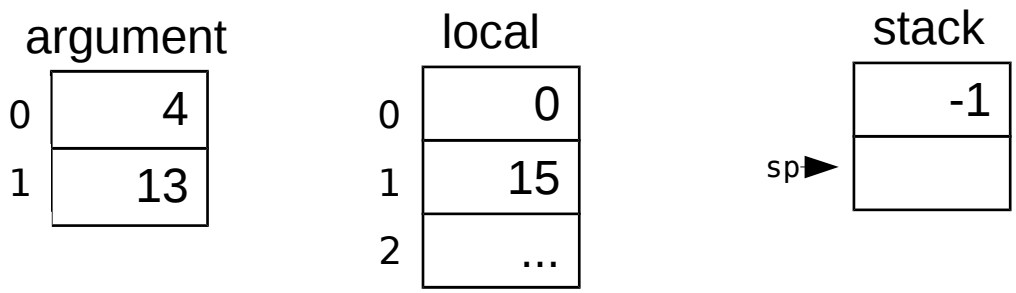
```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```



Memory Commands and Segments

UNIVERSITY OF LEEDS

```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```



Memory Commands and Segments

UNIVERSITY OF LEEDS

```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```

argument

0	4
1	13

local

0	0
1	15
2	-1

stack

sp ►

Memory Commands and Segments

UNIVERSITY OF LEEDS

```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```

argument

0	4
1	13

local

0	0
1	15
2	-1

stack

	0
sp ►	

Memory Commands and Segments

UNIVERSITY OF LEEDS

```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```

argument

0	4
1	13

local

0	0
1	15
2	-1

stack

	0
	4
sp►	

Memory Commands and Segments

UNIVERSITY OF LEEDS

```
push constant 0
pop local 0
push constant 15
pop local 1
push local 1
push argument 1
gt
pop local 2
push local 0
push argument 0
add
...
```

argument

0	4
1	13

local

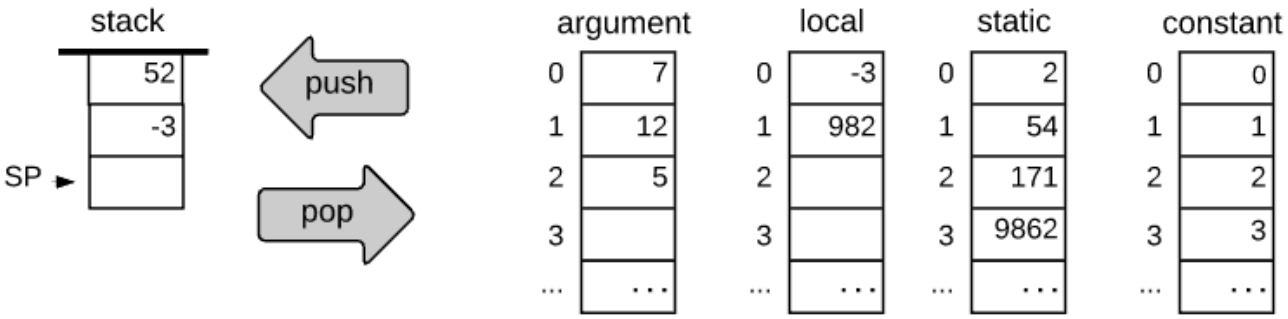
0	0
1	15
2	-1

stack

	4
sp ►	

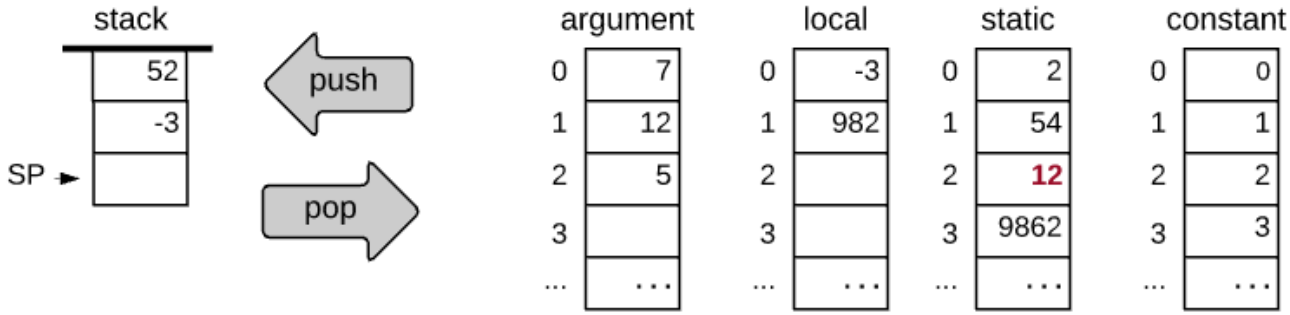
Memory Commands and Segments

UNIVERSITY OF LEEDS

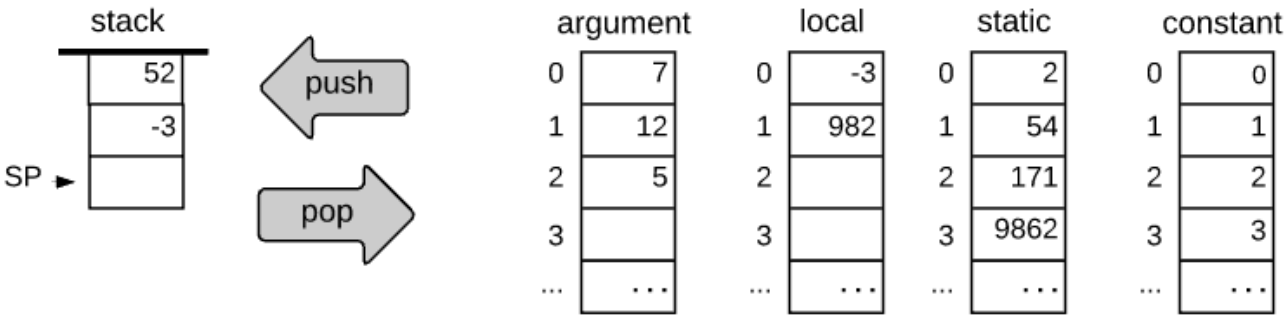



