We acknowledge and pay our respects to the Kaurna people,
the traditional custodians whose ancestral lands we gather on.

We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

THE UNIVERSITY of ADELAIDE

Computer Systems

Lecture 06: Virtual Machine and The Stack

Review: The whole system

- **Programming Language** — Chapters 10-11
  - Compiler
- **Virtual Machine** — Chapters 7-8
  - Translator
- **Assembly Language** — Chapters 6
  - Assembler
- **Machine Language** — Chapters 4-5
  - System Architecture
- **Hardware Platform** — Chapters 1-3
  - Gate Logic
- **Chips and Logic Gates**
  - Electronics

Review: Assembling A-instructions

Symbolic: `@value` // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.

value (v = 0 or 1)

Binary:

0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v

<u>Translation to binary:</u>

- If *value* is a non-negative decimal number, simple
- If *value* is a symbol…

THE UNIVERSITY *of* ADELAIDE

Review: Assembling C-instructions

Symbolic: *dest*=*comp*;*jump* // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is ommitted;
// If *jump* is empty, the ";" is omitted.

	comp								dest			jump			
Binary: 1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

(when a=0) *comp*	c1	c2	c3	c4	c5	c6	(when a=1) *comp*
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

j1 ($out < 0$)	j2 ($out = 0$)	j3 ($out > 0$)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If $out > 0$ jump
0	1	0	JEQ	If $out = 0$ jump
0	1	1	JGE	If $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

THE UNIVERSITY of ADELAIDE

Review: Handling Symbols

(Also called *symbol resolution*)

Assembly programs typically have many symbols:

- Labels that mark destinations of jump commands
- Labels that mark special memory locations
- Variables

In Hack assembler there are three categories:

- Pre-defined symbols (used by the Hack platform)
- Labels (User–defined symbols)
- Variables (User-defined symbols)

Review: How do we build a symbol table?

Initialisation

- Create an empty table and put any pre-defined symbols in there.

First Pass

- Go through the source code and add all the user-defined labels to the table.
- The label's value is the location of the first instruction after the label.

Second Pass

- Go through the source code and use the symbol table to translate the commands. This is where names get turned into actual numbers.

THE UNIVERSITY of ADELAIDE

Review: Example

What does the symbol table look like for this program?

When this program has finished assembling, what is the resulting machine code?

```
// Computes 1+...+RAM[0]
// And stored the sum in RAM[1]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0
(LOOP)
    @i    // if i>RAM[0] goto WRITE
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

The Virtual Machine

THE UNIVERSITY of ADELAIDE

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.

Compiler

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
   ...
```

THE UNIVERSITY of ADELAIDE

Motivation

Jack code

```
function void vfunc()
{
    return ;
}
```

Compiler

Hack VM code

```
function voidfn.vfunc 0
push constant 0
return
```

VM Translator

Assembler

```
(voidfn.vfunc)
@SP
AM=M+1
A=A-1
M=0
(_4)
@LCL
D=M
@5
A=D-A
D=M
@R14
M=D
@SP
AM=M-1
D=M
@ARG
A=M
M=D
D=A+1
@R15
M=D
@LCL
D=M
@SP
M=D
@SP
AM=M-1
D=M
@THAT
M=D
@SP
AM=M-1
D=M
@THIS
M=D
@SP
AM=M-1
D=M
@ARG
M=D
@SP
AM=M-1
D=M
@LCL
M=D
@R15
D=M
@SP
M=D
@R14
A=M
0;JMP
```

