Assembler – Perspective

Assembly elements

- Mnemonics (mov, add, etc)
- Data directives (strings, etc)
- Assembly directives (labels, variables, comments)

```
; read a byte from stdin in Linux
mov eax, 3 ; 3 is recognized by the system as meaning "read"
mov ebx, 0 ; read from standard input
mov ecx, variable ; address to pass to – byte stored here
mov edx, 1 ; input length (one byte)
int 0x80 ; call the kernel
```

Many different assemblers

- Different directives
- Different calling conventions for different operating systems
  - Pass parameters in registers
  - Pass parameters on stack

Assembler - Perspective

Why programming in machine language / assembly?

- Whenever optimized code is needed
  - for speed (large computations)
  - for memory space (embedded systems, sensor systems)
- Hardware-near programming (peripherals)
- In-line assembly as extension of high-level language code

Machine language

- Typically not visible to the programmers
- Often not public for special purpose processors

Understanding an Assembler is an excellent practice before meeting more challenging translators, e.g. a VM Translator and a compiler, as we will see in the next lectures.

Assembler - Perspective

Translating from Assemble to Machine language was simple:

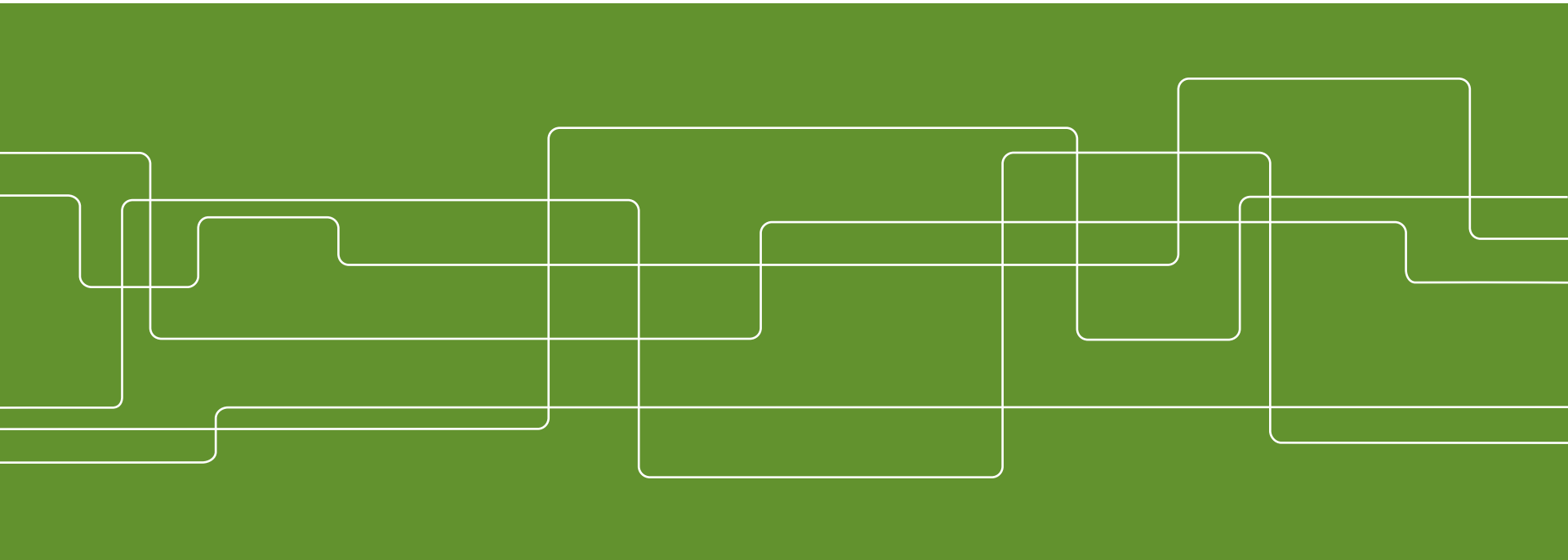
- Each comment is translated to 16 bits
- Simple program flow
- Global, 16-bit variables, simple memory management

What to expect in the next step towards high level languages?