```
let  static 2 = argument 1
```

?

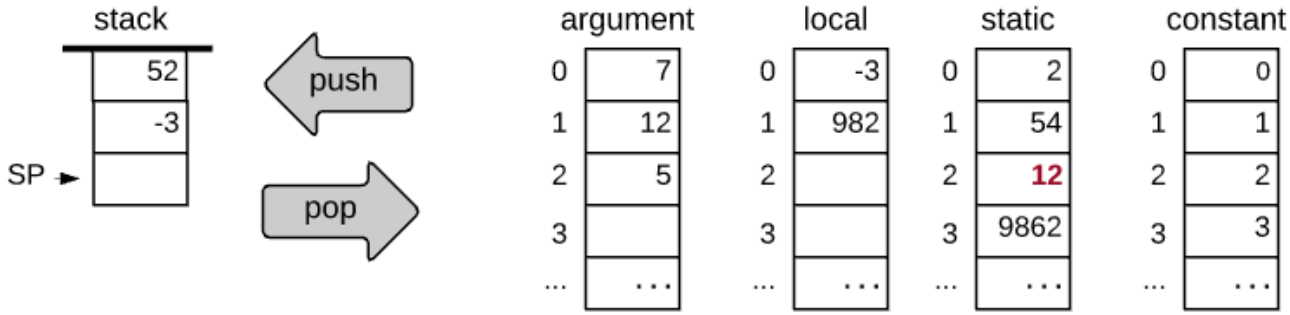


Memory Commands and Segments

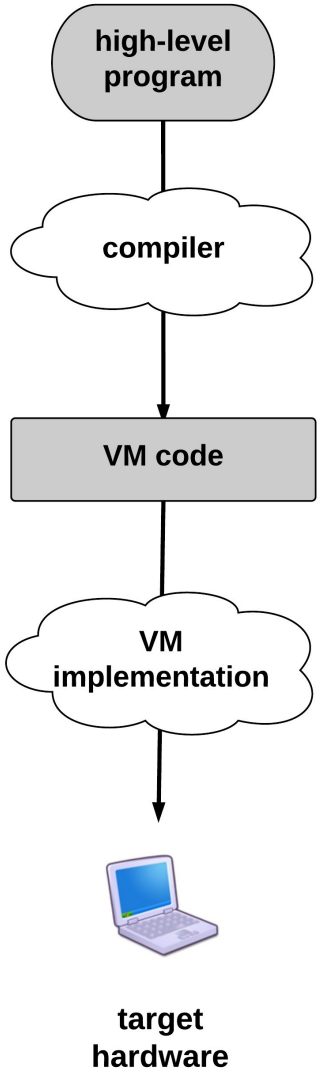