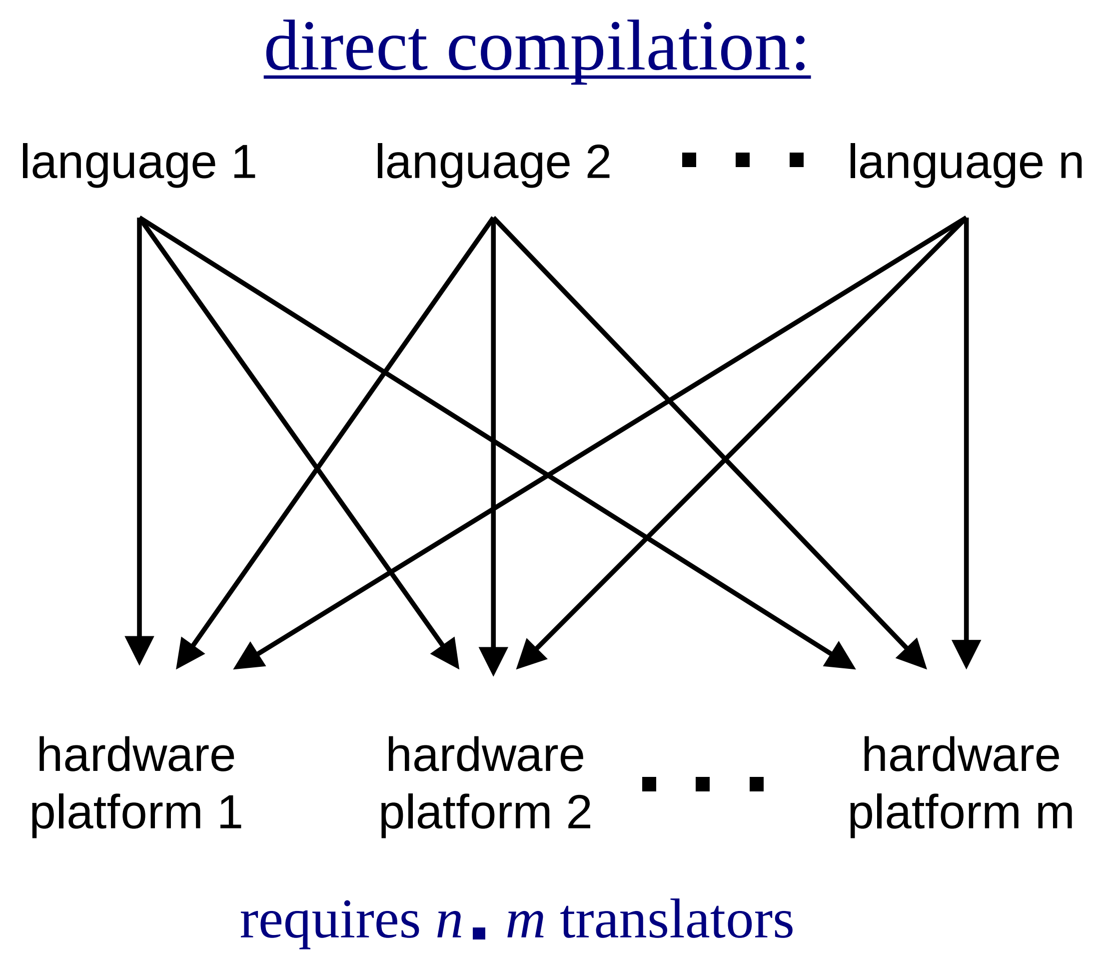
Assembler

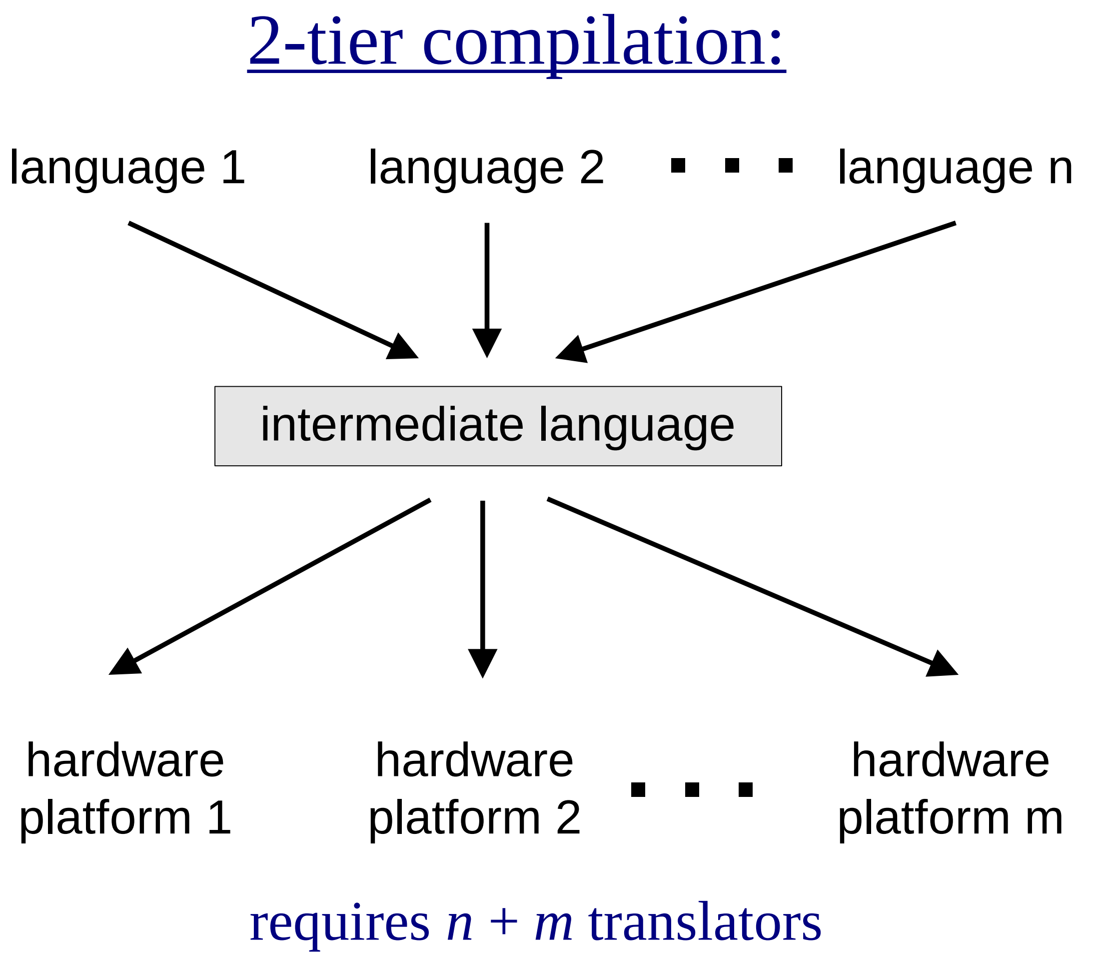
Machine Code

```
0000000000000000
1111110111101000
1110110010100000
1110101010001000
0000000000000001
1111110000010000
0000000000000101
1110010011100000
1111110000010000
0000000000001110
1110001100001000
0000000000000000
1111110010101000
1111110000010000
0000000000000010
1111110000100000
1110001100001000
1110110111010000
0000000000001111
1110001100001000
0000000000000001
1111110000010000
0000000000000000
1110001100001000
0000000000000000
1111110010101000
1111110000010000
0000000000000100
1110001100001000
0000000000000000
1111110010101000
1111110000010000
0000000000000011
1110001100001000
0000000000000000
1111110010101000
1111110000010000
0000000000000010
1110001100001000
0000000000000000
1111110010101000
1111110000010000
0000000000000001
1110001100001000
0000000000001111
1111110000010000
0000000000000000
1110001100001000
0000000000001110
1111110000100000
1110101010000111
```

THE UNIVERSITY of ADELAIDE

Compilation models

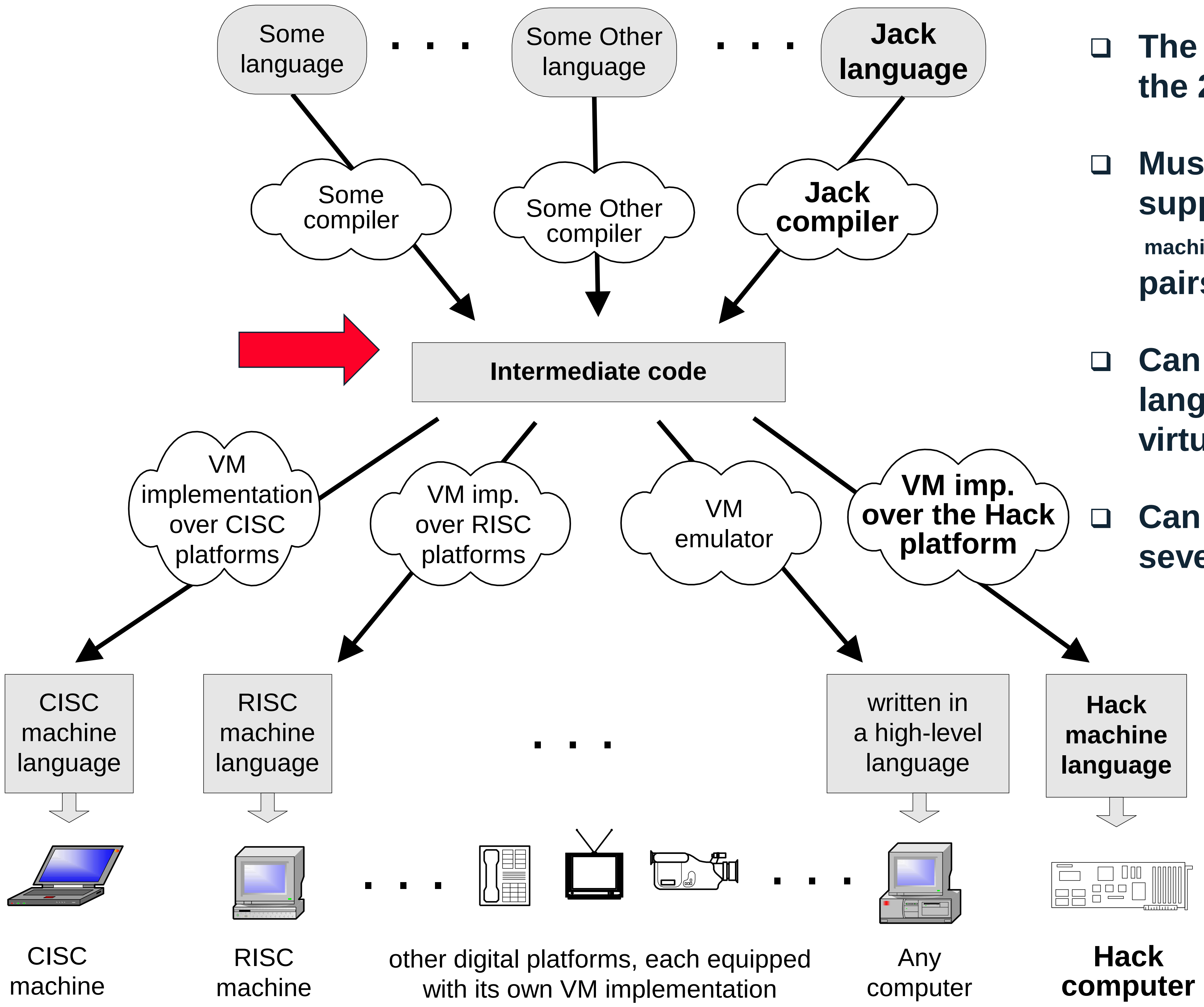




Two-tier compilation:

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

The big picture



The intermediate code:

- The interface between the 2 compilation stages
- Must be sufficiently general to support many <high-level language, machine-language> pairs
- Can be modeled as the language of an abstract virtual machine (VM)
- Can be implemented in several different ways.