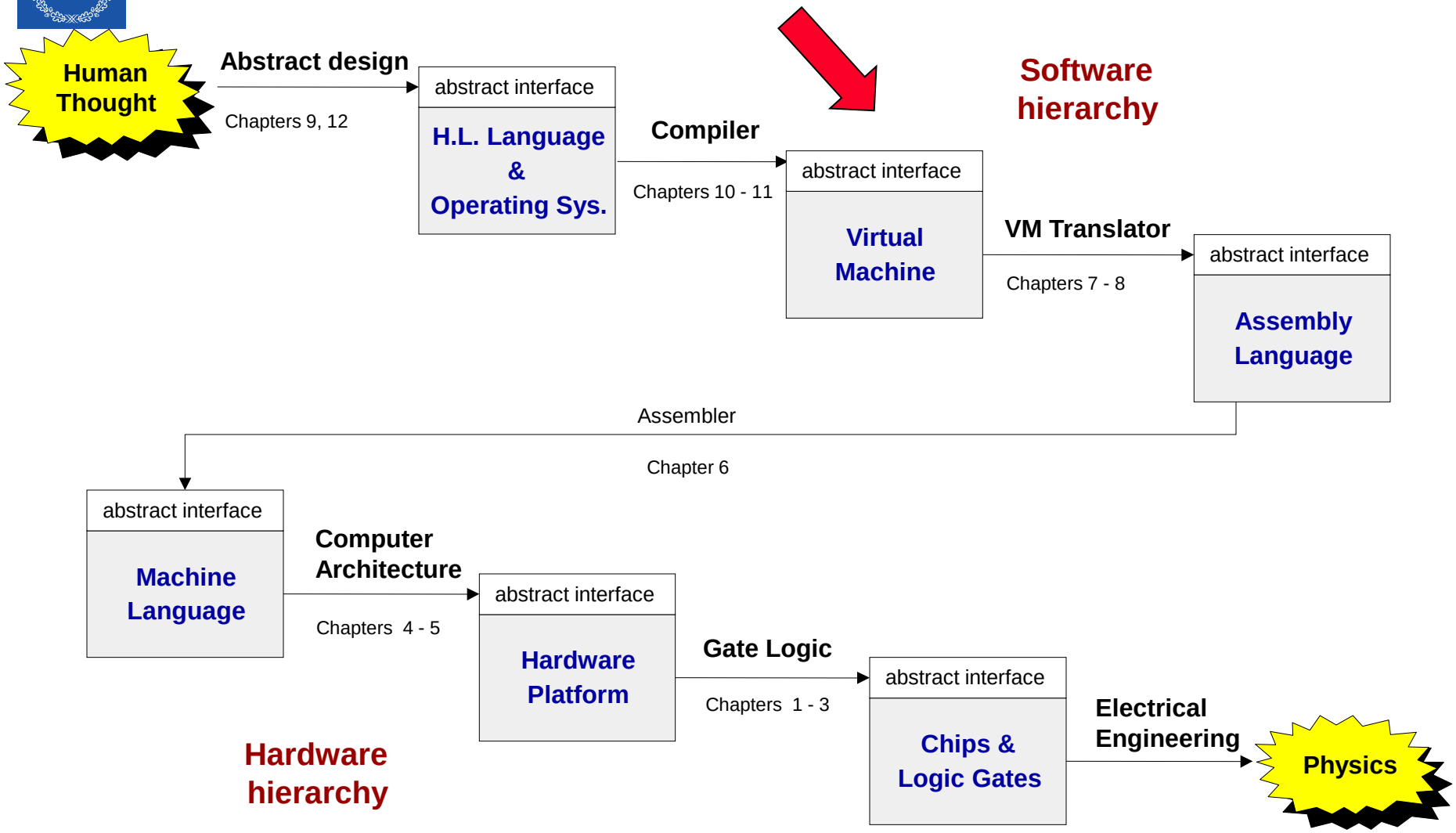
- More complex arithmetic tasks
- Array handling
- Object handling
- Program flow commands
- Subroutine calling, local variables

EP1200 Introduction to Computing Systems Engineering

Virtual Machine I



Where we are at:



Motivation

Jack code (example)

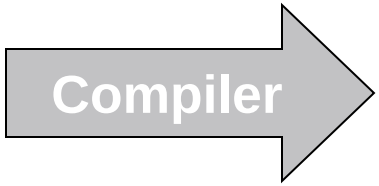
```
class Main {
  static int x;

  function void main() {
    // Inputs and multiplies two numbers
    var int a, b, x;
    let a = Keyboard.readInt("Enter a number");
    let b = Keyboard.readInt("Enter a number");
    let x = mult(a,b);
    return;
  }
}

  // Multiplies two numbers.
  function int mult(int x, int y) {
    var int result, j;
    let result = 0; let j = y;
    while ~(j = 0) {
      let result = result + x;
      let j = j - 1;
    }
    return result;
  }
}
```

Our ultimate goal:

Translate high-level programs into executable code.

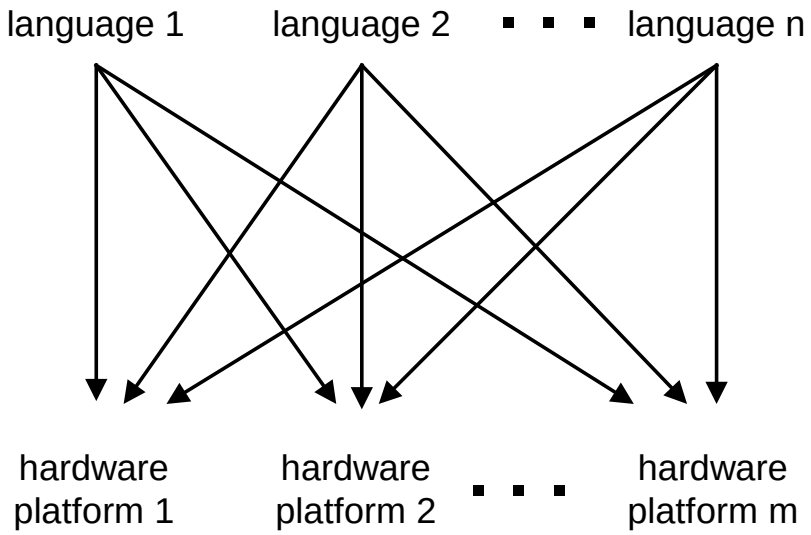


Hack code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
...
```