```
let  static 2 = argument 1
```

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



The stack machine model



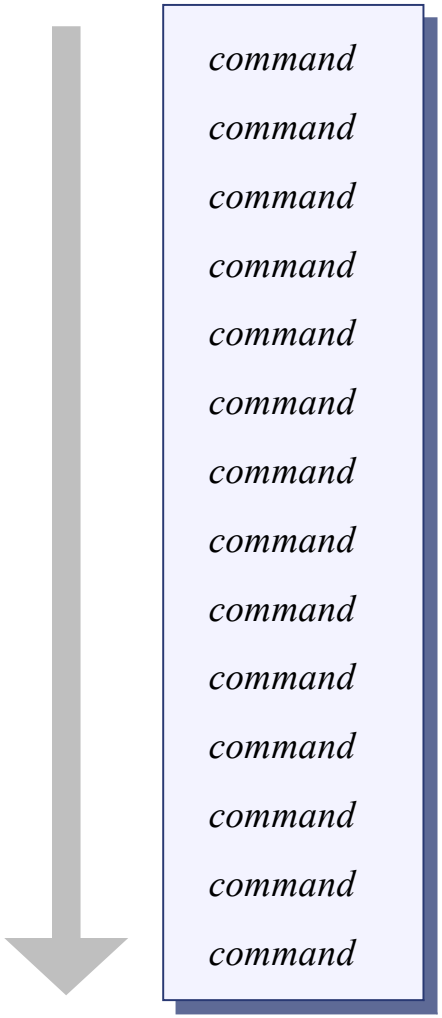
<u>Stack machine, manipulated by:</u>

- Arithmetic / logical commands
  - → Definition ✓
  - → Implementation
- Memory segment commands
  - → Definition ✓
  - → Implementation
- Branching commands
  - → Definition
  - → Implementation
- Function commands
  - → Definition
  - → Implementation

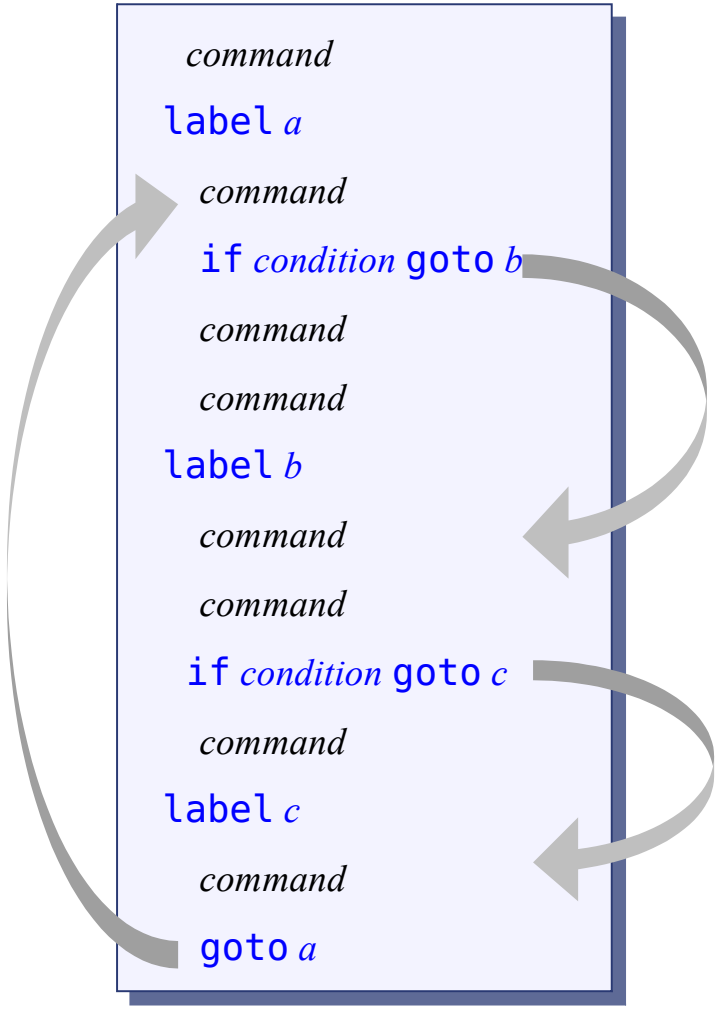
Branching commands

UNIVERSITY OF LEEDS

No branching



Several branching



Branching:

- Unconditional
- Conditional

Branching commands

UNIVERSITY OF LEEDS

High-level program