The VM model and language

Perspective:

From here till the end of the next lecture we describe the VM model used in the Hack-Jack platform

Other VM models (like Java's JVM/JRE and .NET's IL/CLR) are similar in spirit but differ in scope and details.

Several different ways to think about the notion of a virtual machine:

- Abstract software engineering view:
  the VM is an interesting abstraction that makes sense in its own right
- Practical software engineering view:
  the VM code layer enables "managed code" (e.g. enhanced security)
- Pragmatic compiler writing view:
  a VM architecture makes writing a compiler much easier
  (as we'll see later in the course)
- Opportunistic empire builder view:
  a VM architecture allows writing high-level code once and have it run on many target platforms with little or no modification.

Lecture plan

<u>Goal:</u> Specify and implement a VM model and language:

<u>Arithmetic / Boolean commands</u>

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

This week

<u>Memory access commands</u>

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

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

<u>Program flow commands</u>

`label` (declaration)

`goto` (label)

`if-goto` (label)

Next week

<u>Function calling commands</u>

`function` (declaration)

`call` (a function)

`return` (from a function)

<u>Our game plan:</u> (a) describe the VM abstraction (above)
(b) propose how to implement it over the Hack platform.

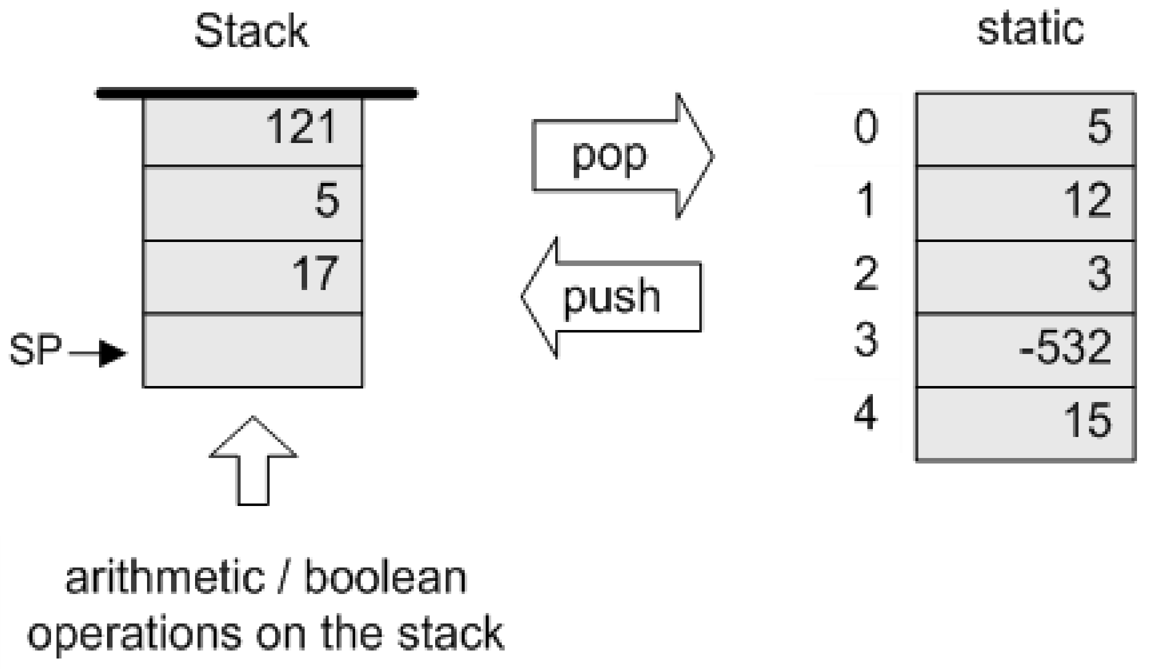
Our VM model is *stack-oriented*

All operations are done on a stack

Data is saved in several separate *memory segments*

All the memory segments behave the same

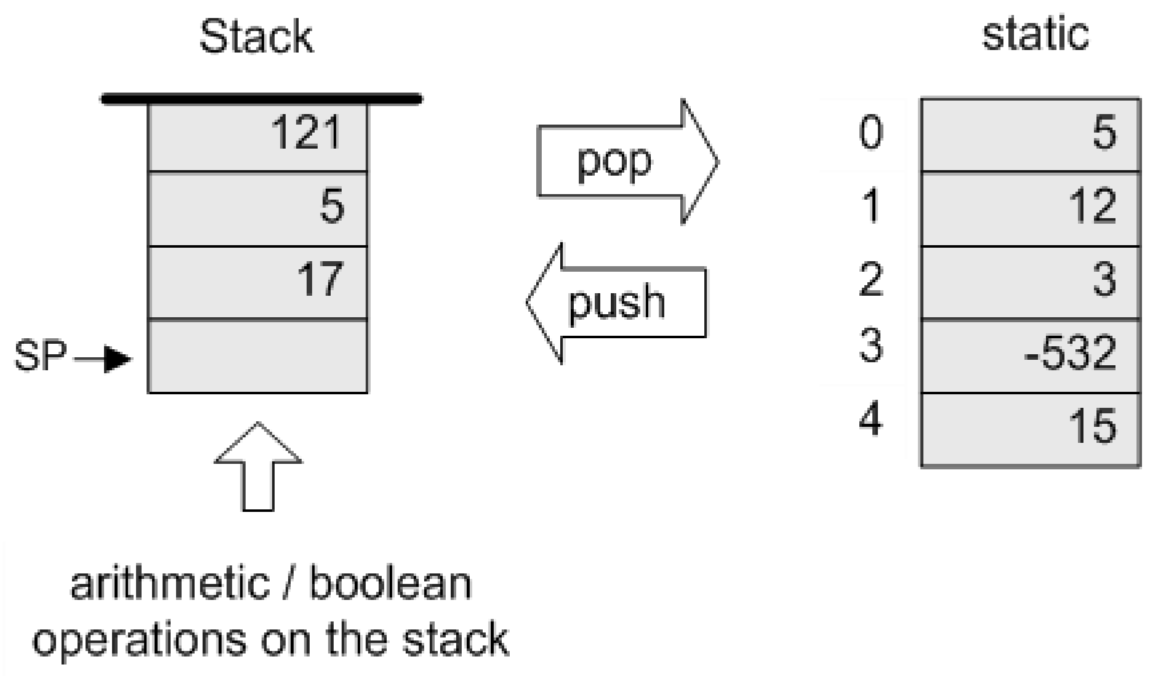
One of the memory segments m is called `static`, and we will use it (as an arbitrary example) in the following examples:



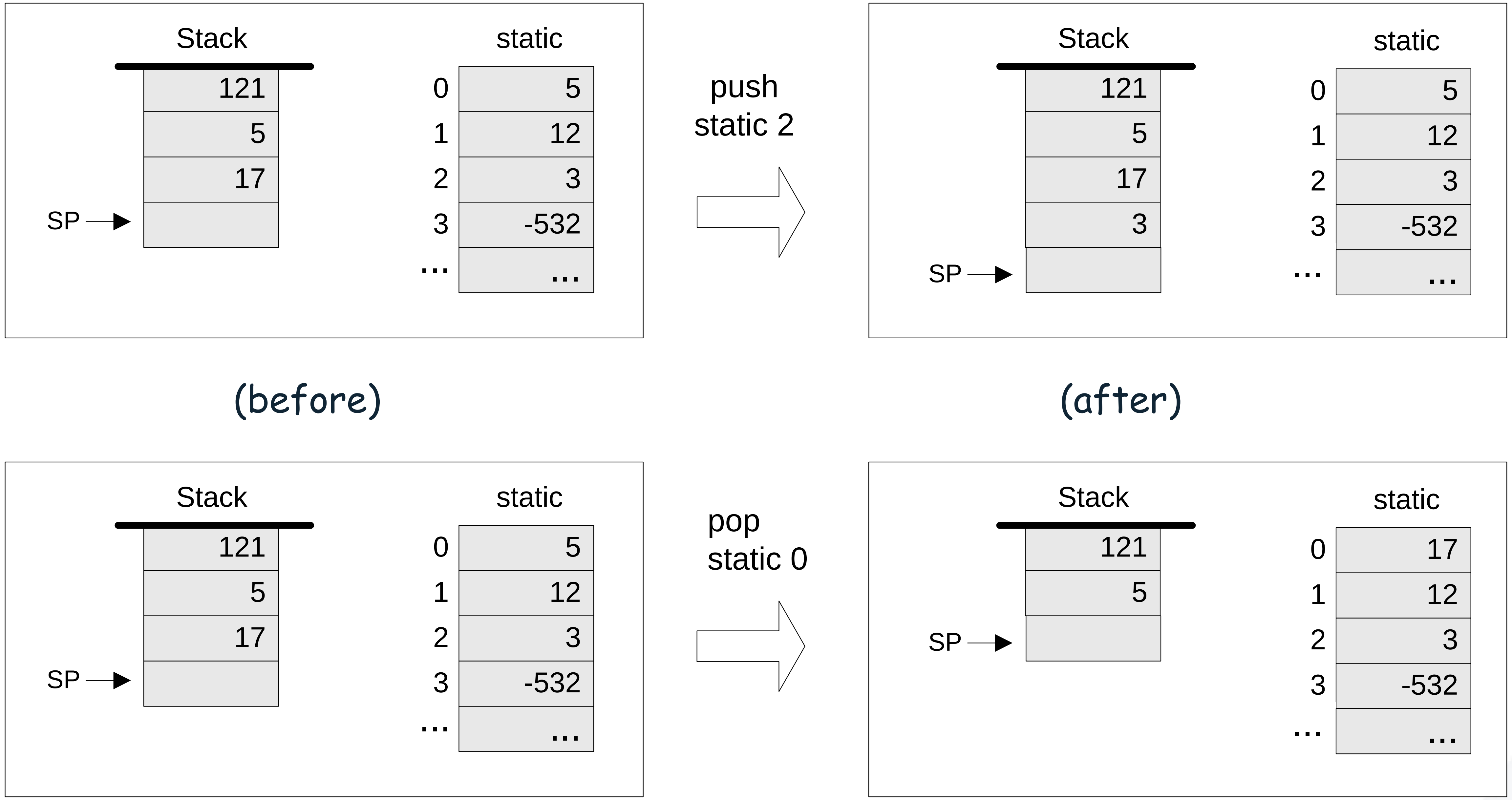
Data types

Our VM model features a single 16-bit data type that can be used as:

- an integer value (16-bit 2's complement: -32768, ... , 32767)
- a Boolean value (-1 and 0, standing for true and false)
- a pointer (memory address)



Memory access operations



The stack:

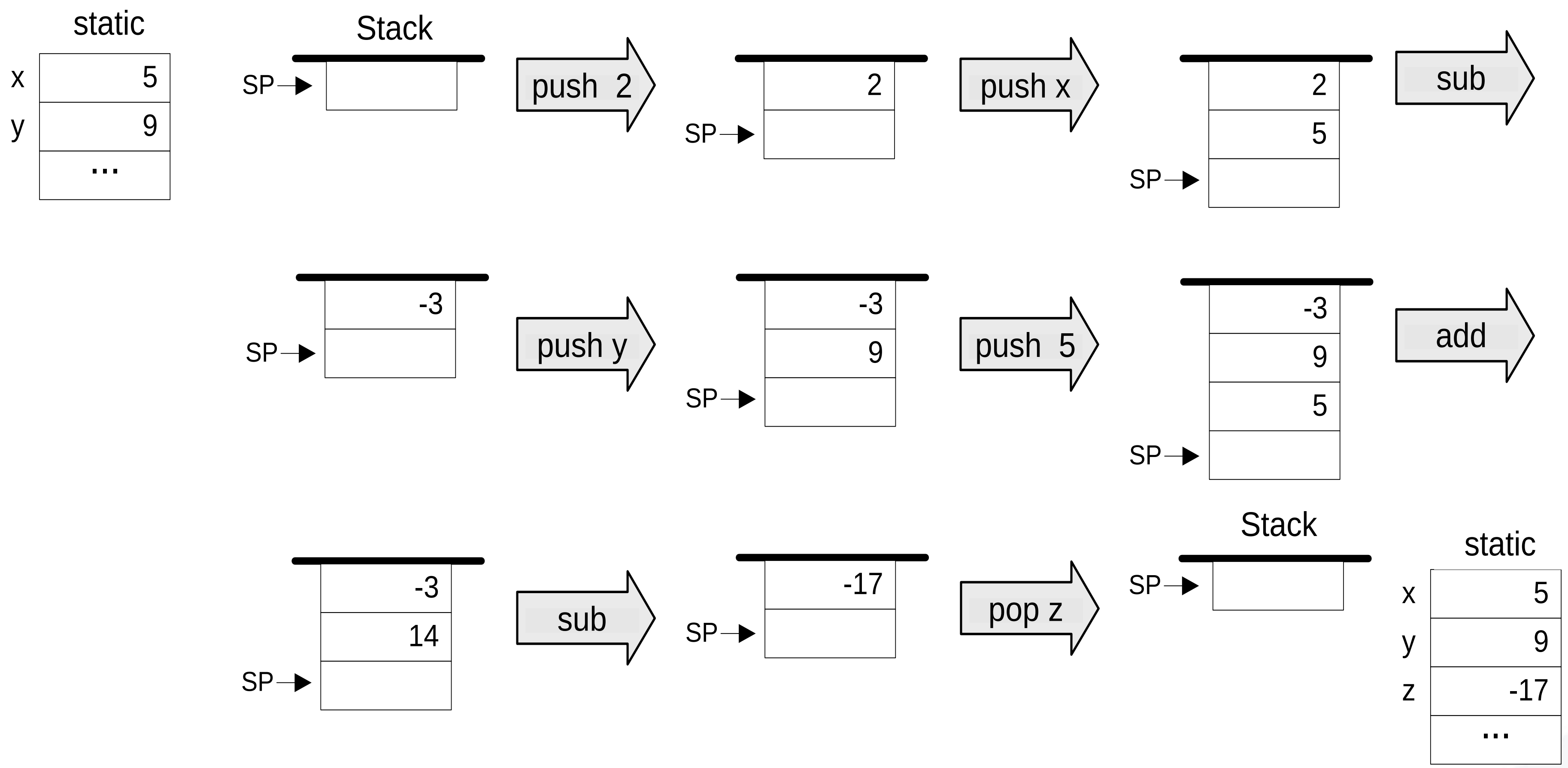
- A classical LIFO data structure
- Elegant and powerful
- Several hardware / software implementation options.

