University of Nottingham
UK | CHINA | MALAYSIA

Virtual Machine Part 1

Dr. Wooi Ping Cheah

Outlines

- Introduction to virtual machine
- VM abstraction
- VM implementation
- VM translator

Hello World

Jack program

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

abstraction

Issues:

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

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

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

Hello World

Jack program

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

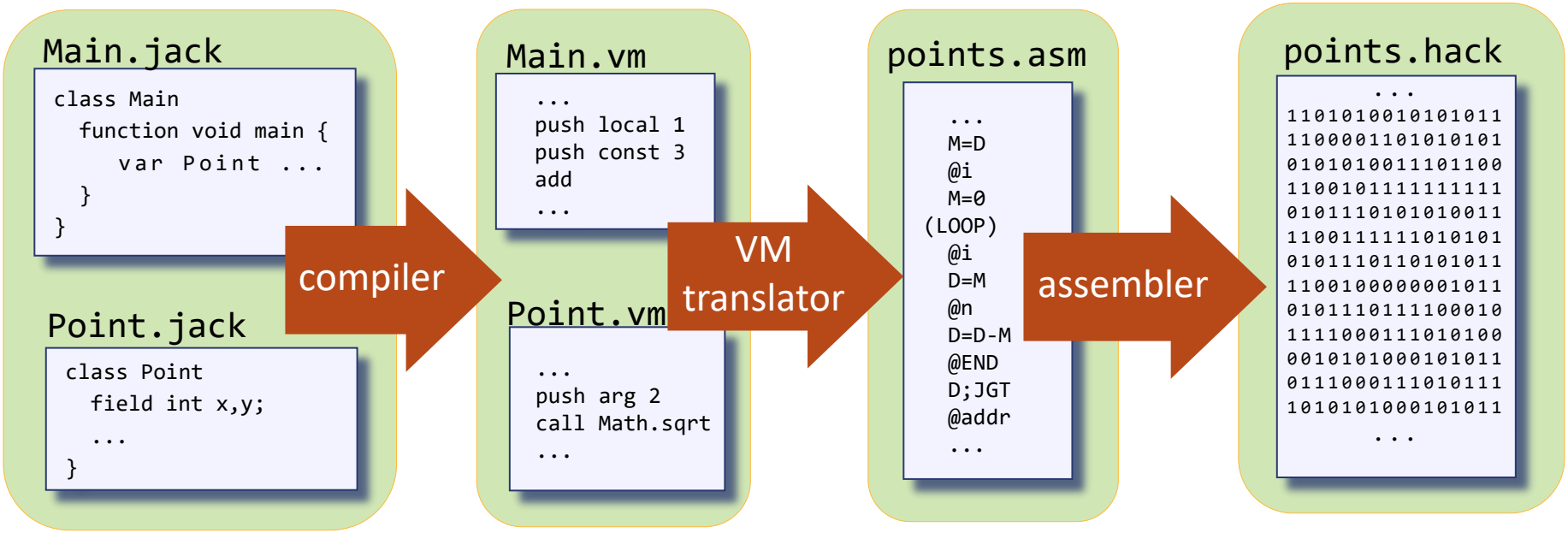
abstraction

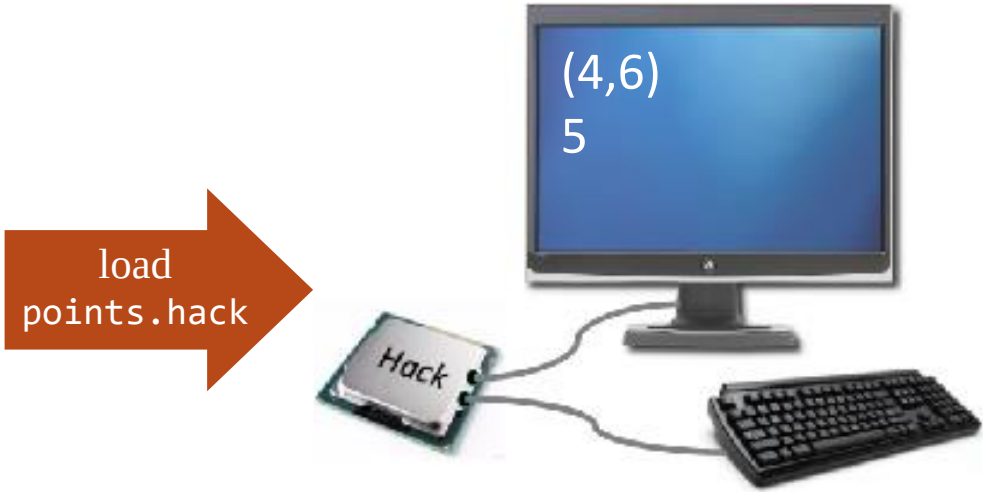
Issues:

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

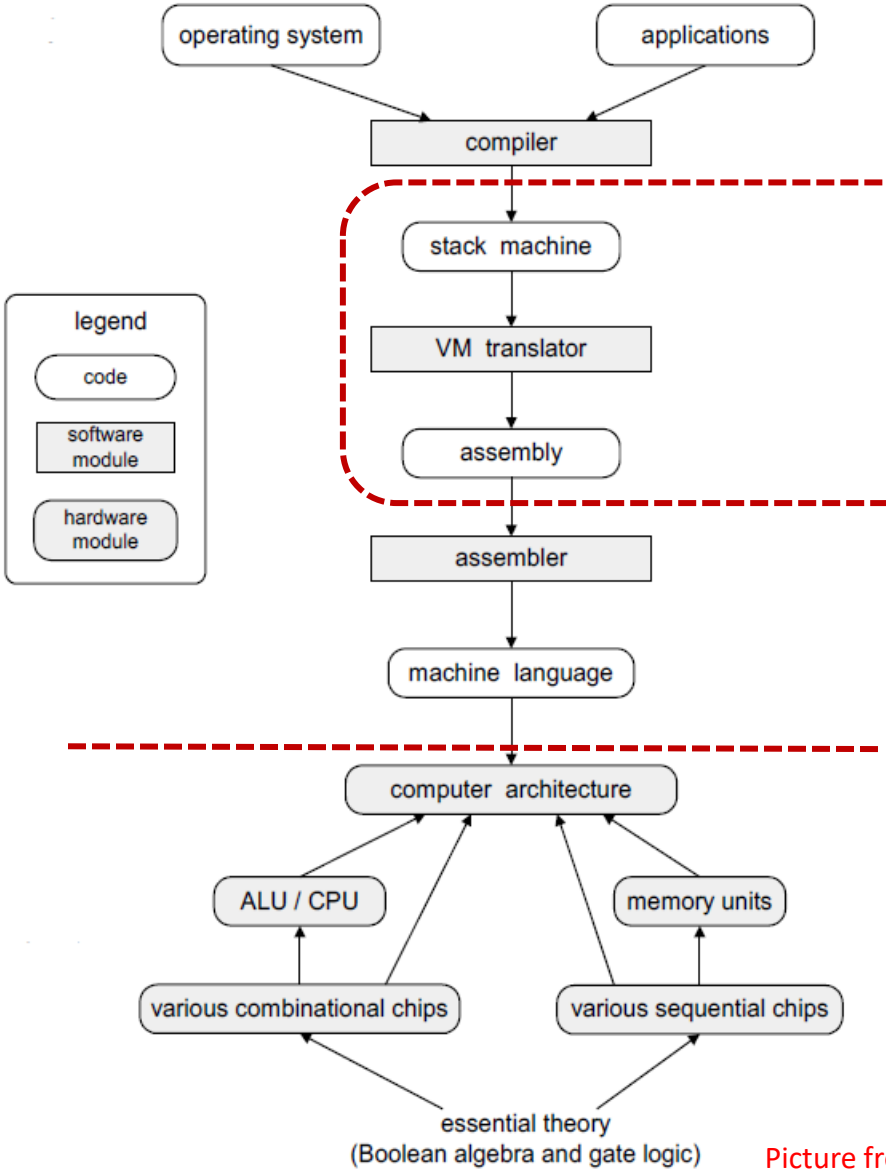
- Q: What makes the abstraction work?
- A:
  - Operating system,
  - Compiler,
  - Virtual machine,
  - Assembler.

From high-level to low-level





Overview of computer system



High-level programming languages

virtual machine

↓

VM translator

↓

assembly language

Computer hardware

Picture from Lecture 1 by Chris.

What is virtual machine?

- "*The VM is an abstract computer that does not exist for real, but can rather be realized on other computer platforms*." Nisan & Schocken.
- Keywords: abstract computer.
  - Not a real computer, a virtual computer.
  - A universal computer, can run on many kinds of real computers.