```
// Returns x * y
int mult(int x, int y) {
   int sum = 0;
   int n = 1;
   // sum = sum + x, y
    times
   while !(n > y) {
      sum += x;
      n++;
   }
   return sum;
}
```

compiler

Pseudo VM code

```
function mult(x,y)
   push 0
   pop sum
   push 1
   pop n
label WHILE_LOOP
   push n
   push y
   gt
   if-goto ENDLOOP
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto WHILE_LOOP
label ENDLOOP
   push sum
   return
```

Branching commands

UNIVERSITY OF LEEDS

High-level program

```
// Returns x * y
int mult(int x, int y) {
   int sum = 0;
   int n = 1;
   // sum = sum + x, y
    times
   while !(n > y) {
      sum += x;
      n++;
   }
   return sum;
}
```

compiler

Pseudo VM code

```
function mult(x,y)
   push 0
   pop sum
   push 1
   pop n
label WHILE_LOOP
   push n
   push y
   gt
   if-goto ENDLOOP
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto WHILE_LOOP
label ENDLOOP
   push sum
   return
```

Unconditional branching:

`goto` *label*

Jumps to execute the command just after *label*.

Branching commands

UNIVERSITY OF LEEDS

High-level program

```
// Returns x * y
int mult(int x, int y) {
   int sum = 0;
   int n = 1;
   // sum = sum + x, y
    times
   while !(n > y) {
      sum += x;
      n++;
   }
   return sum;
}
```

compiler

Pseudo VM code

```
function mult(x,y)
   push 0
   pop sum
   push 1
   pop n
label WHILE_LOOP
   push n
   push y
   gt
   if-goto ENDLOOP
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto WHILE_LOOP
label ENDLOOP
   push sum
   return
```

Unconditional branching:

`goto` *label*

Jumps to execute the command just after *label*.

Conditional branching:

`if-goto` *label*

VM logic:

1. *cond* = `pop`;
2. if *cond* jump to execute the command just after *label*.

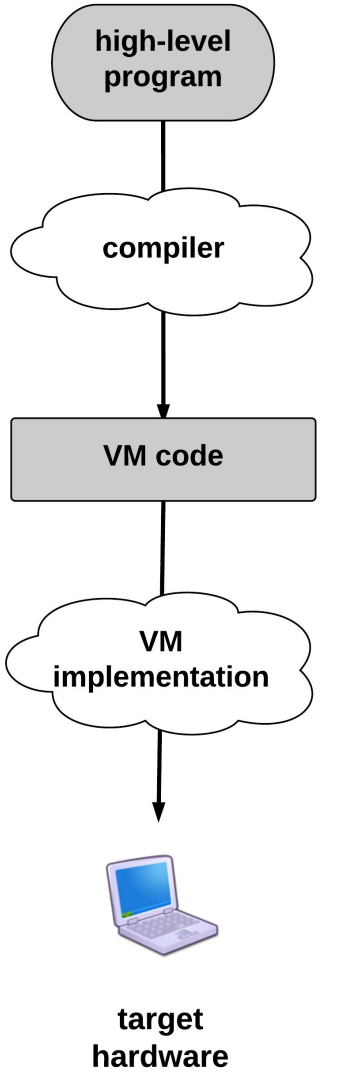
(Requires pushing the condition to the stack before the `if-goto` command)

Branching commands

UNIVERSITY OF LEEDS

- `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

The stack machine model



Stack machine, manipulated by:

- Arithmetic / logical commands
  → Definition ✓
  → Implementation
- Memory segment commands
  → Definition ✓
  → Implementation
- Branching commands
  → Definition ✓
  → Implementation
- Function commands
  → Definition
  → Implementation

Functions (abstraction)

UNIVERSITY OF LEEDS

High-level programming languages can be extended using:

- Subroutines
- Functions
- Procedures
- Methods
- Etc.

functions

(different names of the same thing)

Functions = everything that returns some value, usually based on input arguments

Functions

UNIVERSITY OF LEEDS

High-level program