Evaluation of arithmetic expressions

VM code (example)

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

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

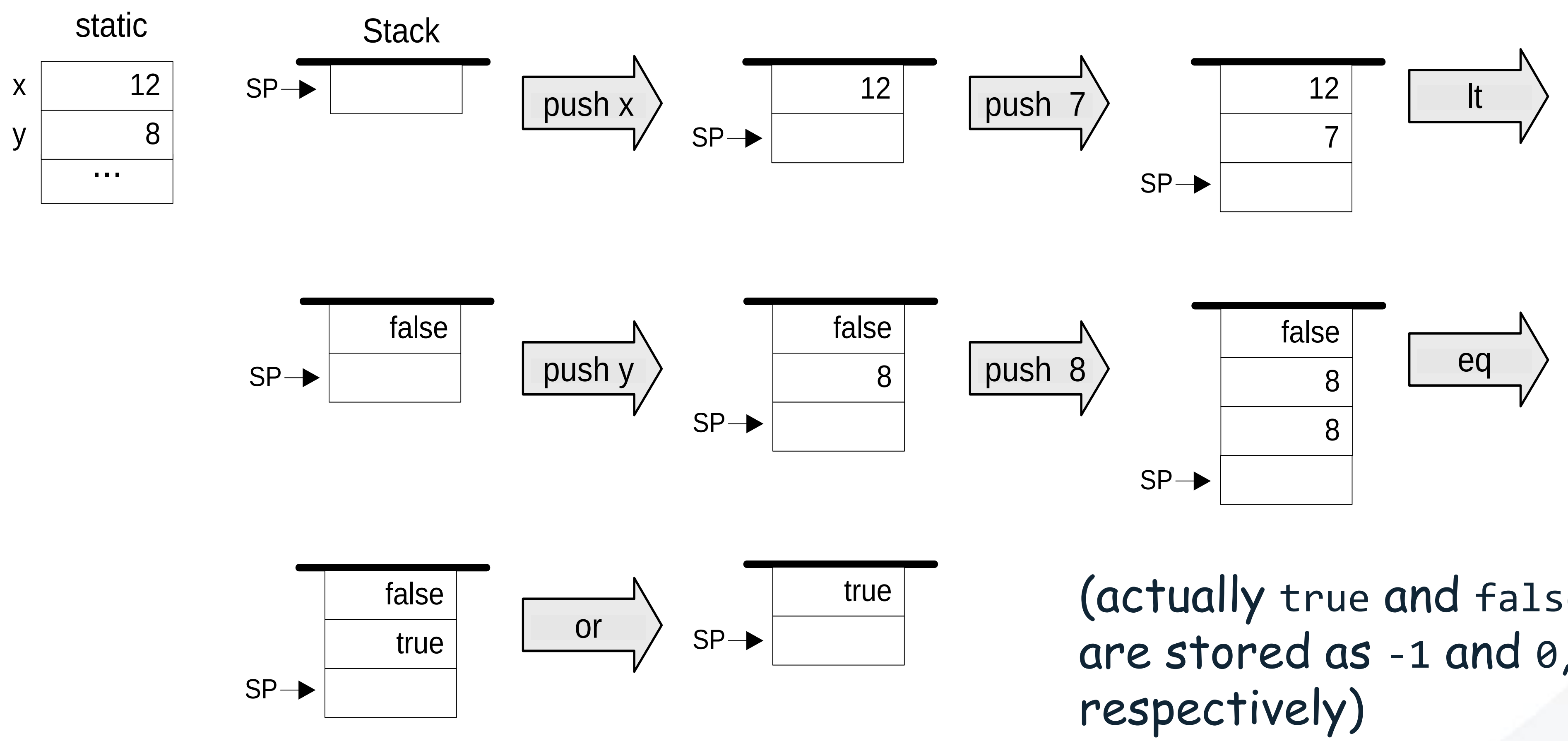


Evaluation of Boolean expressions

VM code (example)

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

(suppose that
x refers to static 0, and
y refers to static 1)

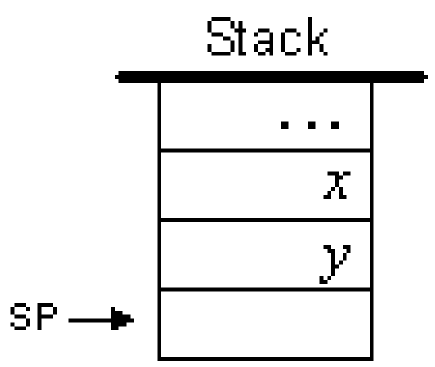


(actually true and false are stored as -1 and 0, respectively)

Arithmetic and Boolean commands in the VM language

(wrap-up)

Command	Return value (after popping the operand/s)	Comment	
`add`	$x+y$	Integer addition	(2's complement)
`sub`	$x-y$	Integer subtraction	(2's complement)
`neg`	$-y$	Arithmetic negation	(2's complement)
`eq`	true if $x=y$ and false otherwise	Equality	
`gt`	true if $x>y$ and false otherwise	Greater than	
`lt`	true if $x<y$ and false otherwise	Less than	
`and`	x And y	Bit-wise	
`or`	x Or y	Bit-wise	
`not`	Not y	Bit-wise	



Expressions to Hack VM Language

`not (a or b)`

```
push static 0
push static 1
or
not
```

`d + c + b + a`

```
push static 3
push static 2
add
push static 1
add
push static 0
add
```

Expressions to Hack VM Language

(4 + a) * (c - 9)

```
push constant 4
push static 0
add
push static 1
push constant 9
sub
call Math.multiply 2
```

true and false

```
push constant 0
not
push constant 0
and
```

The VM's Memory segments

A VM program is designed to provide an interim abstraction of a program written in some high-level language

Modern OO high-level languages normally feature the following variable kinds:

Class level:

- Static variables (class-level variables)
- Private variables (aka "object variables" / "fields" / "properties")

Method level:

- Local variables
- Argument variables

When translated into the VM language,

The static, private, local and argument variables are mapped by the compiler on the four memory segments
`static`, `this`, `local`, `argument`

In addition, there are four additional memory segments, whose role will be presented later:
`that`, `constant`, `pointer`, `temp`.

THE UNIVERSITY of ADELAIDE

Memory segments and access commands

The VM abstraction includes 8 separate memory segments named:
static, this, local, argument, that, constant, pointer, temp

As far as VM programming commands go, all memory segments look and behave the same

To access a particular segment entry, use the following generic syntax:

Memory access VM commands:

- pop *memorySegment* *index*
- push *memorySegment* *index*

Where *memorySegment* is `static`, `this`, `local`, `argument`, `that`, `constant`, `pointer`, or `temp`

And *index* is a non-negative integer

Notes:

(In all our code examples thus far, *memorySegment* was `static`)

The roles of the eight memory segments will become relevant when we talk about compiling

At the VM abstraction level, all memory segments are treated the same way.