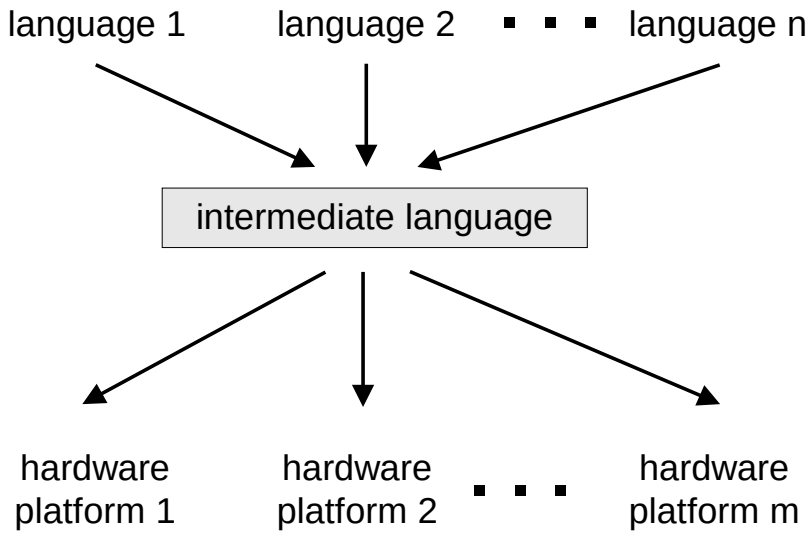
Compilation models

direct compilation:



requires $n \cdot m$ translators

2-tier compilation:

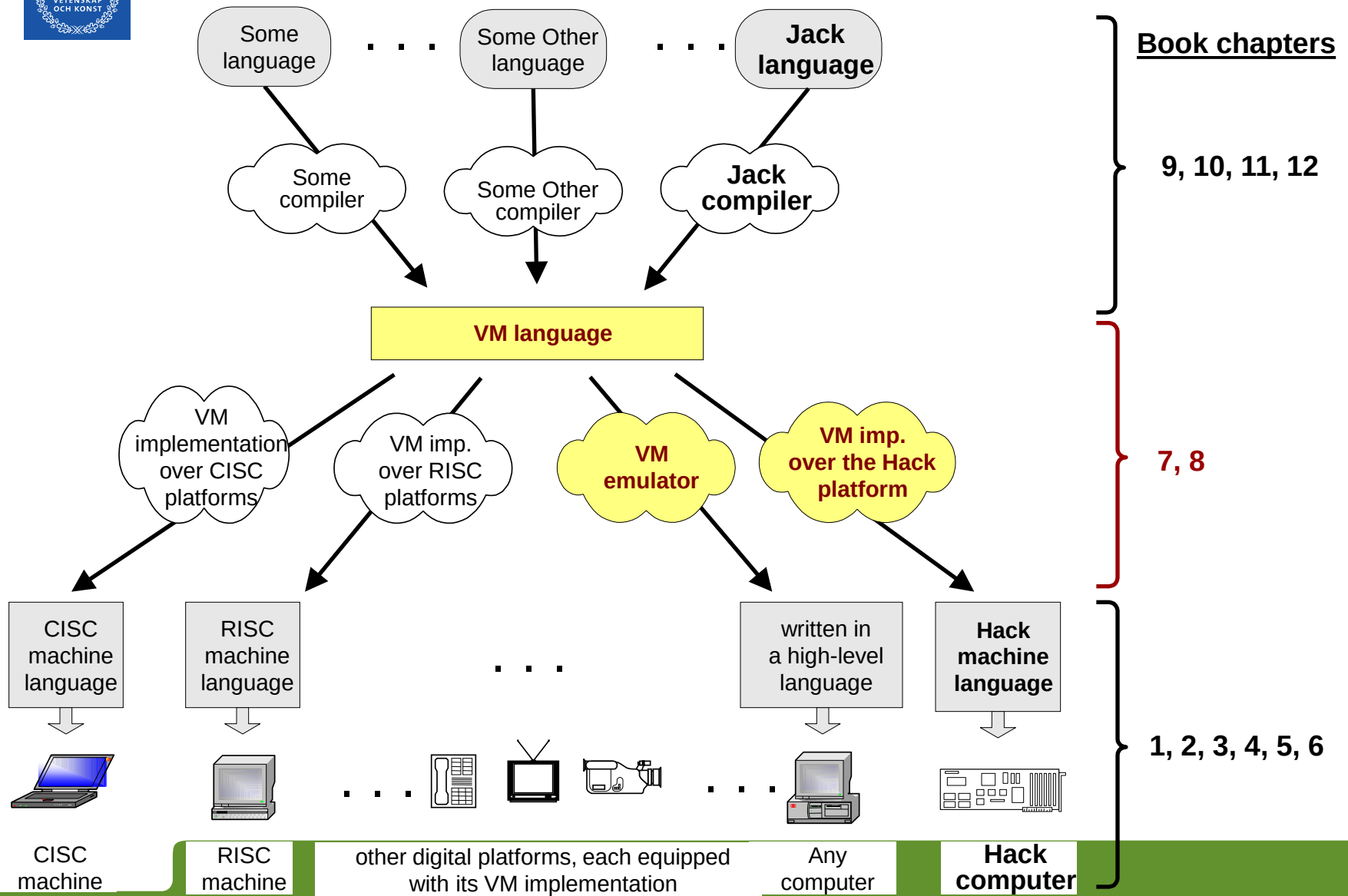


requires $n + m$ translators

Two-tier compilation:

- First compilation stage: depends only on the details of the source language
- Second compilation stage: depends only on the details of the target language.

Our focus (yellow):



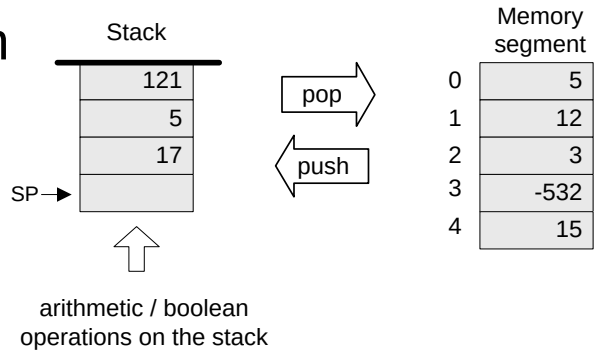
Models of Computation

Register machine

- Operations done on register contents

Stack machine

- Operations done on a stack (`add`, `sub`, `and`, …)
  - Last In First Out (LIFO) operation
- Data saved in various separate *memory segments* (*`static`, `constant`, `local`,…*)