```
...
sqrt(x – 17 + x * 5)
...
```

compiler

Pseudo VM code

```
...
push x
push 17
sub
push x
push 5
call Math.multiply
add
call Math.sqrt
...
```

Functions

UNIVERSITY OF LEEDS

High-level program

```
...
sqrt(x – 17 + x * 5)
...
```

compiler

Pseudo VM code

```
...
push x
push 17
sub
push x
push 5
call Math.multiply
add
call Math.sqrt
...
```

The VM language features:

- primitive operations (fixed): add, sub, ...
- abstract operations (extensible): multiply, sqrt, ...

Programming style:

- Applying a primitive operator or calling a function have the same look-and-feel.

Functions

UNIVERSITY OF LEEDS

High-level program

```
// Returns x * y
int mult(int x, int y) {
   int sum = 0;
   int n = 1;
   // sum = sum + x, y times
   while !(n > y) {
      sum += x;
      n++;
   }
   return sum;
}
```

compiler

Pseudo VM code

```
function mult(x,y)
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto END
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto LOOP
label END
   push sum
   return
```

Final VM code

```
function mult 2      // 2   local
    vars.
   push constant 0   // sum=0
   pop local 0
   push constant 1   // n=1
   pop local 1
label LOOP
   push local 1      // if !(n>y)
   push argument 1   //  goto END
   gt
   if-goto END
   push local 0      // sum+=x
   push argument 0
   add
   pop local 0
   push local 1      // n++
   push constant 1
   add
   pop local 1
   goto LOOP
label END
   push local 0     // return sum
   return
```

Functions

UNIVERSITY OF LEEDS

High-level program

```
// Returns x * y
int mult(int x, int y) {
   int sum = 0;
   int n = 1;
   // sum = sum + x, y times
   while !(n > y) {
      sum += x;
      n++;
   }
   return sum;
}
```

compiler

SYNTAX: `function` *name integer*

Pseudo VM code

```
function mult(x,y)
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto END
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto LOOP
label END
   push sum
   return
```

Final VM code