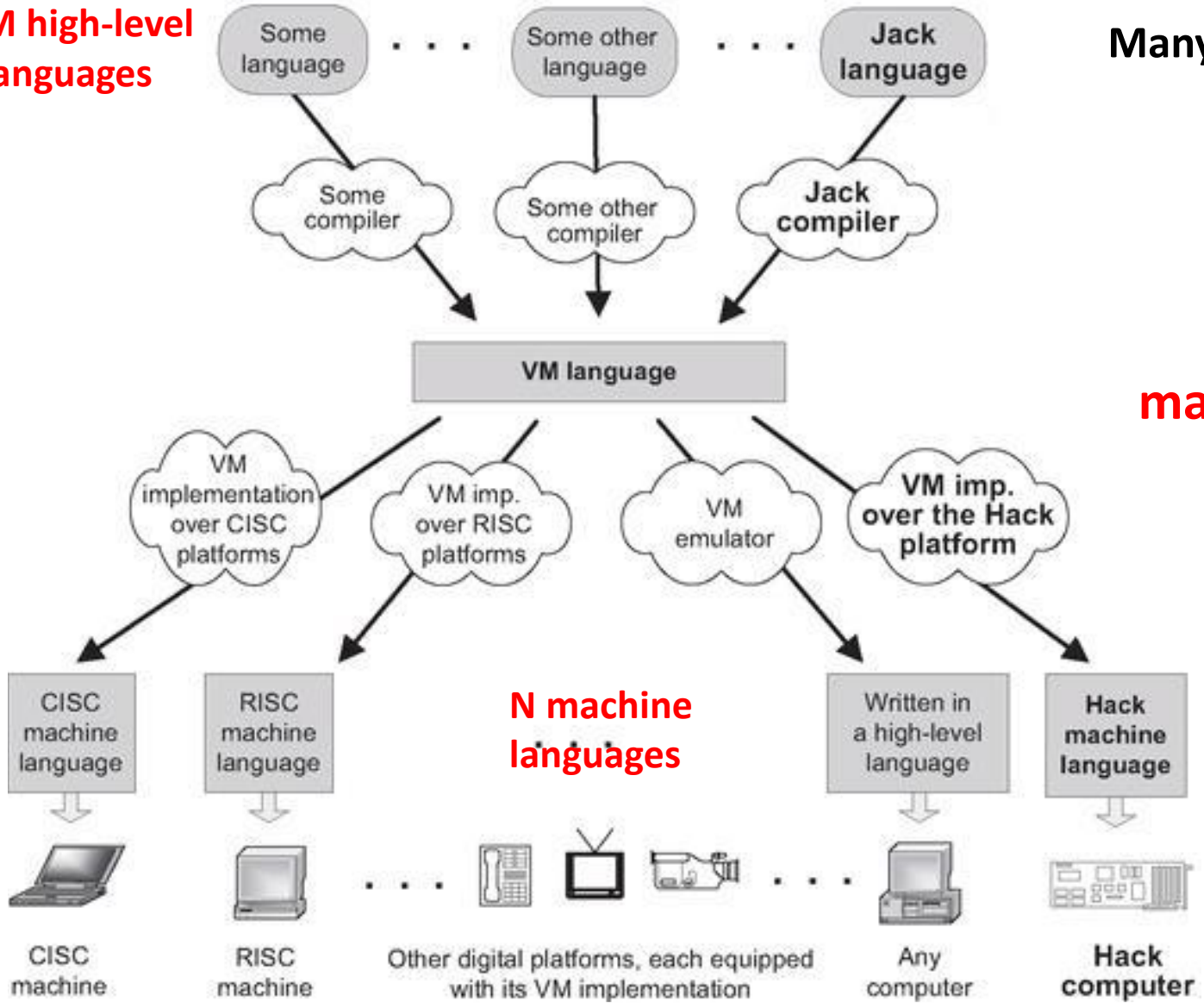
Examples of virtual machine

- Java:
  - Java virtual machine (JVM), main component of Java architecture, part of Java Running Environment.
- .NET infrastructure
  - CLR (Common Language Runtime).

Why we need virtual machine?

- **Code transportability**
  - Many high-level languages can work on the same platform: virtual machine.
  - VM may be implemented with relative ease on multiple target platforms.
  - As a result, VM-based software can run on many processors and operating systems without modifying source code.

Virtual machine paradigm



Many high-level languages

Compiler

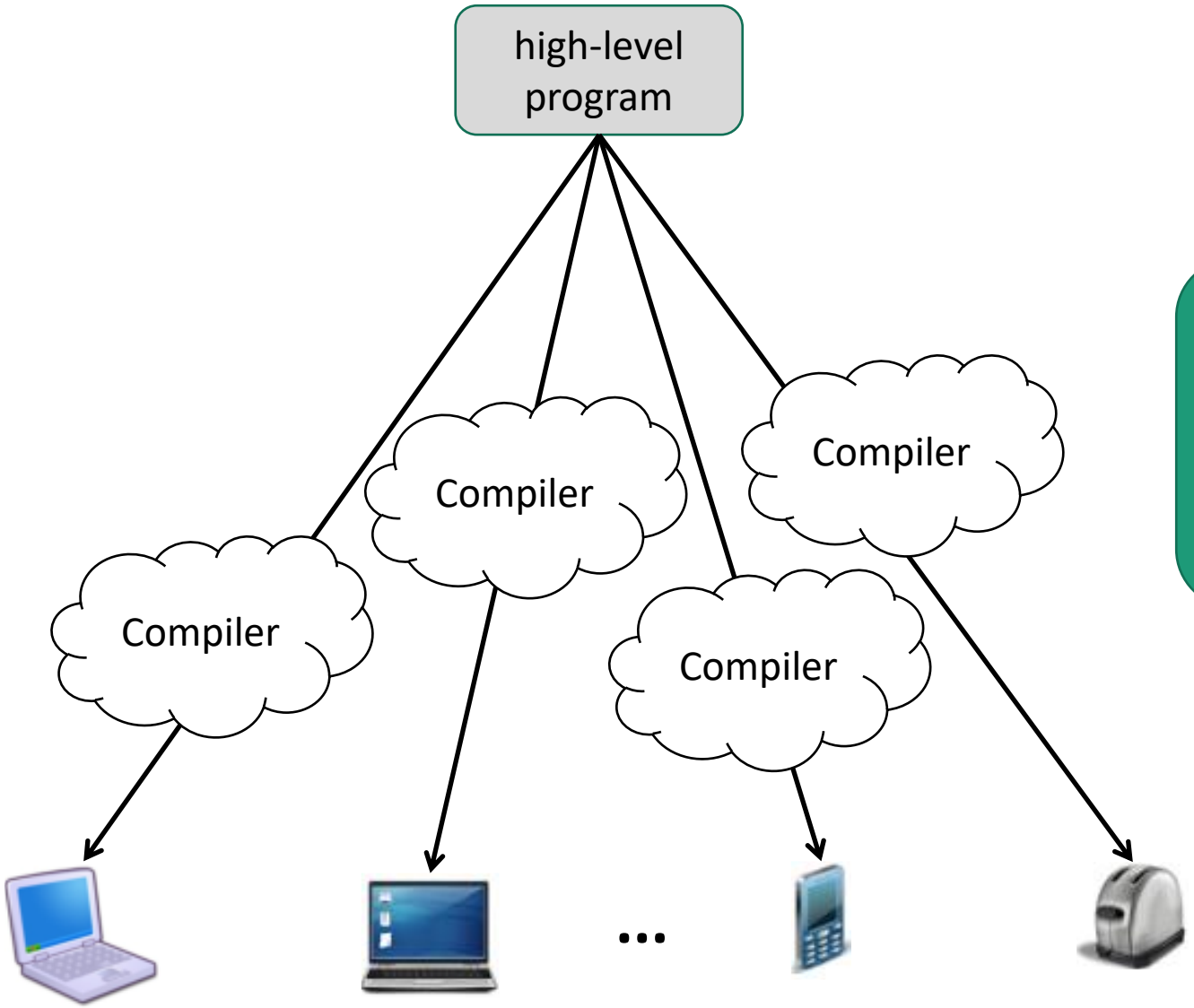
One virtual machine language

VM translator

Machine language

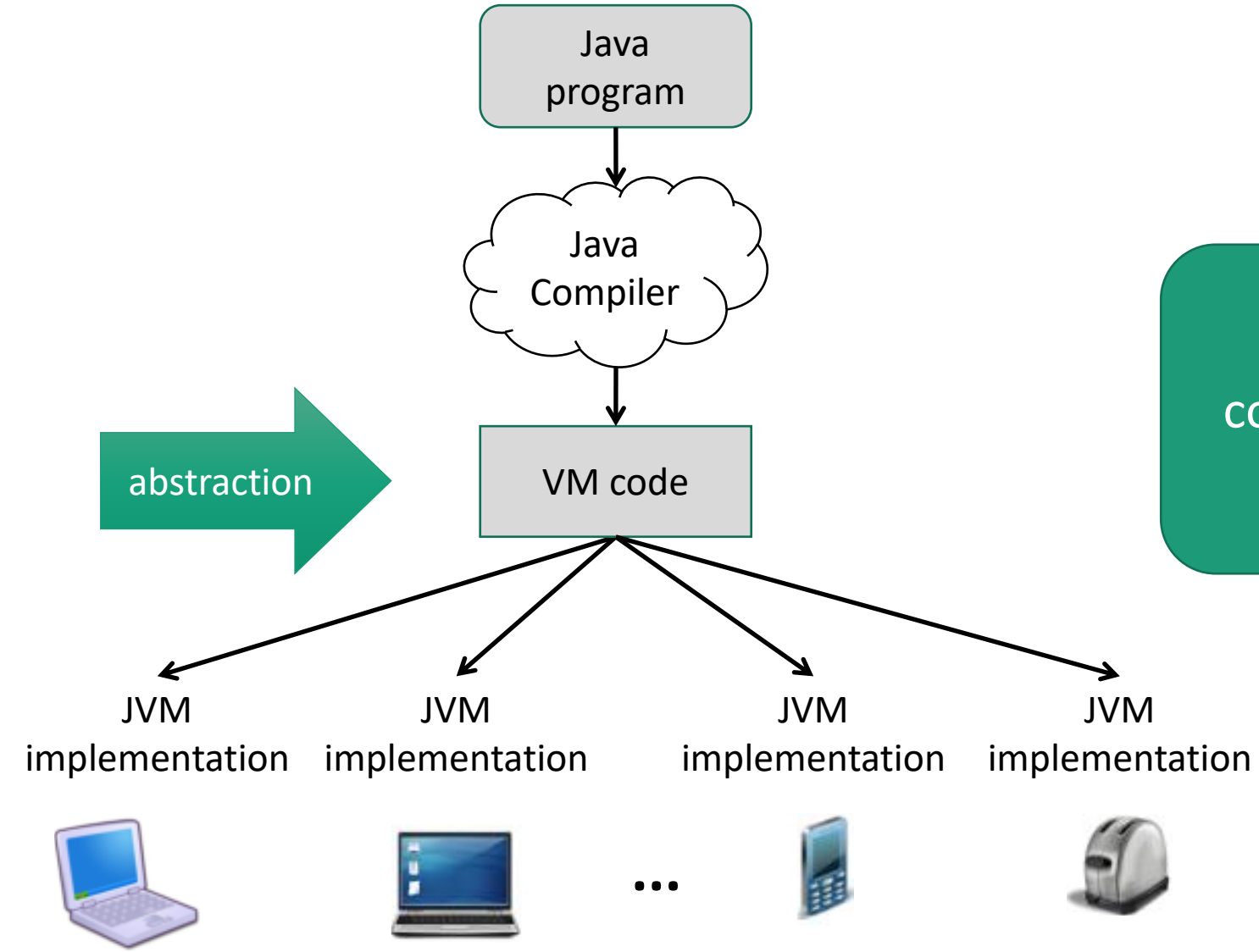
Many computer hardwares

Program compilation: 1-tier



One compiler for each device!