  - Memory segments have similar behavior
- Perfect fit for postfix notation



Writing expressions: Infix, Prefix, Postfix notation

Infix notation

- Operator between operands
  - Example: A * (B + C) / D
- Evaluation
  - Operator precedence and associativity
  - Brackets to override precedence

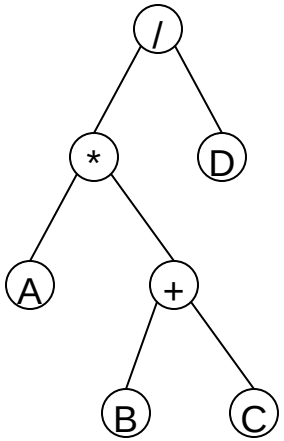
Prefix notation

- Operator before operands
  - Example: /*A+BCD
- Precedence cannot be overridden
- Evaluation left to right

Postfix notation

- Operator after operands
  - Example: ABC+*D/
- Precedence cannot be overriden
- Evaluation left to right

- Parse tree



Example: Evaluation of arithmetic expression

Infix:

z=(2-x)-(y+5)

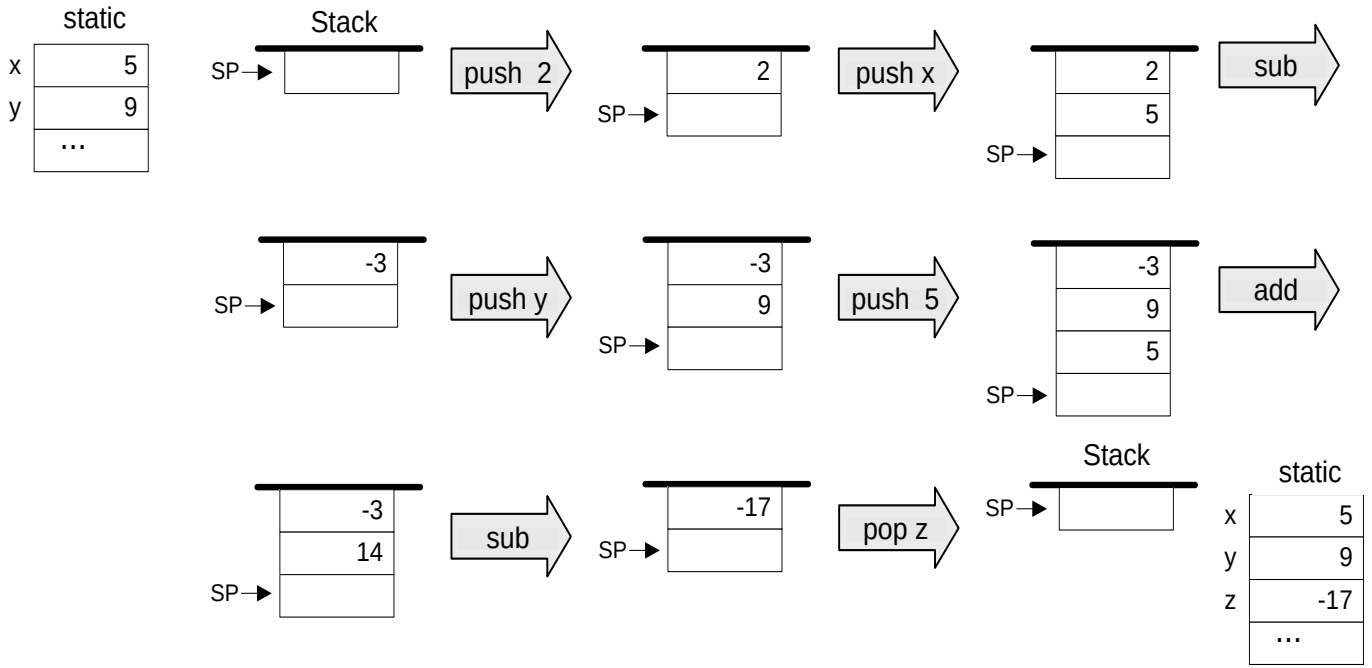
Postfix:

z=2,x,sub,y,5,add,sub

VM code (example)