```
function mult 2      // 2   local
    vars.
   push constant 0   // sum=0
   pop local 0
   push constant 1   // n=1
   pop local 1
label LOOP
   push local 1      // if !(n>y)
   push argument 1   //  goto END
   gt
   if-goto END
   push local 0      // sum+=x
   push argument 0
   add
   pop local 0
   push local 1      // n++
   push constant 1
   add
   pop local 1
   goto LOOP
label END
   push local 0     // return sum
   return
```

Functions: an example

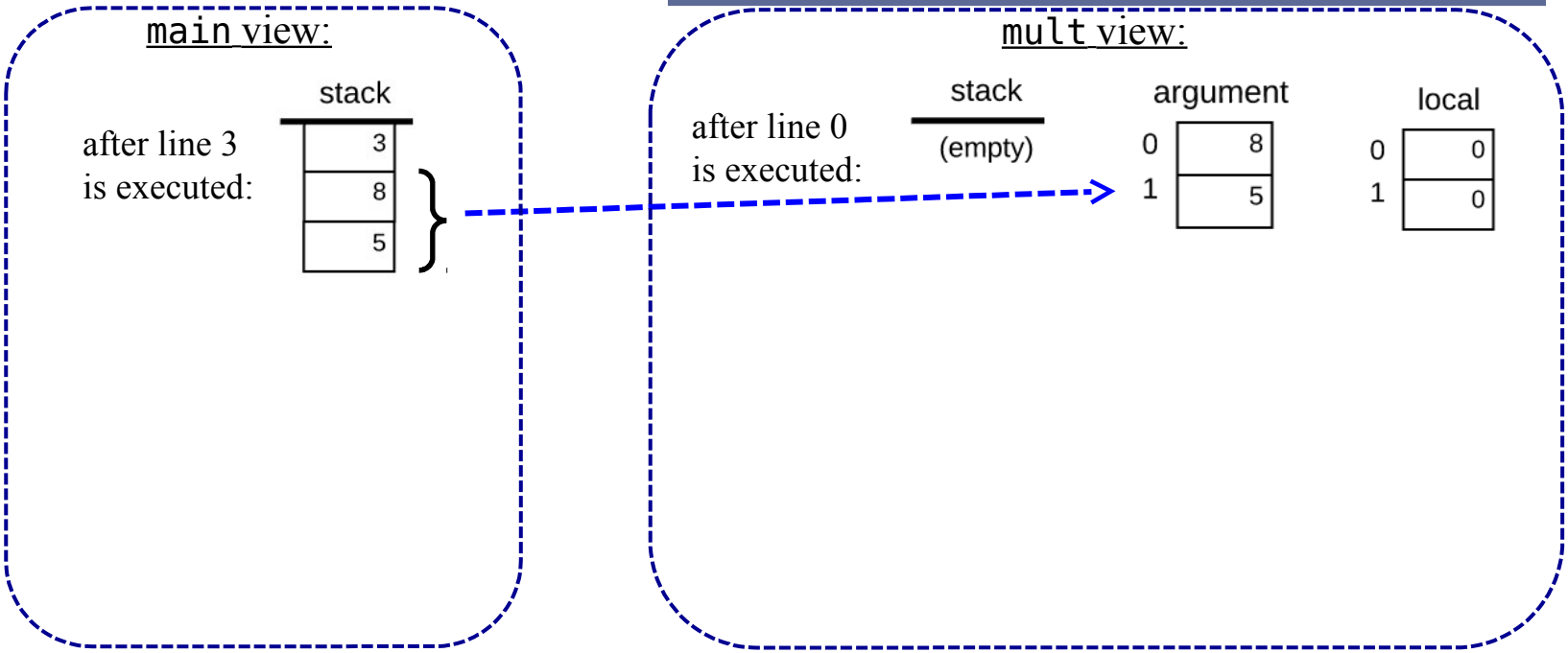
UNIVERSITY OF LEEDS