Program compilation: 2-tier

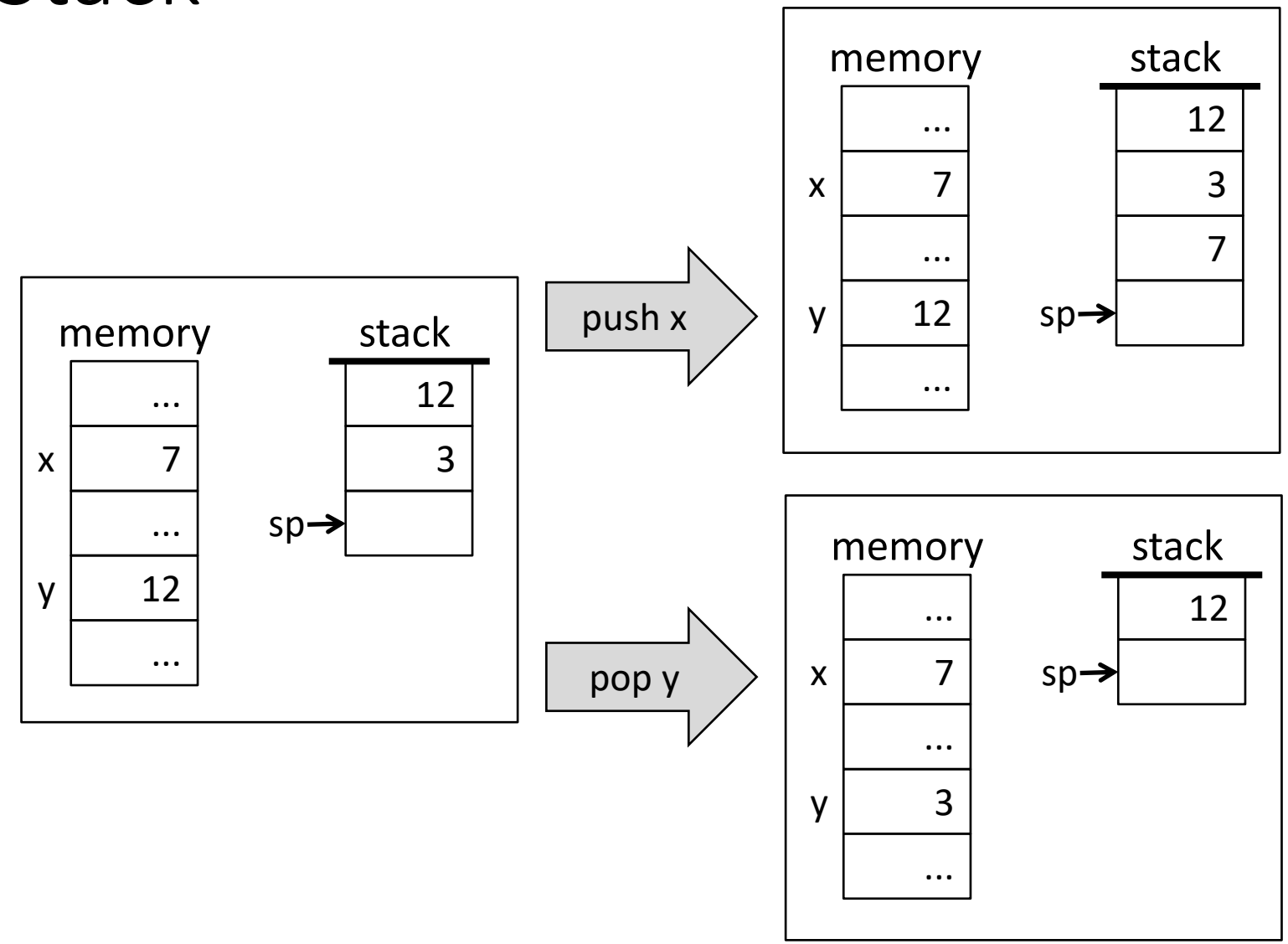


One Java compiler for all devices!

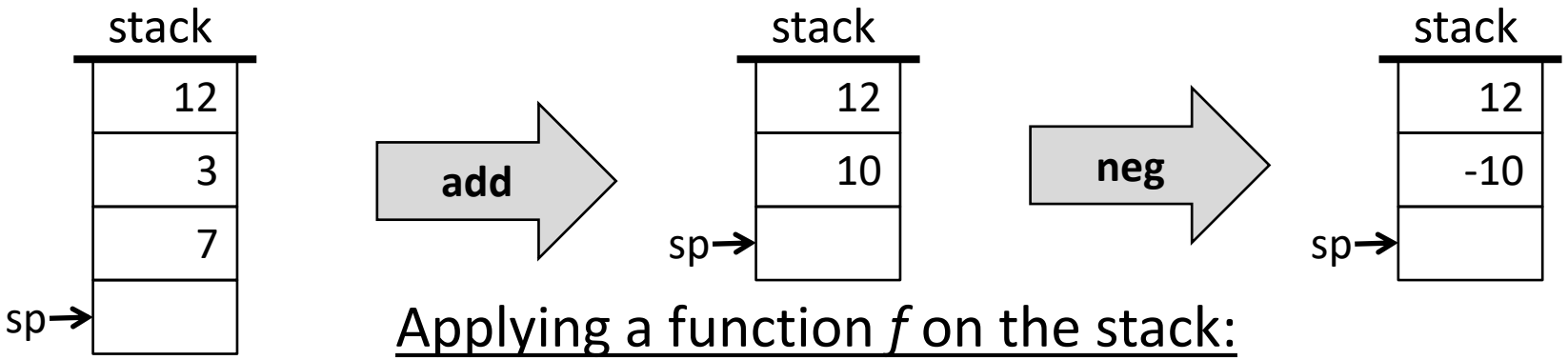
Stack machine model

- Question: where will the operands and the results of the VM operations reside?
- Answer: put them on a stack data structure.
- Stack:
  - push: add an element at the stack's **top**.
  - pop: remove the **top** element.

Stack

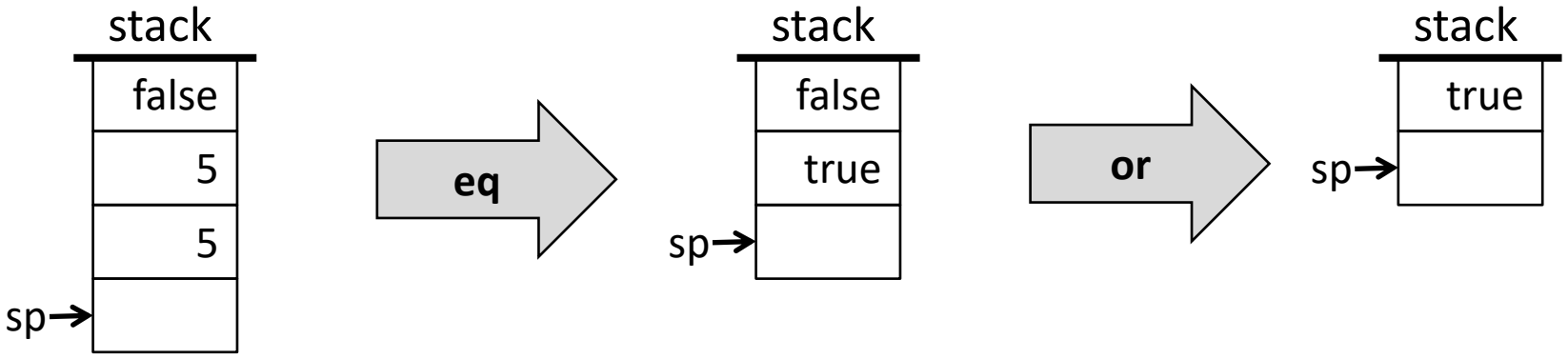

memory
stack
...
x
7
...
y
12
...
12
3
sp
push x
memory
stack
...
x
7
...
y
12
...
12
3
7
sp
pop y
memory
stack
...
x
7
...
y
3
...
12
sp

Stack arithmetic

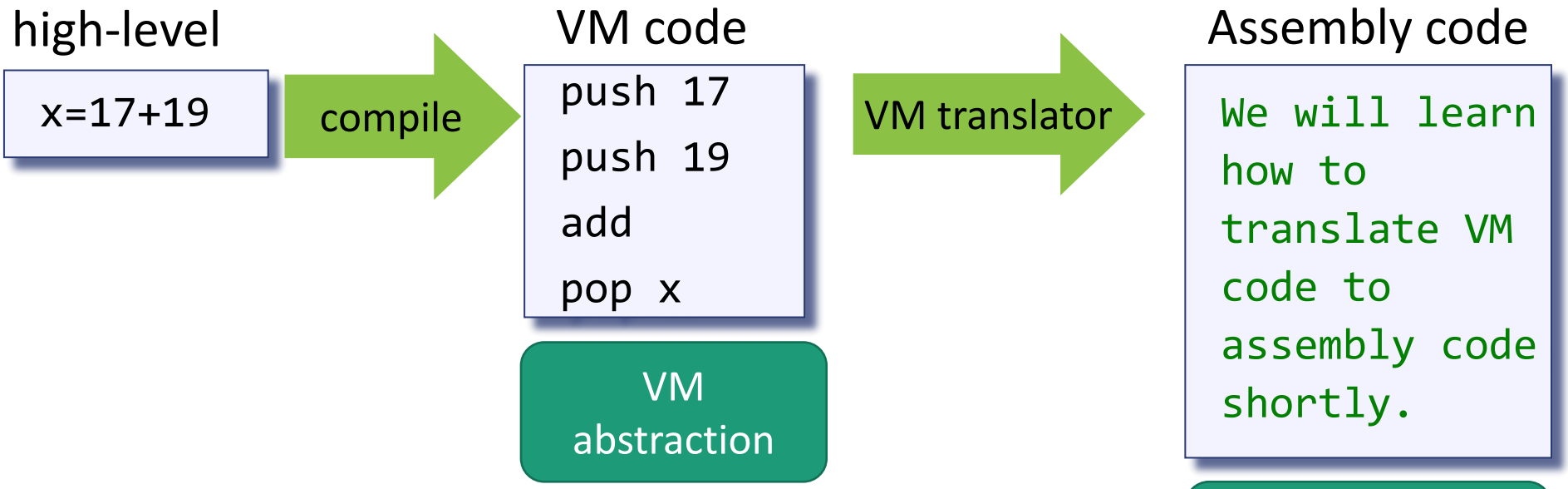


Applying a function *f* on the stack:

- **Pop** the argument(s) from the stack
- Compute *f* on the arguments
- **Push** the result onto the stack.