THE UNIVERSITY *of* ADELAIDE

VM programming

VM programs are normally written by *compilers*, not by humans

However, compilers are written by humans ...

In order to write or optimize a compiler, it helps to first understand the spirit of the compiler's target language – the VM language

The example VM program includes four new VM commands:

- `function` *functionSymbol int* // function declaration
- `label` *labelSymbol* // label declaration
- `goto` *labelSymbol* // jump to execute the command after *labelSymbol*
- `if-goto` *labelSymbol* // `pop x`
  // if `x=true`, jump to execute the command after *labelSymbol*
  // else proceed to execute the next command in the program

For example, to effect `if (x > n) goto loop`, we can use the following VM commands:

```
push x
push n
gt
if-goto loop          // Note that x, n, and the truth value were removed from the stack.
```

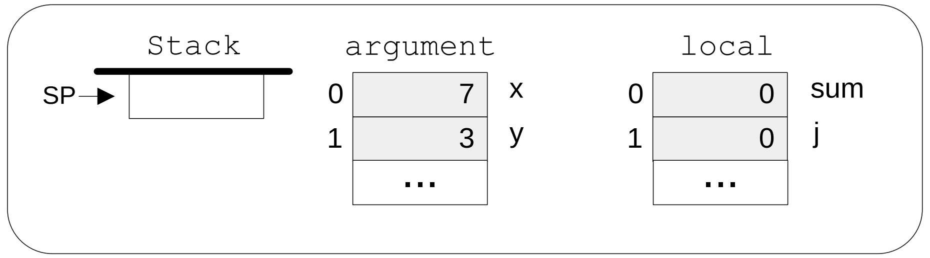
THE UNIVERSITY of ADELAIDE

VM programming (example)

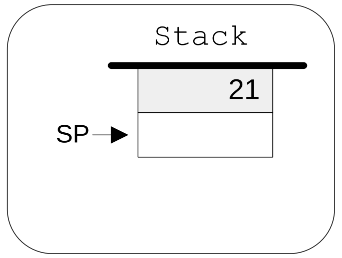
High-level code

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

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



VM code (first approx.)

```
function mult(x,y)
   push 0
   pop result
   push y
   pop j
label loop
   push j
   push 0
   eq
   if-goto end
   push result
   push x
   add
   pop result
   push j
   push 1
   sub
   pop j
   goto loop
label end
   push result
   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 Programming Examples in Class XX