```
// z=(2-x)-(y+5)
push constant 2
push static 0
sub
push static 1
push constant 5
add
sub
pop static 2
```

(assuming that
x refers to static 0,
y refers to static 1,
z refers to static 2)



VM at a Glance

Commands

Arithmetic / Boolean commands

`add`	`eq`	`and`
`sub`	`gt`	`or`
`neg`	`lt`	`not`

Memory access commands

`pop x` (pop into x, which is a variable)

`push y` (y being a variable or a constant)

Program flow commands

`label`	(declaration)
`goto`	(label)
`if-goto`	(label)

Function calling commands

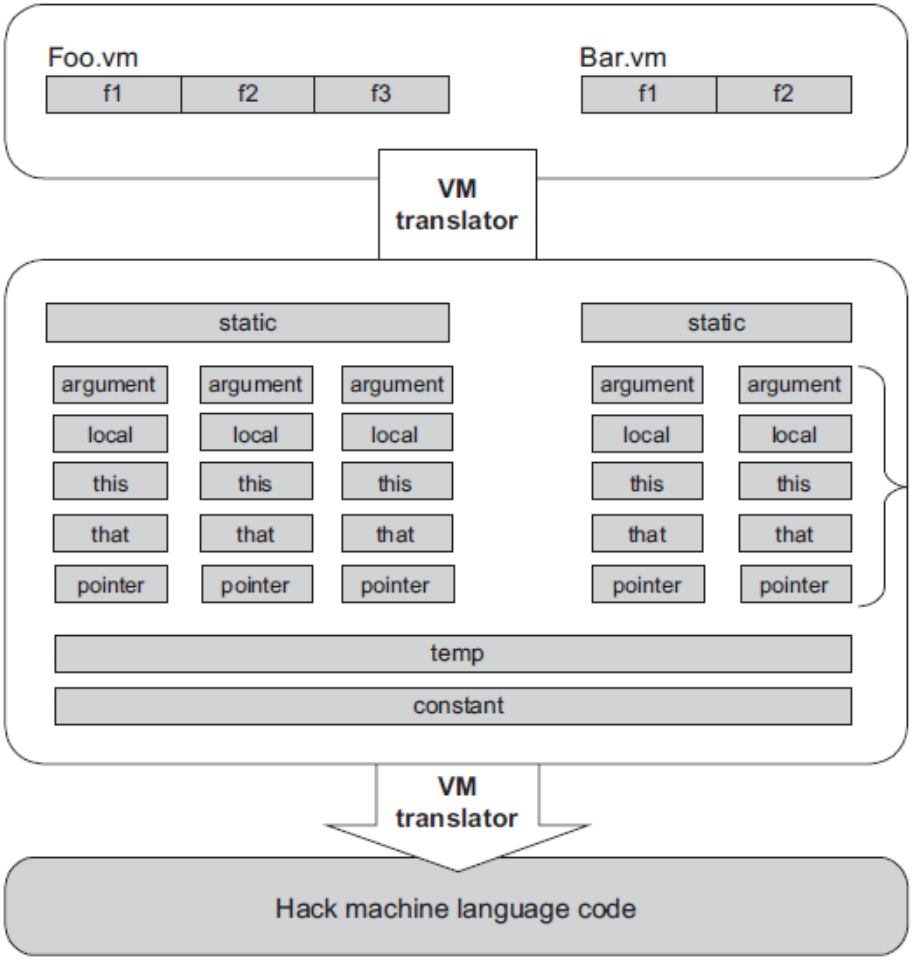
`function`	(declaration)
`call`	(a function)
`return`	(from a function)

Data types

- integer (-32768 to 32767)
- boolean (0 false, -1 true)
- pointer (memory address)

VM at a Glance

Program and memory structure



Segment	Purpose
`argument`	Stores the function's arguments.
`local`	Stores the function's local variables.
`static`	Stores static variables shared by all functions in the same `.vm` file.
`constant`	Pseudo-segment that holds all the constants in the range 0...32767.
`this` `that`	General-purpose segments. Can be made to correspond to different areas in the heap. Serve various programming needs.
`pointer`	A two-entry segment that holds the base addresses of the `this` and `that` segments.
`temp`	Fixed eight-entry segment that holds temporary variables for general use.

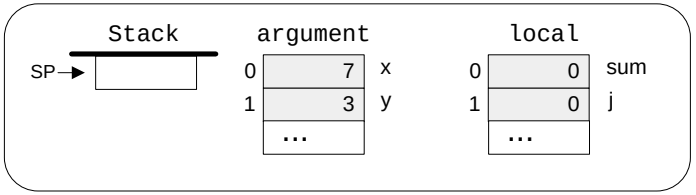
Register	*Name*
RAM[0]	SP
RAM[1]	LCL
RAM[2]	ARG
RAM[3]	THIS
RAM[4]	THAT

VM programming (example)

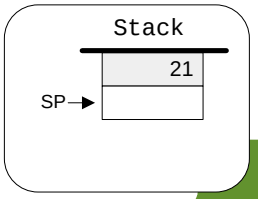
High-level code