Virtual machine (big picture)



Abstraction / implementation

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

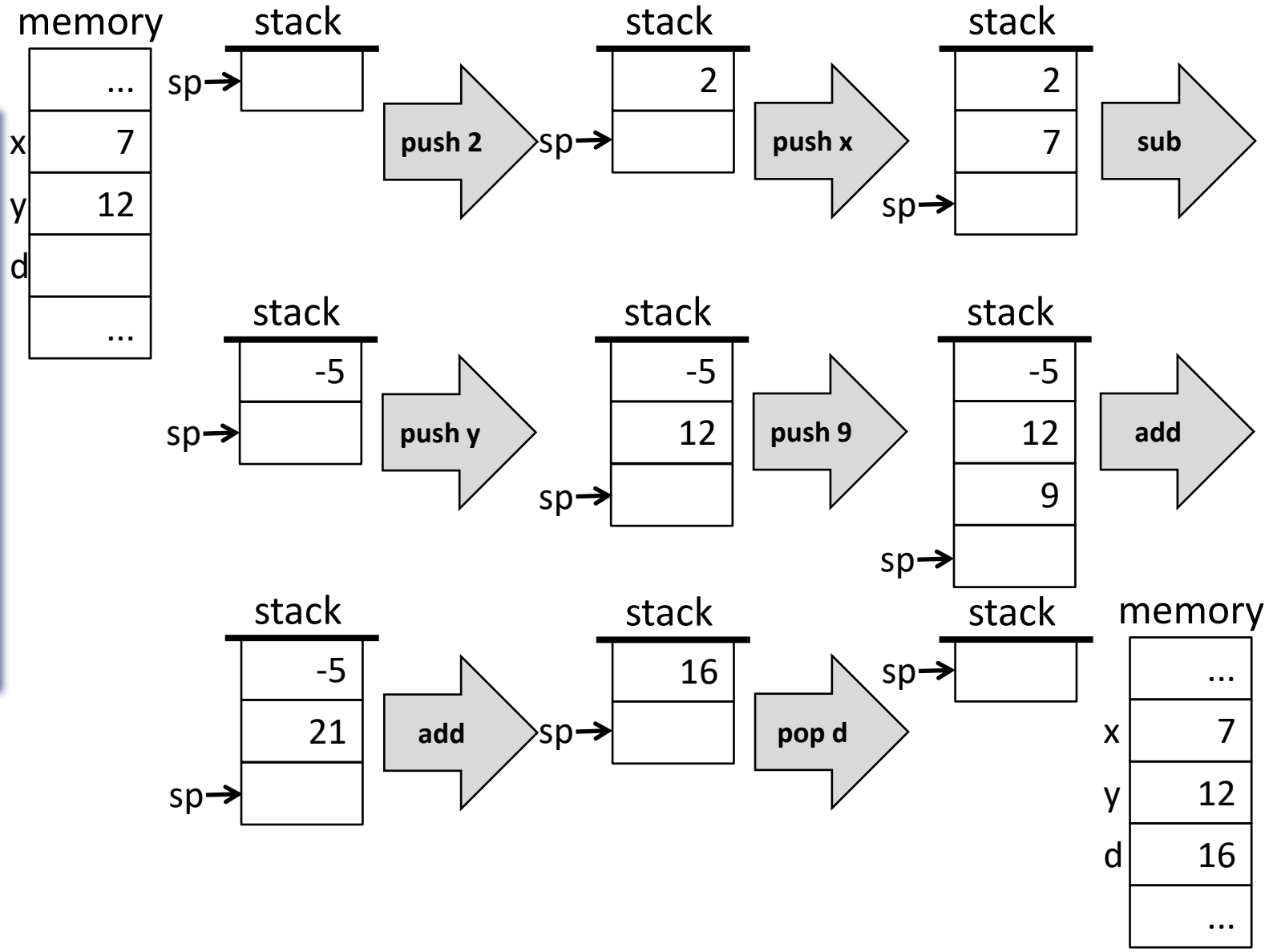
Outlines

- Introduction to virtual machine
- VM abstraction
  - Arithmetic / logical commands
  - Memory segment commands
  - Branching commands
  - Function commands
- VM implementation
- VM translator

Arithmetic commands

VM code

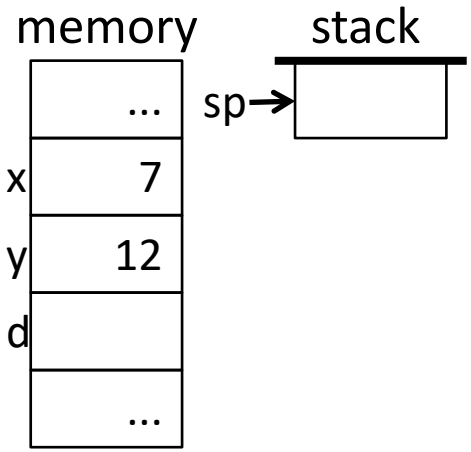
```
// d=(2-x) +
// (y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Quiz: arithmetic commands

VM code

```
// d=(x-5) -
// (y-6)
push x
push 5
sub
push y
push 6
sub
sub
pop d
```

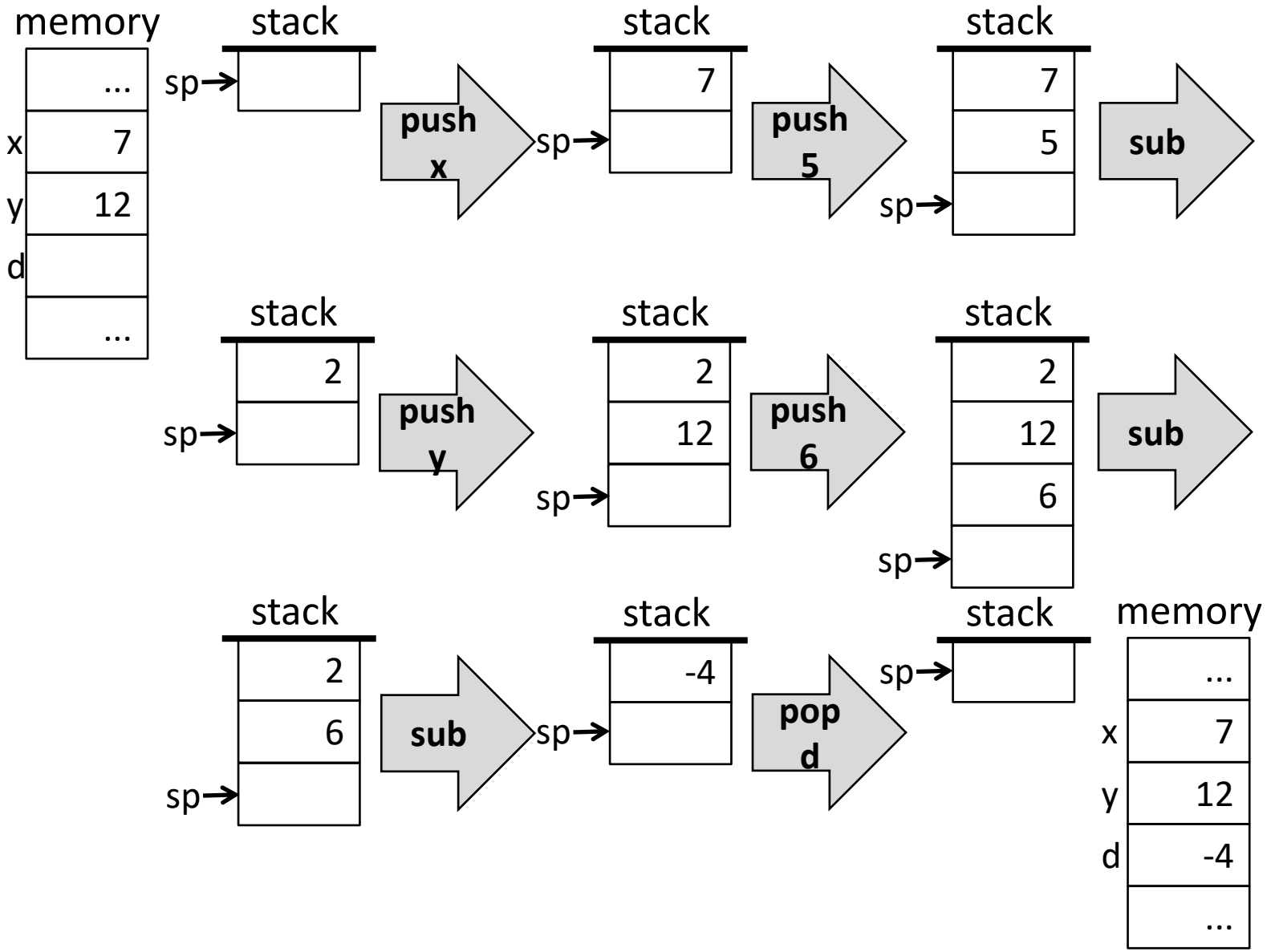


Complete the stack operation as last example and write down each key step.

Answer: arithmetic commands

VM code

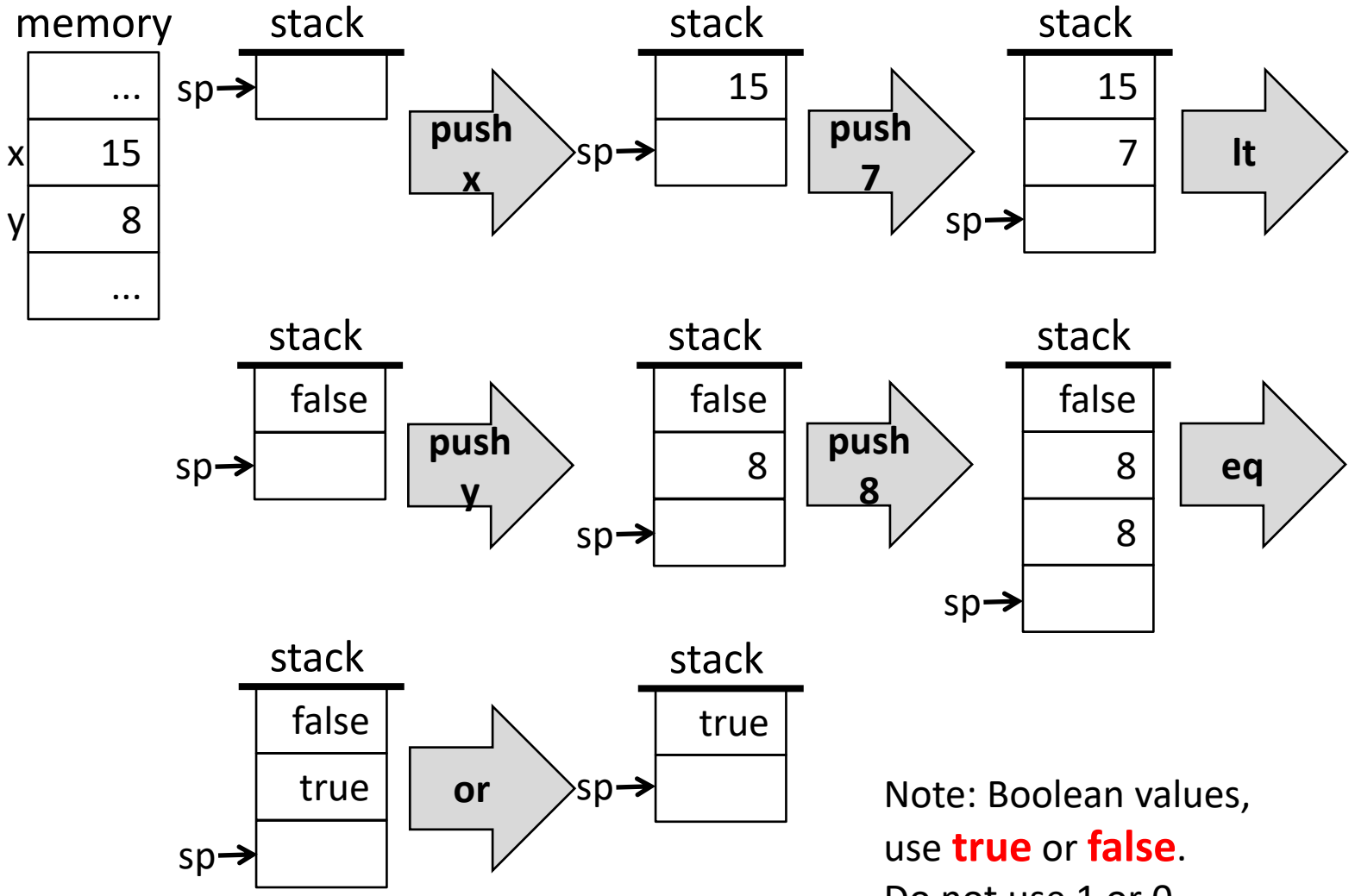
```
// d=(x-5) -
// (y-6)
push x
push 5
sub
push y
push 6
sub
sub
pop d
```



Logical commands

VM code

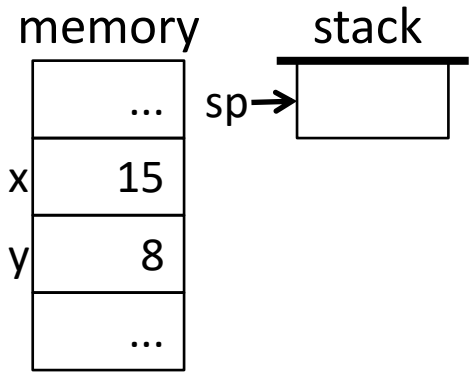
```
// (x<7)
// or
// (y==8)
push x
push 7
lt
push y
push 8
eq
or
```



Quiz: logical commands

VM code