```
// 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



Functions: an example

UNIVERSITY OF LEEDS

```
// 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

`main` view:

after line 3 is executed:

stack

3
8
5

`mult` view:

after line 0 is executed:

stack

(empty)

local

0	0
1	0

after line 7 is executed:

stack

1
5

argument

0	8
1	5

local

0	0
1	1

Functions: an example

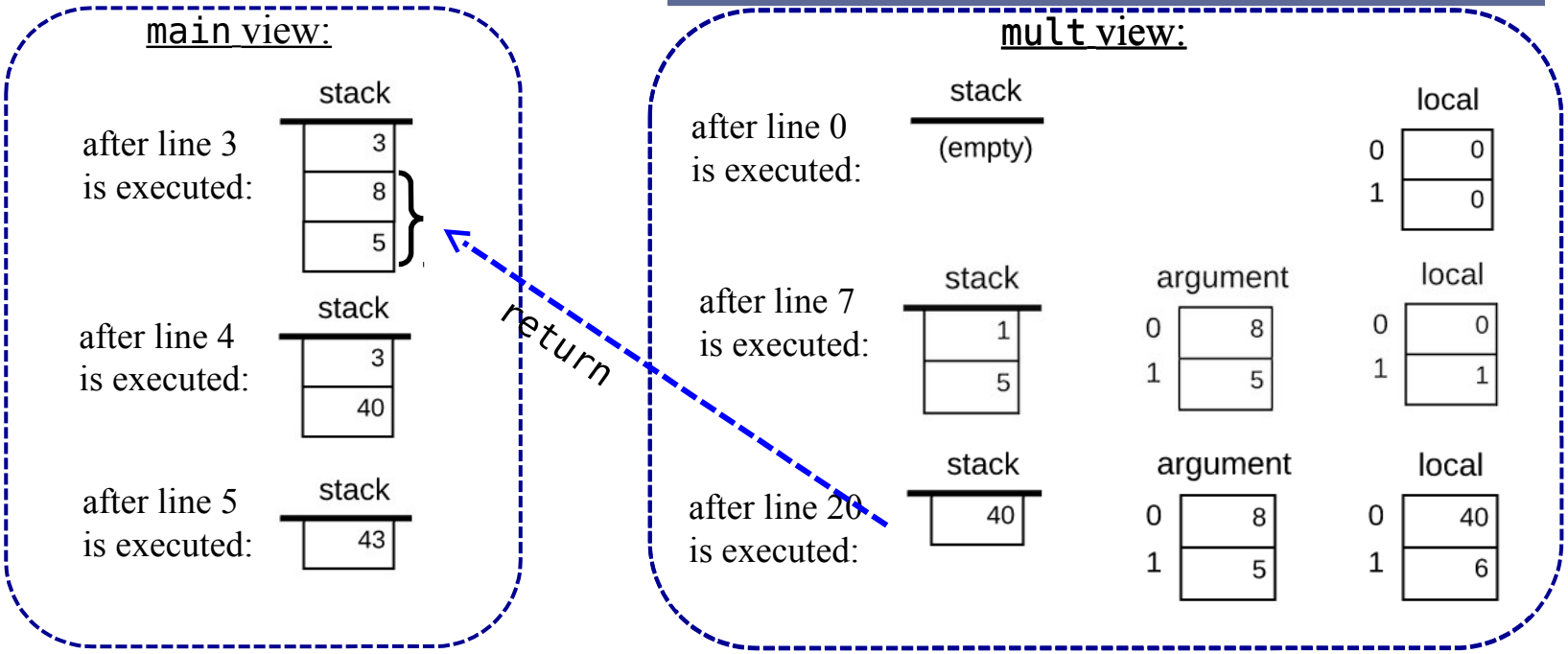
UNIVERSITY OF LEEDS

```
// 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



Functions: an example

UNIVERSITY OF LEEDS

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

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Implementation

We can write low-level code that manages the parameter passing, the saving and re-instating of function states, etc.

This task can be realized by writing code that...

- Handles the VM command **`call`**
- Handles the VM command **`function`**
- Handles the VM command **`return`**.

Functions, Call, Return

UNIVERSITY OF LEEDS

The VM language features three function-related commands:

- **`function`** f n
  Here starts the code of a function named f that has n local variables;

- **`call`** f m
  Call function f, stating that m arguments have already been pushed onto the stack by the caller;

- **`return`**
  Return to the calling function.

The Function Calling Protocol

UNIVERSITY OF LEEDS

<u>The calling function view:</u>

- Before calling the function, the caller must push as many arguments as necessary onto the stack;
- Next, the caller invokes the function using the `call` command;
- After the called function returns, the arguments that the caller has pushed before the call have disappeared from the stack, and a *return value* (that always exists) appears at the top of the stack;
- After the called function returns, the caller's memory segments `argument, local , static , this , that,` and `pointer` are the same as before the call, and the `temp` segment is undefined.