```
function mult (x,y) {
  int sum, j;
  sum = 0;
  j = y;
  while ~(j = 0) {
    sum = sum + x;
    j = j - 1;
  }
  return sum;
}
```

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



VM code (first approx.)

```
function mult(x,y)
   push 0
   pop sum
   push y
   pop j
label loop
   push j
   push 0
   eq
   if-goto end
   push sum
   push x
   add
   pop sum
   push j
   push 1
   sub
   pop j
   goto loop
label end
   push sum
   return
```

VM code

```
function mult 2
  push   constant 0
  pop    local 0
  push   argument 1
  pop    local 1
label    loop
  push   local 1
  push   constant 0
  eq
  if-goto end
  push   local 0
  push   argument 0
  add
  pop    local 0
  push   local 1
  push   constant 1
  sub
  pop    local 1
  goto   loop
label    end
  push   local 0
  return
```

VM to Assembly Compilation - Example

Compilation done 1 VM code line at a time

- Parse VM code → Compile to assembly → Write ASM code
  - Usually many Assembly instructions
- Create globally unique labels from locally unique labels

VM relies on stack → stack pointer (SP) in assembly

- SP is predefined variable (at RAM 0)
- M[SP] : address of next empty position on stack

VM code

```
function mult 2
  push   constant 0
  pop    local 0
  push   argument 1
  pop    local 1
label    loop
  push   local 1