```
// (x>7)
// and
// (y>7)
push x
push 7
gt
push y
push 7
gt
and
```

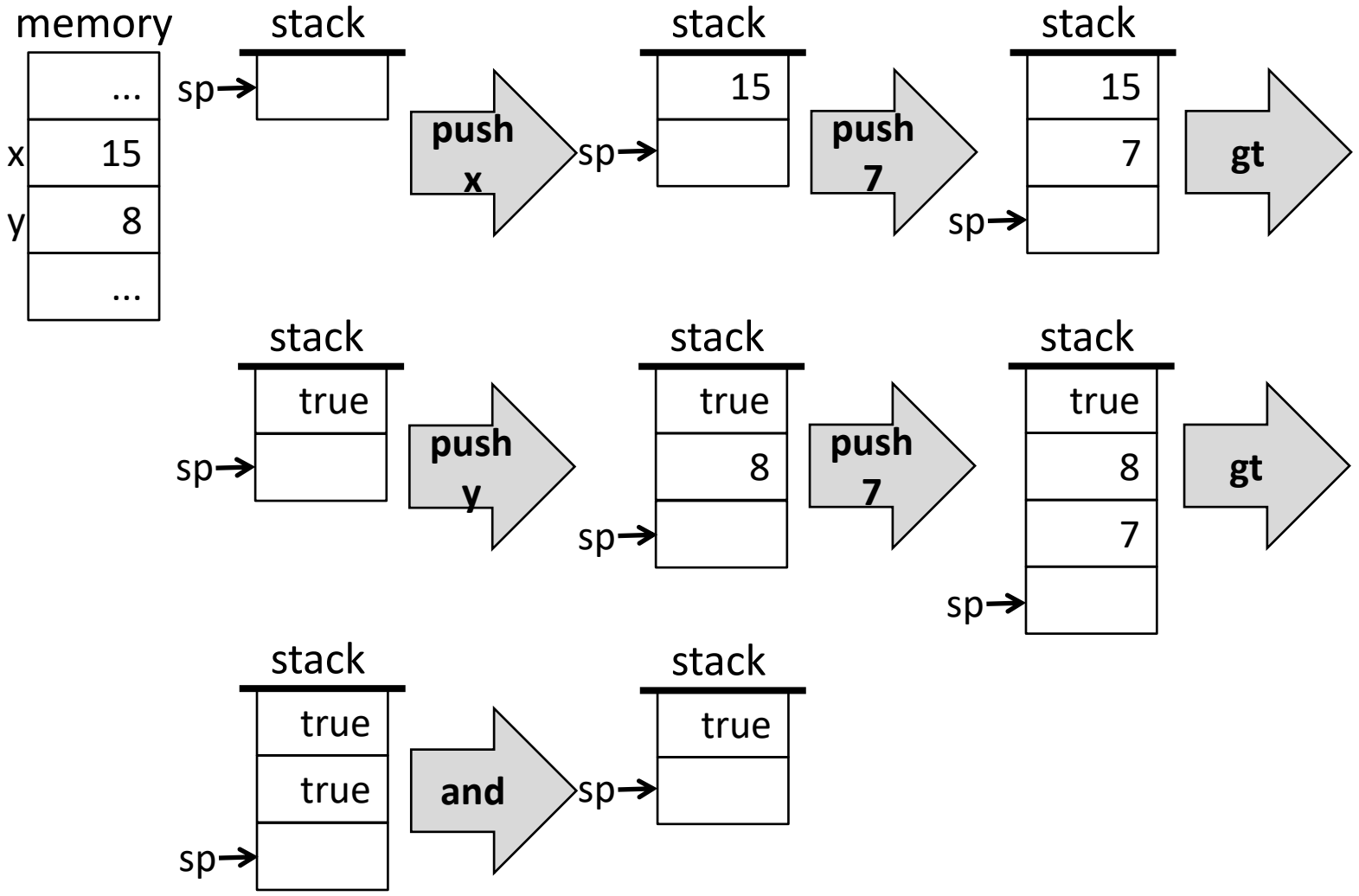


Complete the stack operation as last example and write down each key step.

Answer: logical commands

VM code

```
// (x>7)
// and
// (y>7)
push x
push 7
gt
push y
push 7
gt
and
```



Arithmetic / Logical commands

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

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

Outlines

- Introduction to virtual machine
- VM abstraction
  - Arithmetic / logical commands
  - Memory segment commands
  - Branching commands
  - Function commands
- VM implementation
- VM translator

Variable kinds and memory segments

Source code (Jack)

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

compile

Compiled VM code