The Function Calling Protocol

UNIVERSITY OF LEEDS

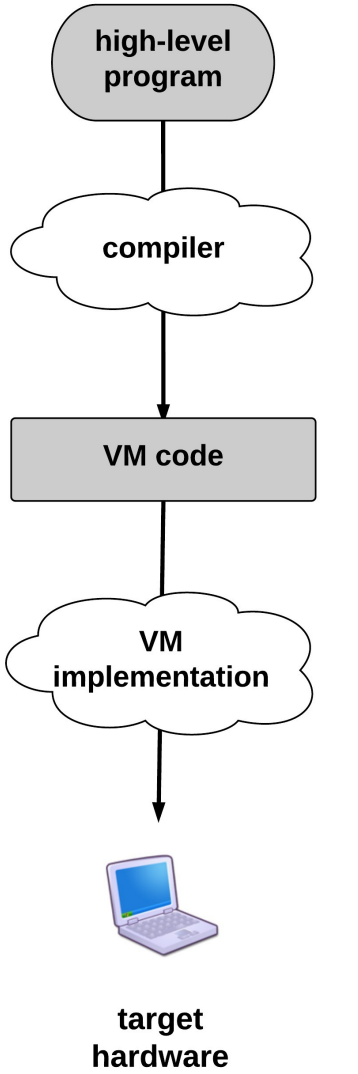
The calling function view:

- Before calling the function, the caller must push as many arguments as necessary onto the stack;
- Next, the caller invokes the function using the `call` command;
- After the called function returns, the arguments that the caller has pushed before the call have disappeared from the stack, and a *return value* (that always exists) appears at the top of the stack;
- After the called function returns, the caller's memory segments `argument, local , static , this , that,` and `pointer` are the same as before the call, and the `temp` segment is undefined.

The called function view:

- When the called function starts executing, its `argument` segment has been initialized with actual argument values passed by the caller and its `local` variables segment has been allocated and initialized to zeros.
- Before returning, the called function must push a value onto the stack.

The stack machine model

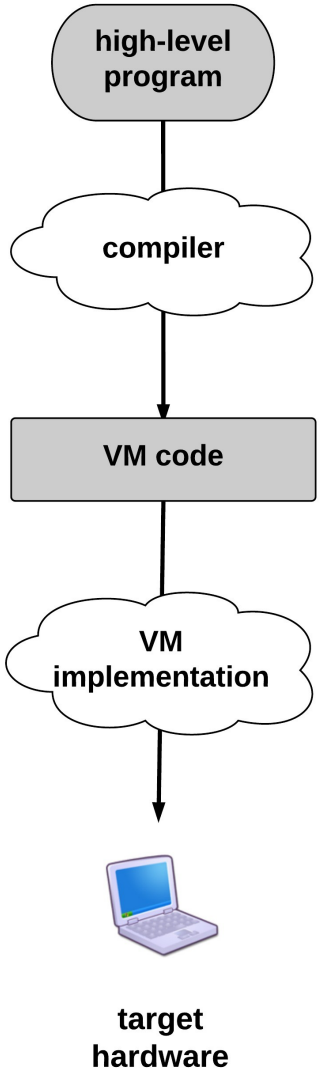


Stack machine, manipulated by:

- Arithmetic / logical commands
  → Definition ✓
  → Implementation
- Memory segment commands
  → Definition ✓
  → Implementation
- Branching commands
  → Definition ✓
  → Implementation
- Function commands
  → Definition ✓
  → Implementation

The stack machine model

UNIVERSITY OF LEEDS



Stack machine, manipulated by:

- Arithmetic / logical commands
  → Definition ✓
  → Implementation
- Memory segment commands
  → Definition ✓
  → Implementation
- Branching commands
  → Definition ✓
  → Implementation
- Function commands
  → Definition ✓
  → Implementation

Now we can write VM-programs!

A *VM program* is a collection of one or more files with a `.vm` extension, each consisting of one or more functions.

Within a `.vm` file, each VM command appears in a separate line, and in one of the following formats:

- *command*
  (e.g., `add`),
- *command arg*
  (e.g., `goto loop`)
- *command arg1 arg2*
  (e.g., push local 3).

The arguments are separated from each other and from the command part by an arbitrary number of spaces.

“ // ” comments can appear at the end of any line and are ignored. Blank lines are permitted and ignored.