  push   constant 0
  eq
  if-goto end
  push   local 0
  push   argument 0
  add
  pop    local 0
  push   local 1
  push   constant 1
  sub
  pop    local 1
  goto   loop
label    end
  push   local 0
  return
```

VM to Assembly Compilation - Example

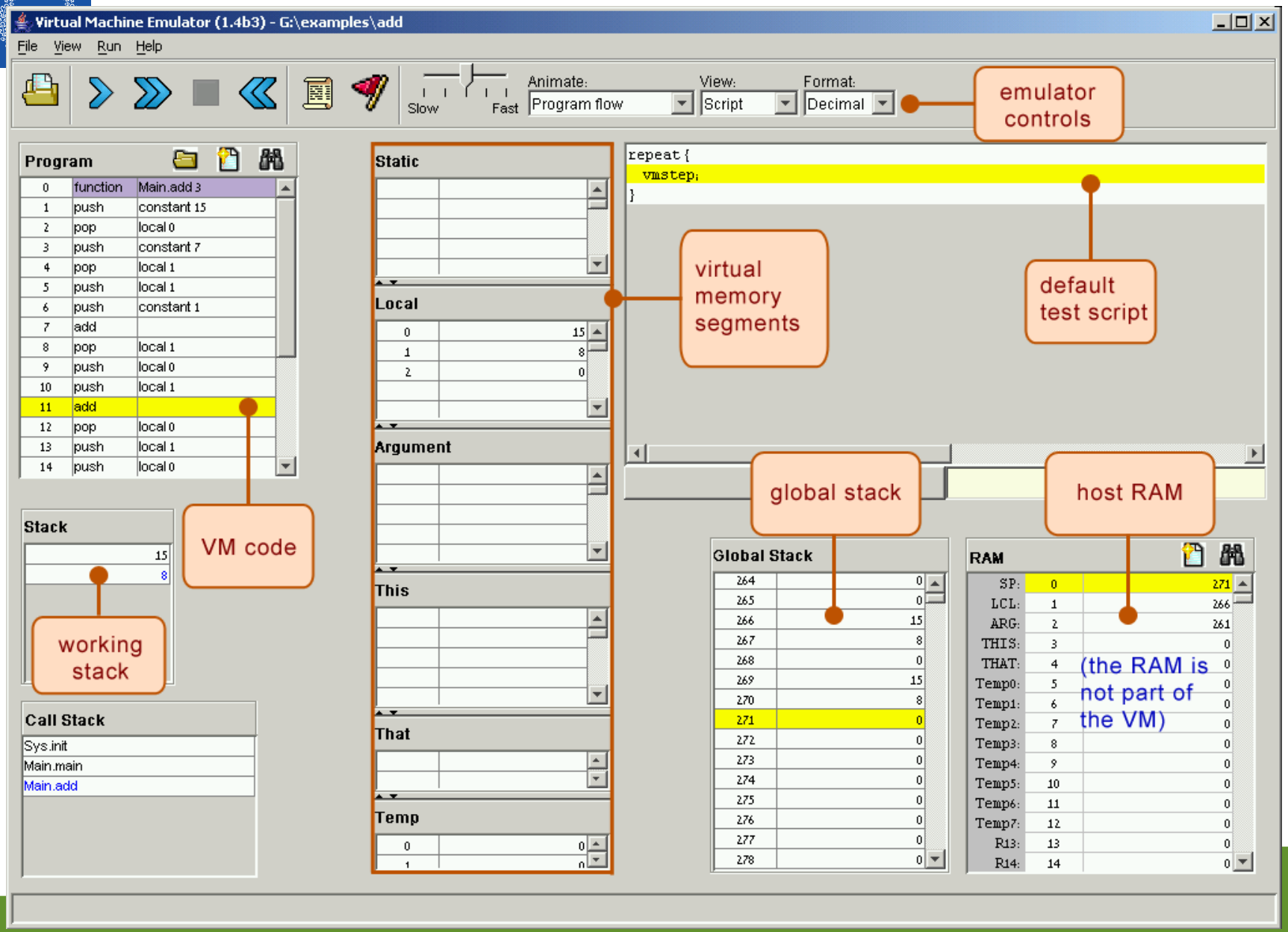
VM code

```
pop local 1
    \\ stack -> local 1
```

Assembly code

```
@LCL      // find the local segment address in the memory
D=M
D=D+1     // address of local 1 in D
@R13
M=D       // address stored in virtual register R13
@SP       // find address of stack pointer  (first empty space)
AM=M-1    // decrease with one, store address in A
          // (address of the field we need, and to location of SP after pop
D=M       // store value from top of stack, M[A] in D
M=0       // clean the now empty stack position
@R13      //
A=M       // get address of local 1
M=D       // write value removed from stack to local 1
```

Software implementation: Our VM emulator



Project

Read Chapter 7 of the book

Do project 7 from the course web page

- Implement parts of a VM to assembly compiler by extending the provided code
- You are provided C++ and Python skeleton files:
  - Choose the one you prefer

•Submit your solution by 8 am on 21 April 2017

We recommend that you rely on these files for the implementation of the project.