```
...
...
...
...
push s1      static 0
push y       argument 1
add
pop c        local 2
...
```

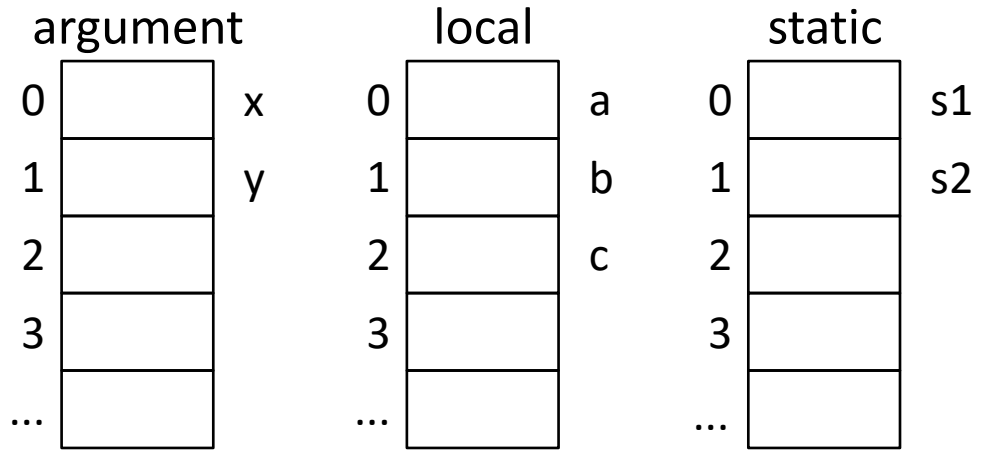
Following compilation, all the symbolic references are replaced with references to virtual memory segments.

Variable kinds

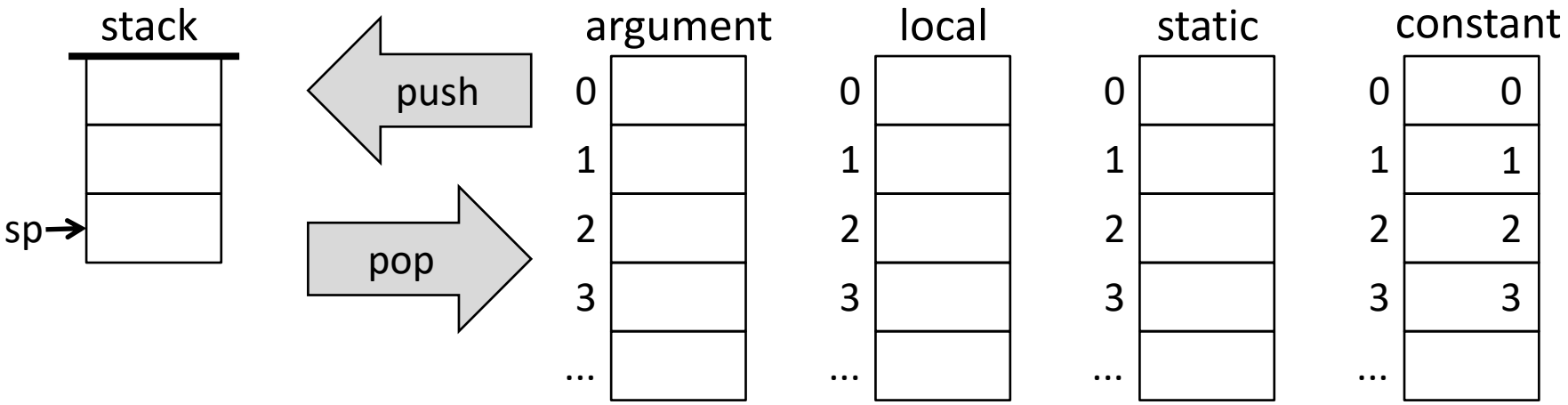
- Argument variables
- Local variables
- Static variables

(More kinds later)

Virtual memory segments:

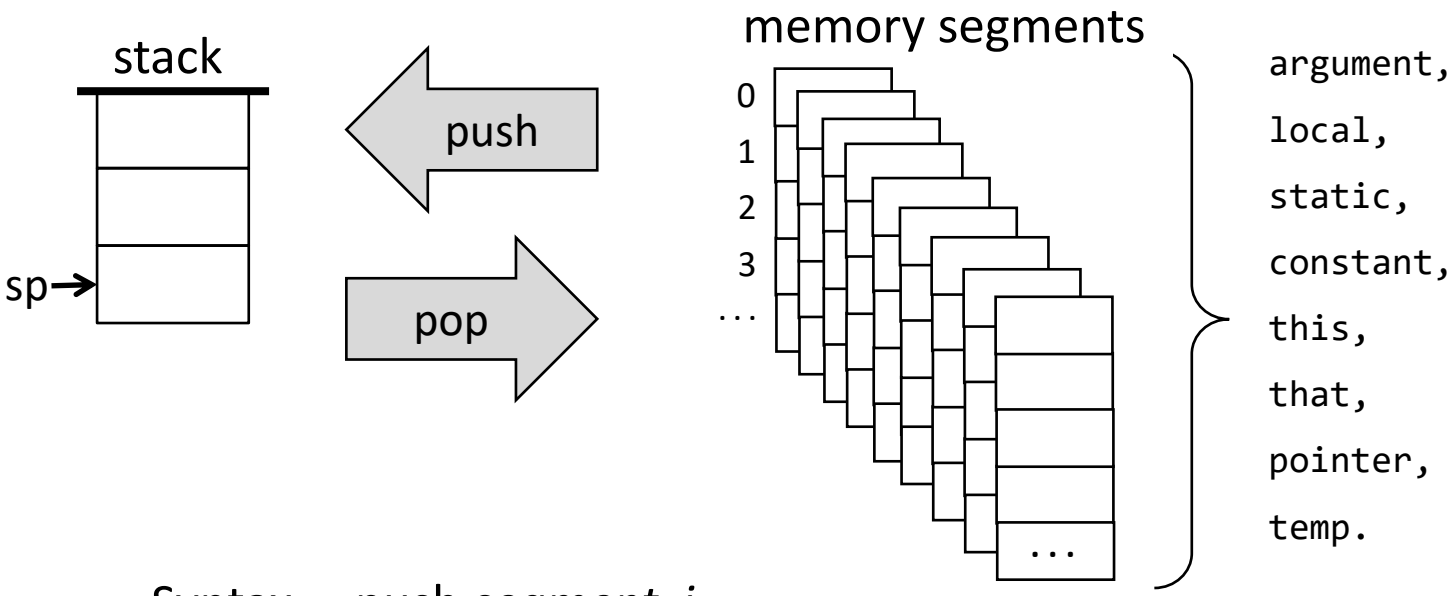


Memory segments



- Syntax: push / pop segment i
- Examples:
  - push constant 17
  - pop local 2
  - pop static 5
  - push argument 3

Memory segments



Syntax: push *segment i*

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

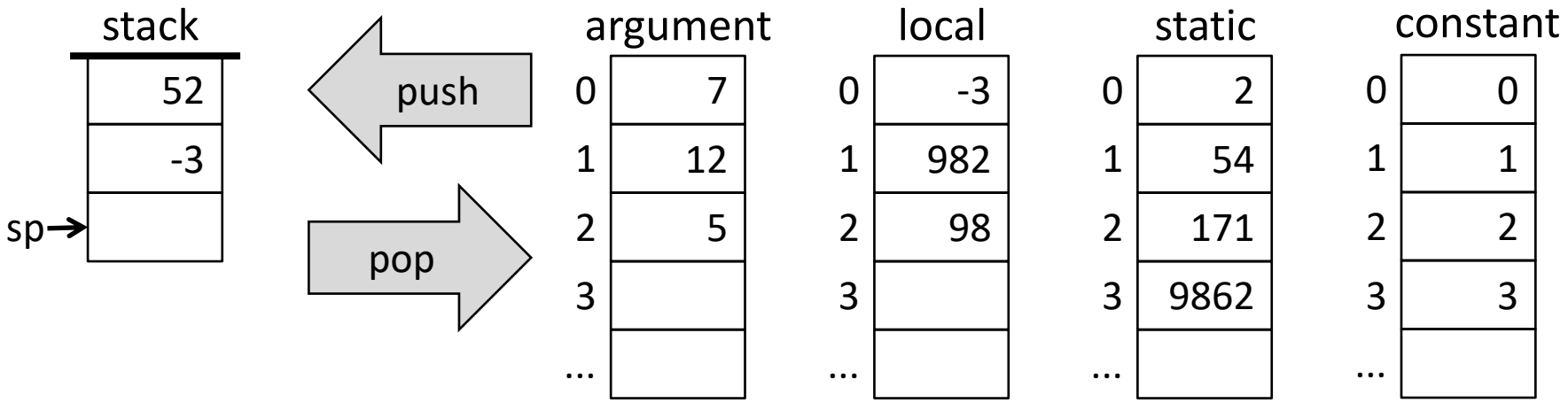
and *i* is a non-negative integer.

Syntax: pop *segment i*

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

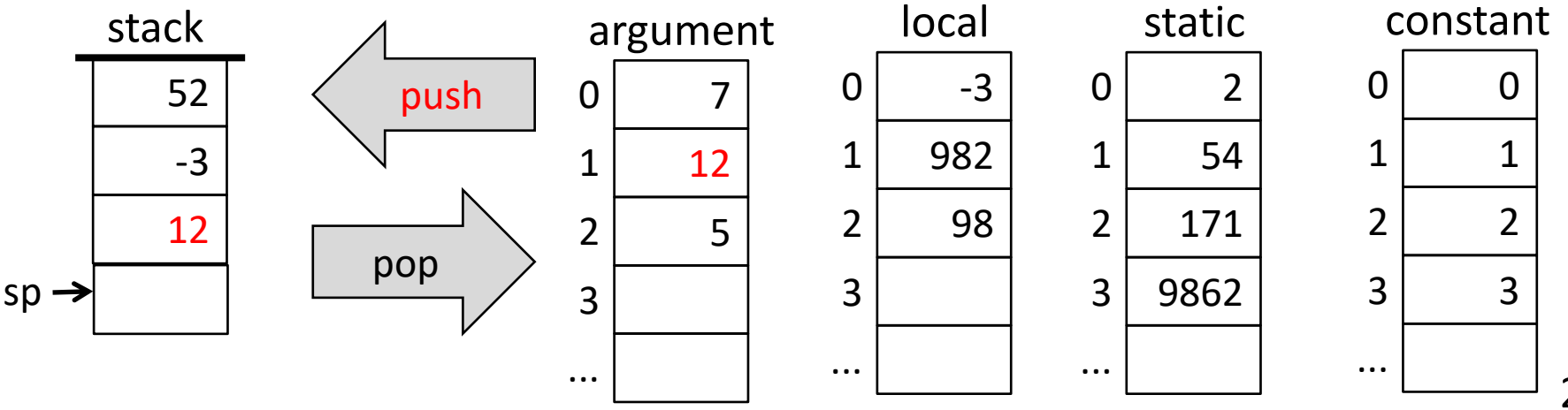
and *i* is a non-negative integer.

Memory segment commands

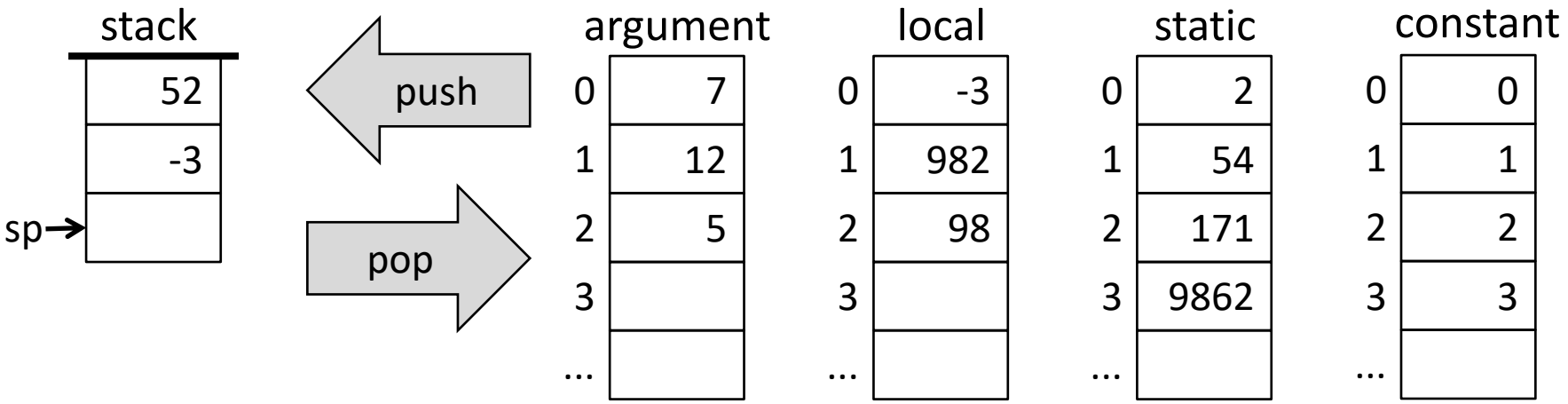


let static 2 = argument 1

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

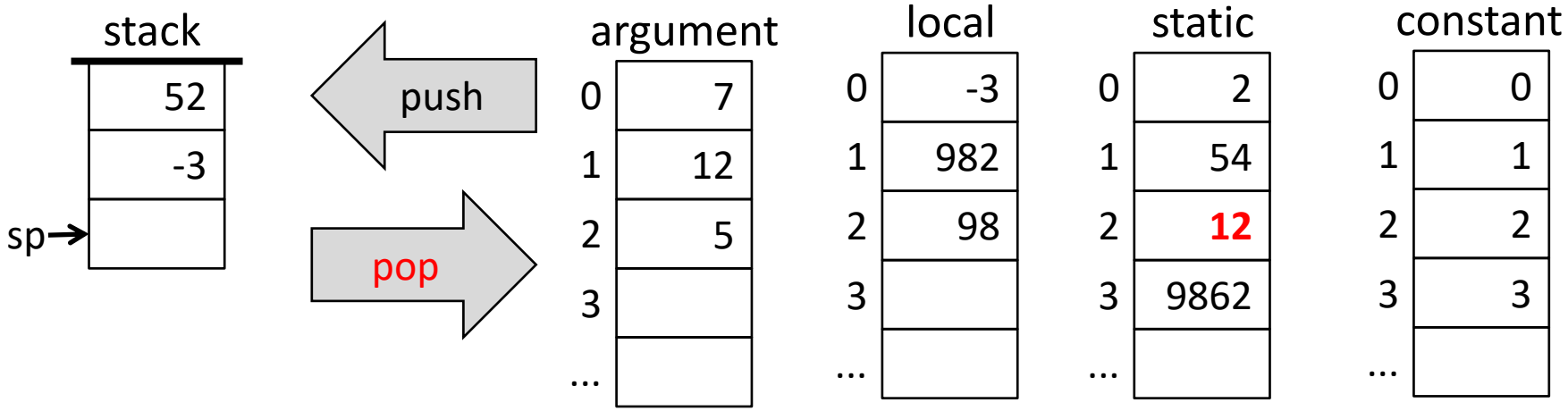


Memory segment commands

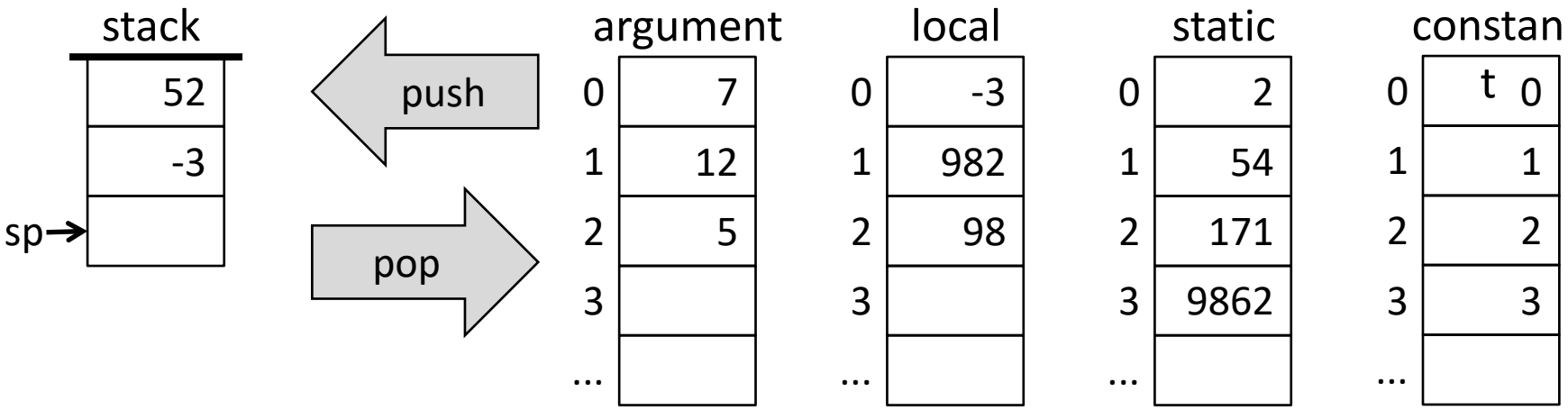


let static 2 = argument 1

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



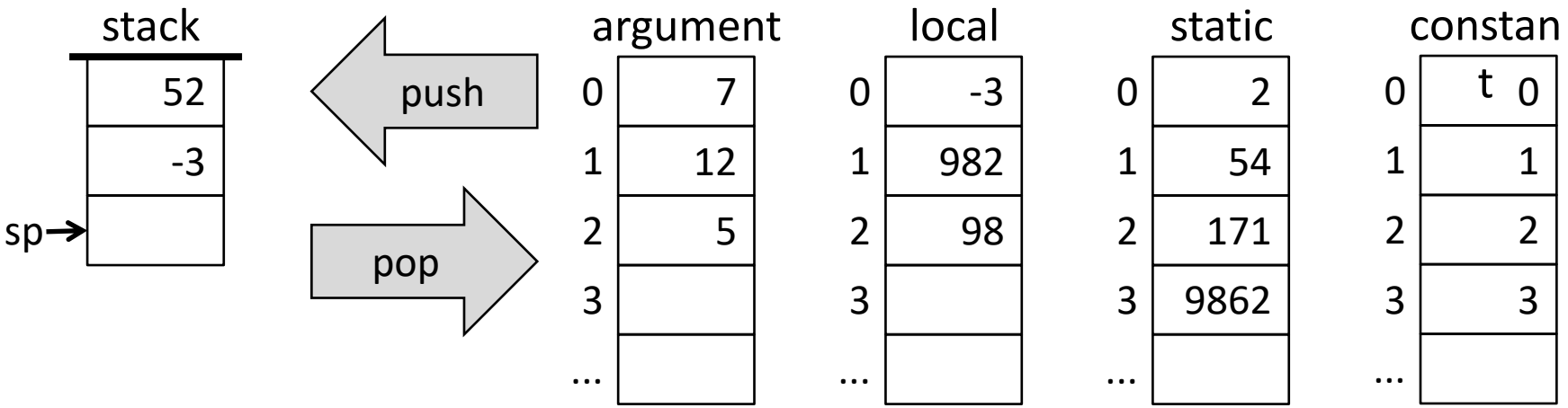
Quiz: memory segment commands



let local 2 = argument 1

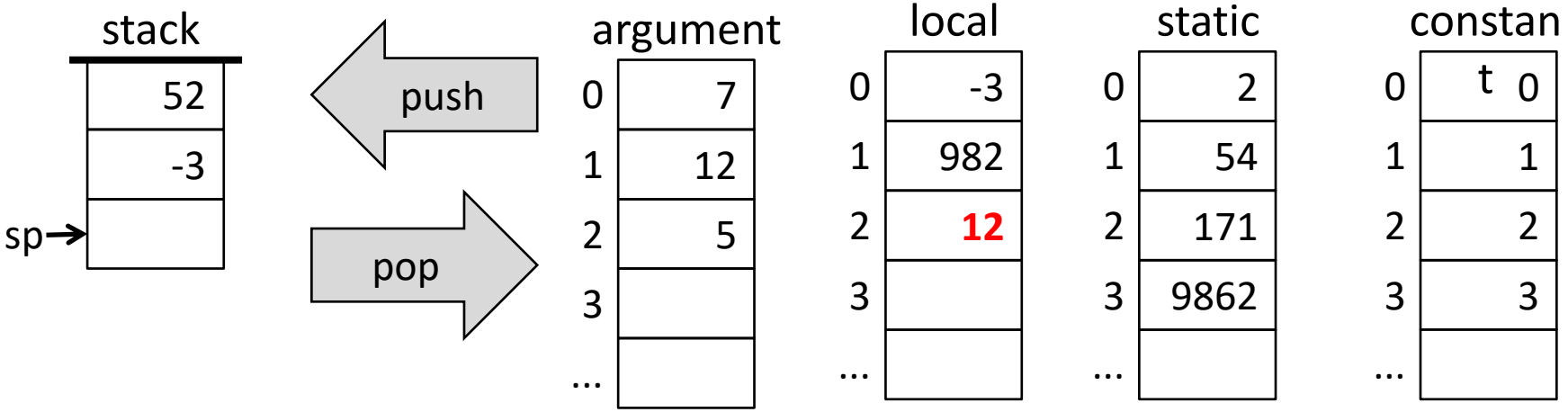
How do stack and memory segments change?
Please write down the key steps for stack operation and memory changes, similarly as last example.

Answer: memory segment commands



let local 2 = argument 1

```
push argument 1
pop  local 2
```



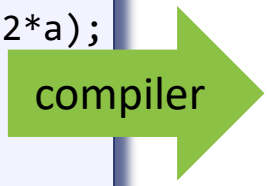
Outlines

- Introduction to virtual machine
- VM abstraction
  - Arithmetic / logical commands
  - Memory segment commands
  - Branching commands
  - Function commands
- VM implementation
- VM translator

Program control

High-level code

```
if !(a==0)
  x=(-b+sqrt(disc(a,b,c)))/(2*a);
else
  x=-c/b;
// code continues
```



VM code (pseudo)

```
    push a
    push 0
    eq
    not
    if-goto A_NEQ_ZERO
    // We get here if a==0
    push c
    neg
    push b
    call div
    pop x
    goto CONTINUE
label A_NEQ_ZERO
    // We get here if !(a==0)
    push b
    neg
    push a
    push b
    push c
    call disc
    call sqrt
    add
    push 2
    push a
    call mult
    call div
    pop x
label CONTINUE
    // code continues
```

branching

function calls

Program control

High-level code

```
if !(a==0)
  x=(-b+sqrt(disc(a,b,c)))/(2*a);
else
  x=-c/b;
// code continues
```

VM code (pseudo)

```
    push a
    push 0
    eq
    not
    if-goto A_NEQ_ZERO
    // We get here if a==0
    push c
    neg
    push b
    call div
    pop x
    goto CONTINUE
label A_NEQ_ZERO
    // We get here if !(a==0)
    push b
    neg
    push a
    push b
    push c
    call disc
    call sqrt
    add
    push 2
    push a
    call mult
    call div
    pop x
label CONTINUE
    // code continues
```

Program control

VM branching commands:

- goto *label*
- if-goto *label*
- label *label*

VM function commands:

- call *function*
- function *function*
- return

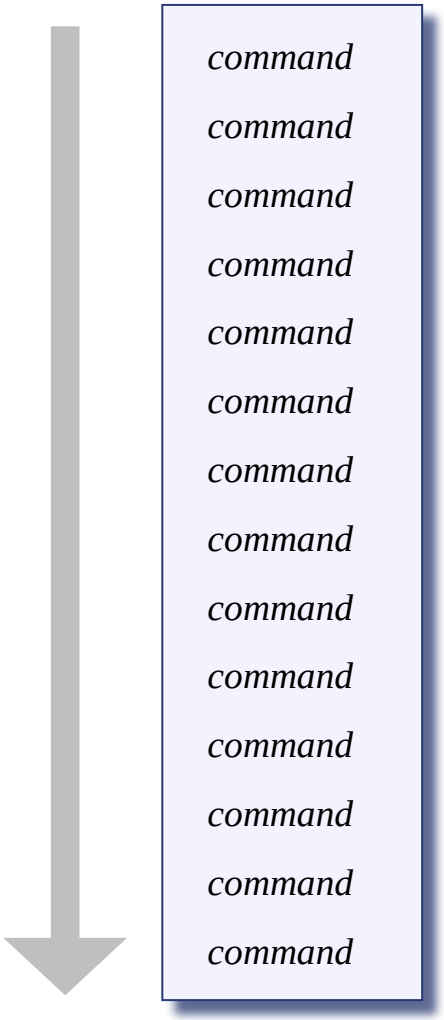
Challenges:

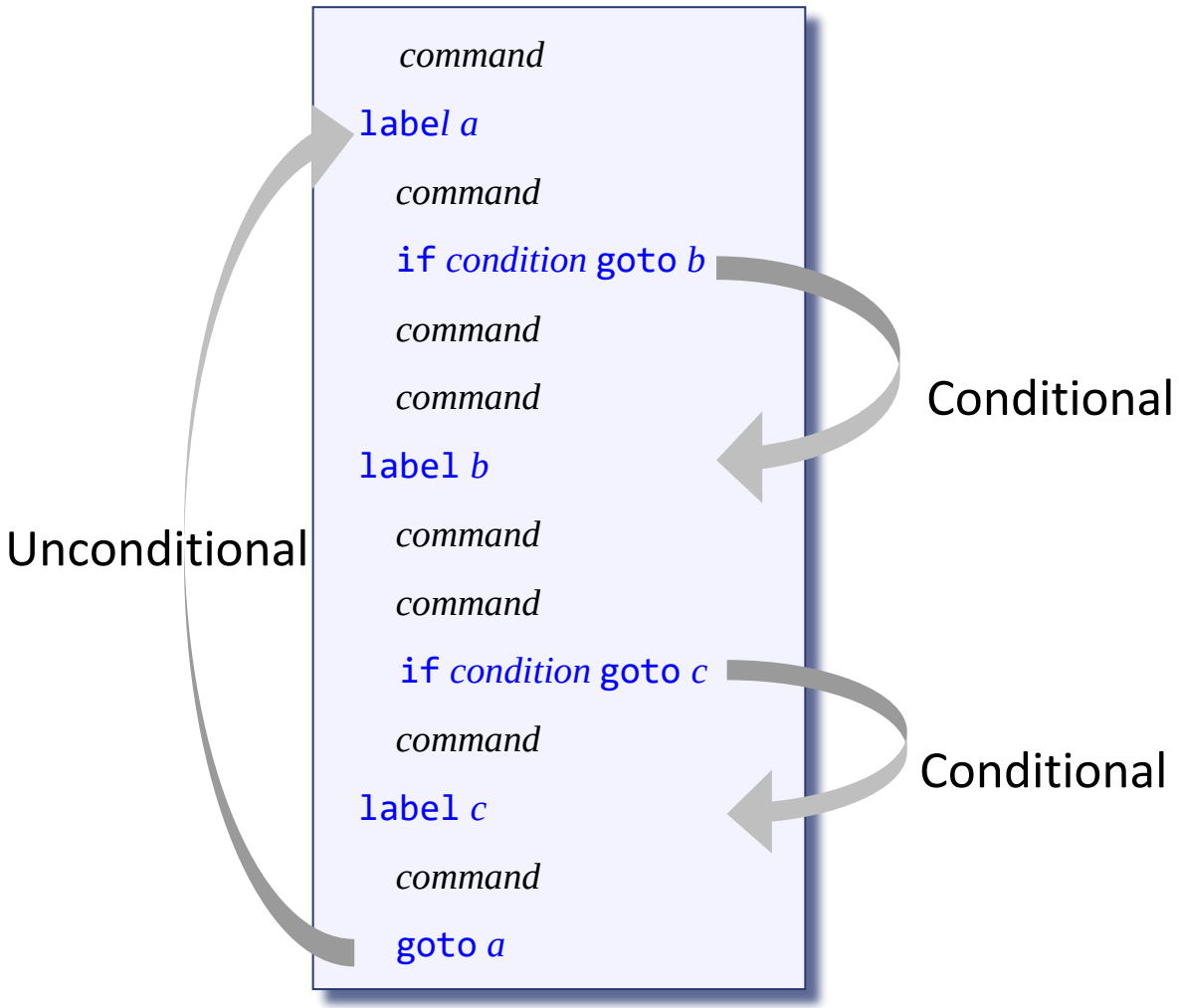
- Understand what the commands do (abstraction),
- Realize the commands on the host platform (implementation).

VM code (pseudo)

```
    push a
    push 0
    eq
    not
    if-goto A_NEQ_ZERO
    // We get here if a==0
    push c
    neg
    push b
    call div
    pop x
    goto CONTINUE
label A_NEQ_ZERO
    // We get here if !(a==0)
    push b
    neg
    push a
    push b
    push c
    call disc
    call sqrt
    add
    push 2
    push a
    call mult
    call div
    pop x
label CONTINUE
    // code continues
```

Branching

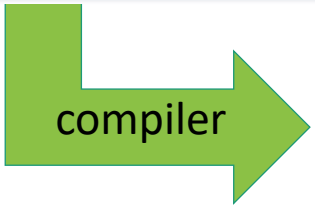

command
command
command
command
command
command
command
command
command
command
command
command
command
command


command
label a
command
if condition goto b
command
command
label b
command
command
if condition goto c
command
label c
command
goto a
Unconditional
Conditional
Conditional

Branching

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;
}
```



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

Conditional branching:

`if-goto` *label*

VM logic:

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

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

Recap

- goto *label*
  - jump to execute the command just after *label*
- if-goto *label*
  - *cond* = push
  - if *cond* jump to execute the command just after *label*
- label *label*
  - label declaration command
- Implementation (VM translation):
  - Translate each branching command into assembly instructions that effect the specified operation on the host machine.

The assembly language has similar branching commands.

Outlines

- Introduction to virtual machine
- VM abstraction
  - Arithmetic / logical commands
  - Memory segment commands
  - Branching commands
  - Function commands
- VM implementation
- VM translator

Functions

- High-level programming languages can be extended using:
  - Subroutines
  - Functions
  - Procedures
  - Methods

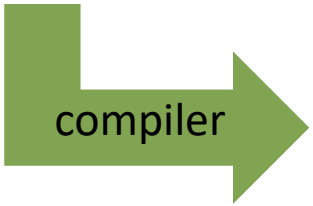
  } functions

  (different names of the same thing)

Functions in VM language

High-level program

```
...
sqrt(x – 17 + x * 5)
...
```



Pseudo VM

```
...
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, ...

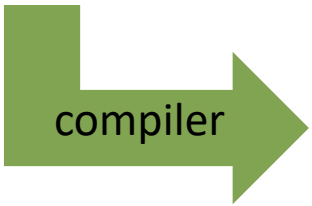
Programming style:

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

Functions in VM language: defining

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;
}
```



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 in VM language: executing

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

caller

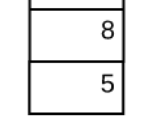
```
// Computes the product of two given arguments
0 function mult 2 // 2 local vars used in this func.
1   push constant 0
2   pop local 0
3   push constant 1
4   pop local 1
5 label LOOP
6   push local 1
7   push argument 1
    //... computes the product into local 0
19 label END
20   push local 0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

callee

`main` view:

after line 3 is executed:

stack



after line 4 is executed:

`mult` view:

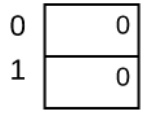
after line 0 is executed:

stack

(empty)

argument

local



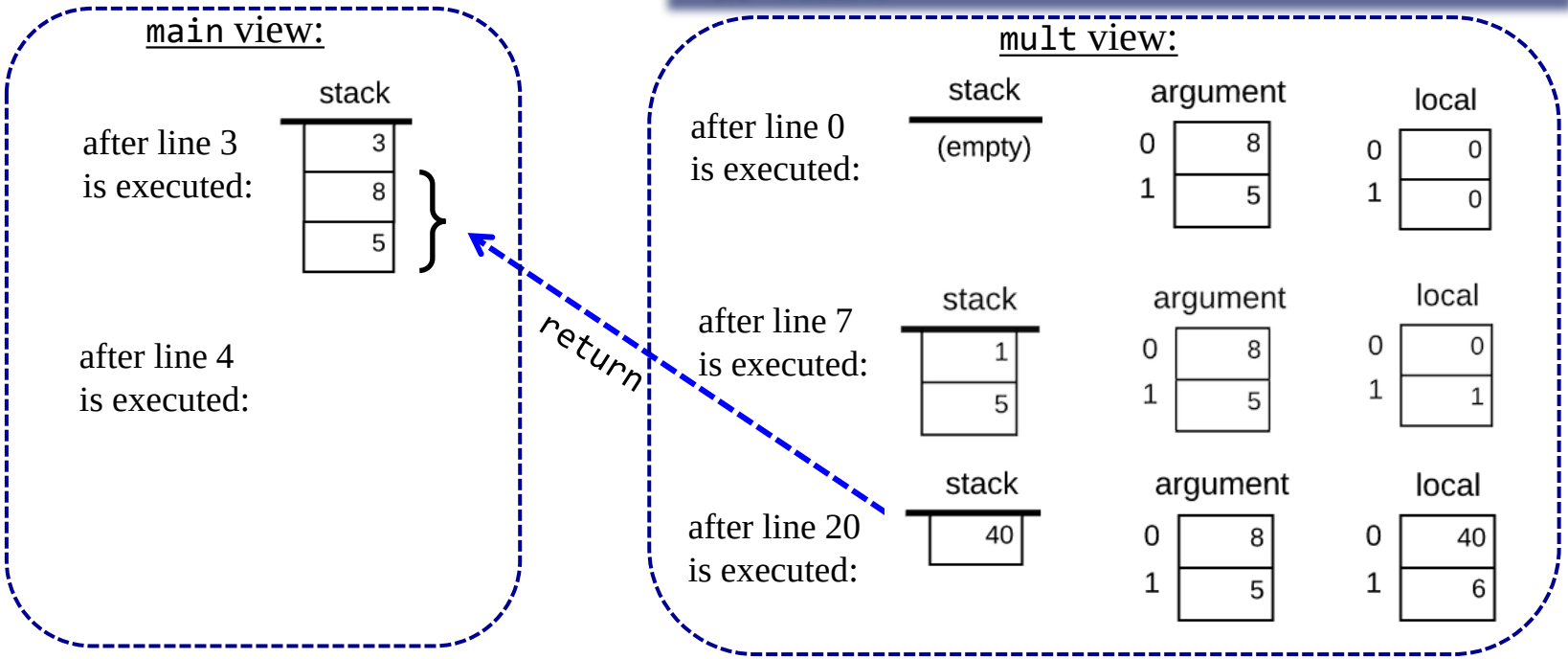
Functions in VM language: executing

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

caller

```
// Computes the product of two given arguments
0 function mult 2 // 2 local vars used in this func.
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 in VM language: executing

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

caller

```
// Computes the product of two given arguments
0 function mult 2 // 2 local vars used in this func.
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

after line 4 is executed:

stack: 3, 40

after line 5 is executed:

stack: 43

mult view:

after line 0 is executed:

stack: (empty)

argument: 0 → 8, 1 → 5

local: 0 → 0, 1 → 0

after line 7 is executed:

stack: 1, 5

argument: 0 → 8, 1 → 5

local: 0 → 0, 1 → 1

after line 20 is executed:

stack: 40

argument: 0 → 8, 1 → 5

local: 0 → 40, 1 → 6

Functions in VM language: implementation

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

caller

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

callee

Implementation

We can write low-level code to

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

Functions in VM language: implementation

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

caller

```
// Computes the product of two given arguments
0 function mult 2 // 2 local vars used in this func.
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

Handling function call:

- Determine the **return address** within the *caller*'s code;
- **Save** the *caller*'s return address, stack and memory segments;
- **Pass parameters** from the *caller* to the *callee*;
- **Jump** to execute the *callee*.

Functions in VM language: implementation

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

caller

```
// Computes the product of two given arguments
0 function mult 2 // 2 local vars used in this func.
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

Handling function:

- Initialize the local variables of the *callee*;
- Handle some other simple initializations (later);
- Execute the *callee* function.

Functions in VM language: implementation

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

caller

```
// Computes the product of two given arguments
0 function mult 2 // 2 local vars used in this func.
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

Handling return:

(a function always ends by pushing a return value on the stack)

- **Return** the *return value* to the *caller*;
- **Recycle** the memory resources used by the *callee*;
- **Reinstate** the *caller*'s stack and memory segments;
- **Jump** to the return address in the *caller*'s code.

Summary

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