```
function int add(int x,int y) { return x + y ; }
```

```
function XX.add 0
push argument 0
push argument 1
add
return
```

```
function void hello(String hi) { do Unix.print(hi) ; }
```

```
function XX.hello 0
push argument 0
call Unix.print 1
push constant 0
return
```

VM programming: multiple functions

Compilation:

- A Jack application is a set of 1 or more class files (just like .java files).
- When we apply the Jack compiler to these files, the compiler creates a set of 1 or more .vm files (just like .class files). Each method in the Jack app is translated into a VM function written in the VM language
- Thus, a VM file consists of one or more VM functions.

Execution:

- At any given point of time, only one VM function is executing (the "current function"), while 0 or more functions are waiting for it to terminate (the functions up the "calling hierarchy")
- For example, a main function starts running; at some point we may reach the command call factorial, at which point the factorial function starts running;
  then we may reach the command call mult, at which point the mult function starts running, while both main and factorial are waiting for it to terminate

The stack: a global data structure, used to save and restore the resources (memory segments) of all the VM functions up the calling hierarchy (e.g. main and factorial). The tip of this stack if the working stack of the current function (e.g. mult).

VM Implementation

VM implementation options:

Software-based (eg emulate the VM model using Java)

Translator-based (eg translate VM programs into the Hack machine language)

Hardware-based (realize the VM model using dedicated memory and registers)

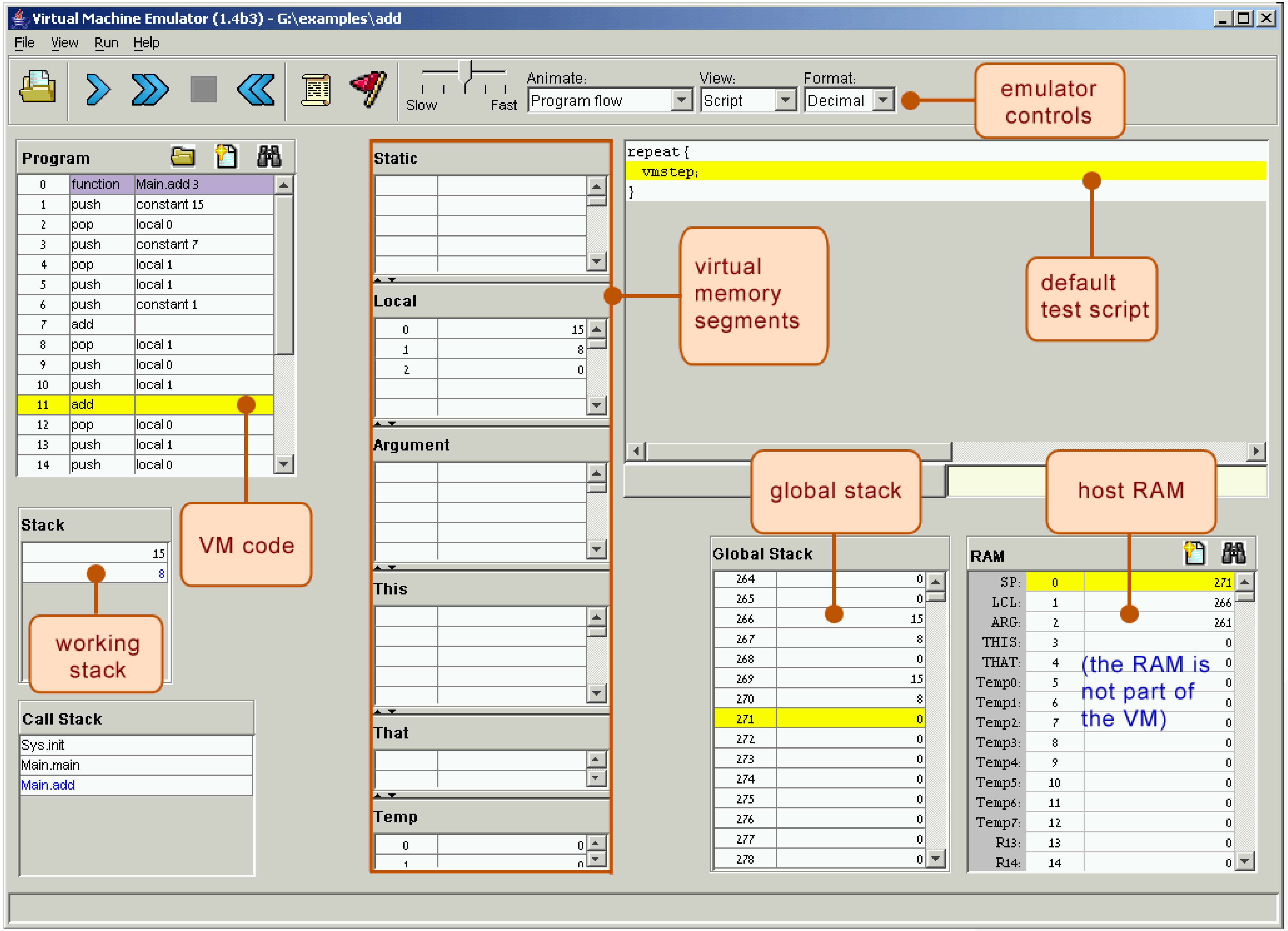
Two well-known translator-based implementations:

JVM: Javac translates Java programs into bytecode;
The JVM translates the bytecode into the machine language of the host computer

CLR: C# compiler translates C# programs into IL code;
The CLR translated the IL code into the machine language of the host computer.

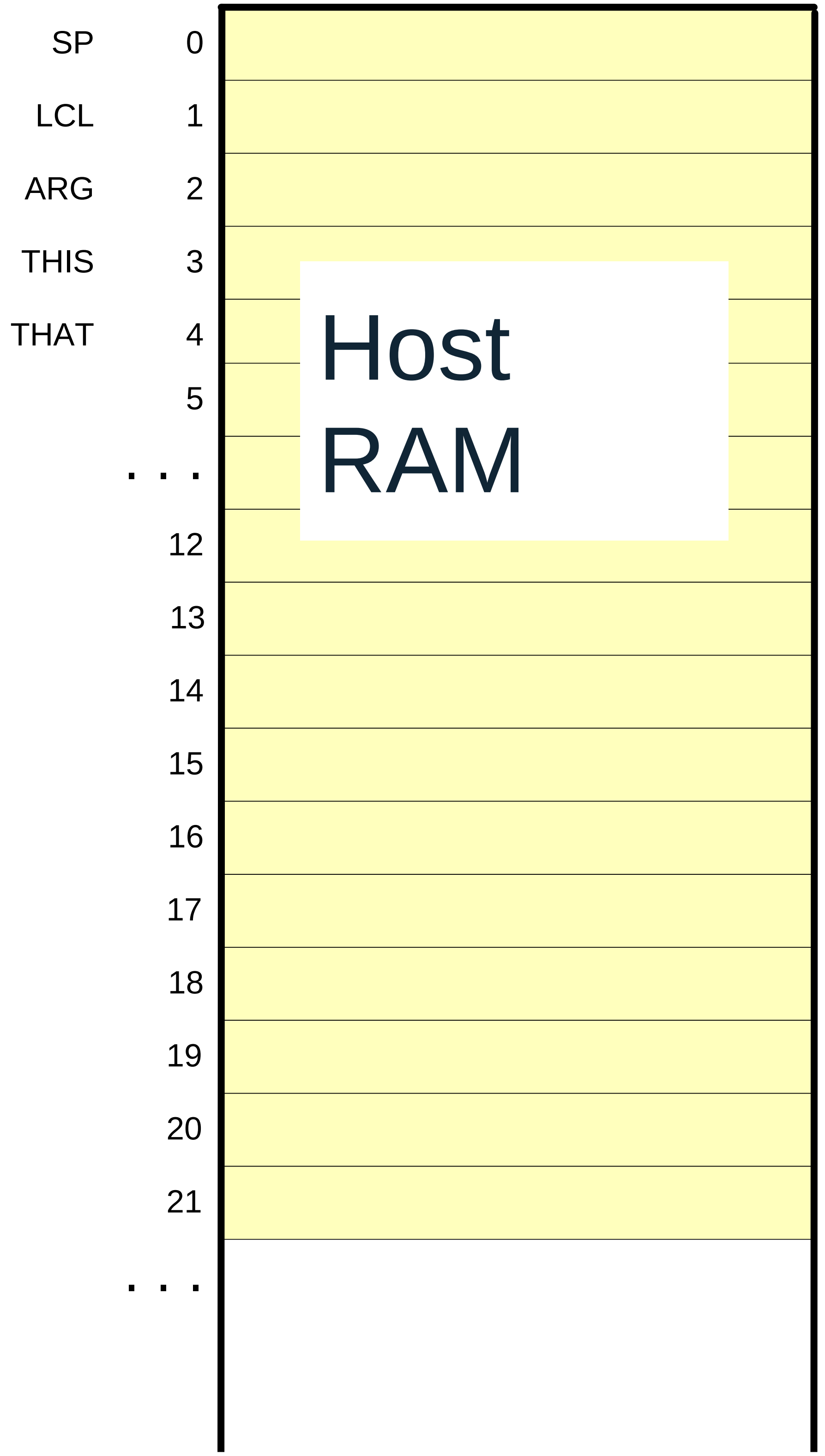
THE UNIVERSITY of ADELAIDE

Software implementation: Hack VM emulator



THE UNIVERSITY of ADELAIDE

VM implementation on the Hack platform



<u>The stack:</u> a global data structure, used to save and restore the resources of all the VM functions up the calling hierarchy.

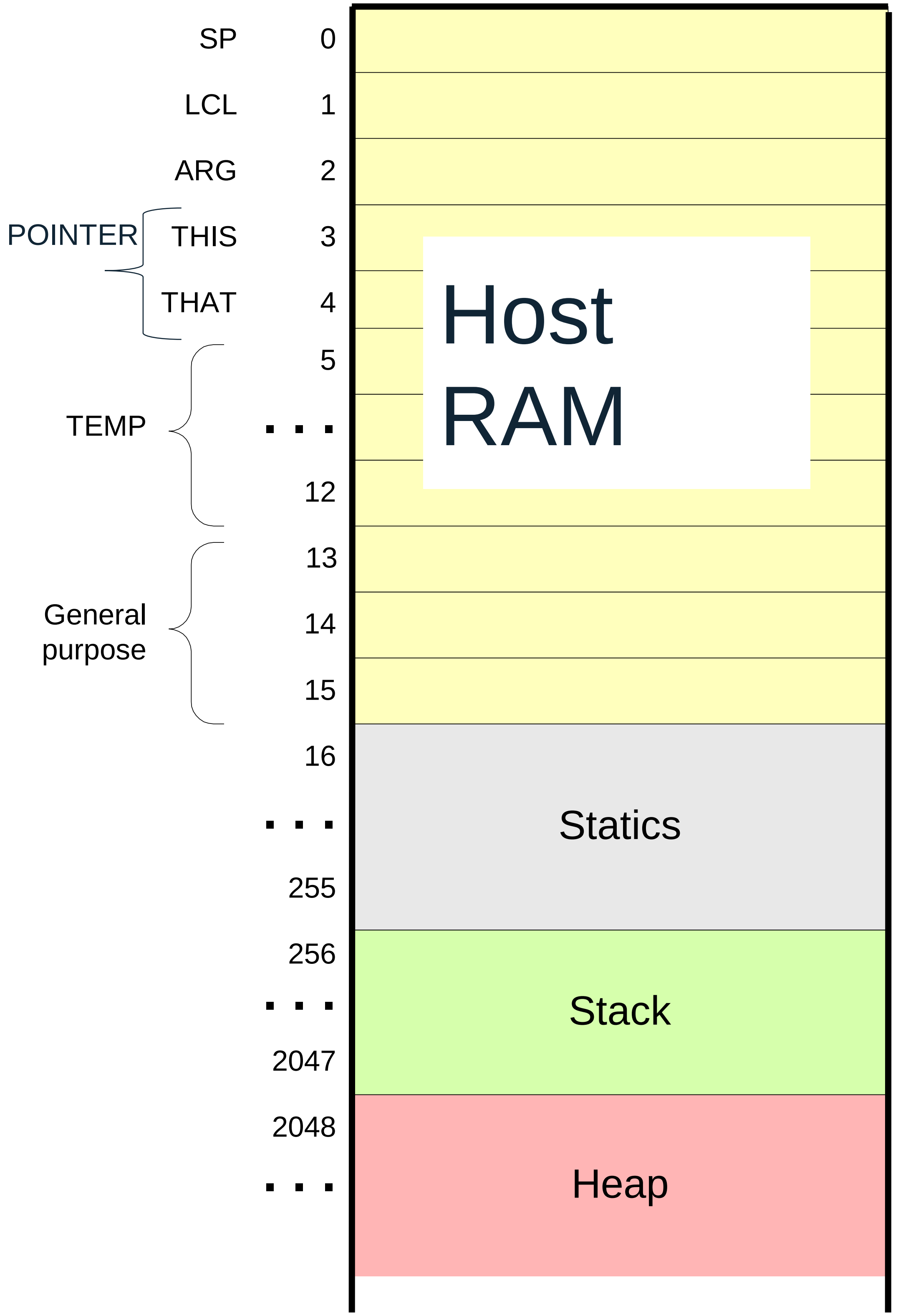
The tip of this stack is the working stack of the current function

<u>static, constant, temp, pointer</u>:
Global memory segments, all functions see the same four segments

<u>local,argument,this,that</u>:
these segments are local at the function level; each function sees its own, private copy of each one of these four segments

<u>The challenge:</u>
represent all these logical constructs on the same single physical address space -- the host RAM.

VM implementation on the Hack platform



Basic idea: the mapping of the stack and the global segments on the RAM is easy (fixed); the mapping of the function-level segments is dynamic, using pointers

The stack: mapped on RAM[256 ... 2047]; The stack pointer is kept in RAM address SP

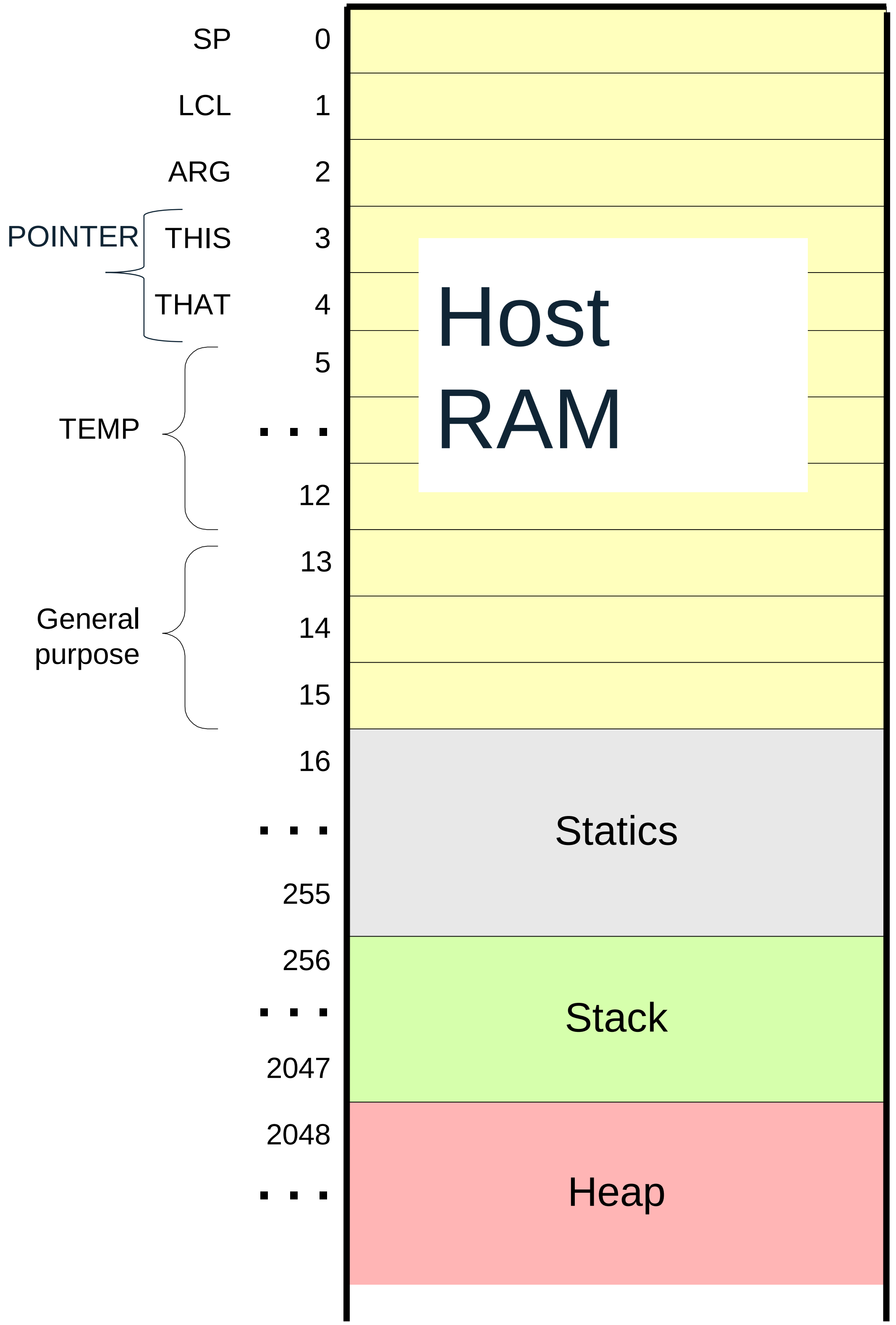
static: mapped on RAM[16 ... 255]; each segment reference static *i* appearing in a VM file named f is compiled to the assembly language symbol f.i (recall that the assembler further maps such symbols to the RAM, from address 16 onward)

local,argument,this,that: these method-level segments are mapped somewhere from address 256 onward, on the “stack” or the “heap”. The base addresses of these segments are kept in RAM addresses LCL, ARG, THIS, and THAT. Access to the *i*-th entry of any of these segments is implemented by accessing RAM[segmentBase + *i*]

constant: a truly virtual segment: access to constant *i* is implemented by supplying the constant *i.*

pointer: RAM[3..4] to change THIS and THAT.

VM implementation on the Hack platform



Practice exercises

Now that we know how the memory segments are mapped on the host RAM, we can write Hack commands that realize the various VM commands. for example, let us write the Hack code that implements the following VM commands:

- push constant 1
- pop static 7 (suppose it appears in a VM file named f)
- push constant 5
- add
- pop local 2
- eq

Tips:

1. The implementation of any one of these VM commands requires several Hack assembly commands involving pointer arithmetic (using commands like A=M)
2. If you run out of registers (you have only two ...), you may use R13, R14, and R15.

VM Translator Parsing

- Memory locations R13, R14, R15 can be used as temporary variables if required
- `push constant 1`

```
@SP
AM=M+1
A=A-1
M=1
```

- `pop static 7 (in a VM file named Bob.vm)`

```
@SP
AM=M-1
D=M
@Bob.7
M=D
```

VM Translator Parsing

- `push constant 5`

```
@5
D=A
@SP
AM=M+1
A=A-1
M=D
```

- `add`

```
@SP
AM=M-1
D=M
A=A-1
M=D+M
```

Perspective

In this lecture we began the process of building a compiler

Modern compiler architecture:

Front-end (translates from a high-level language to a VM language)

Back-end (translates from the VM language to the machine language of some target hardware platform)

Brief history of virtual machines:

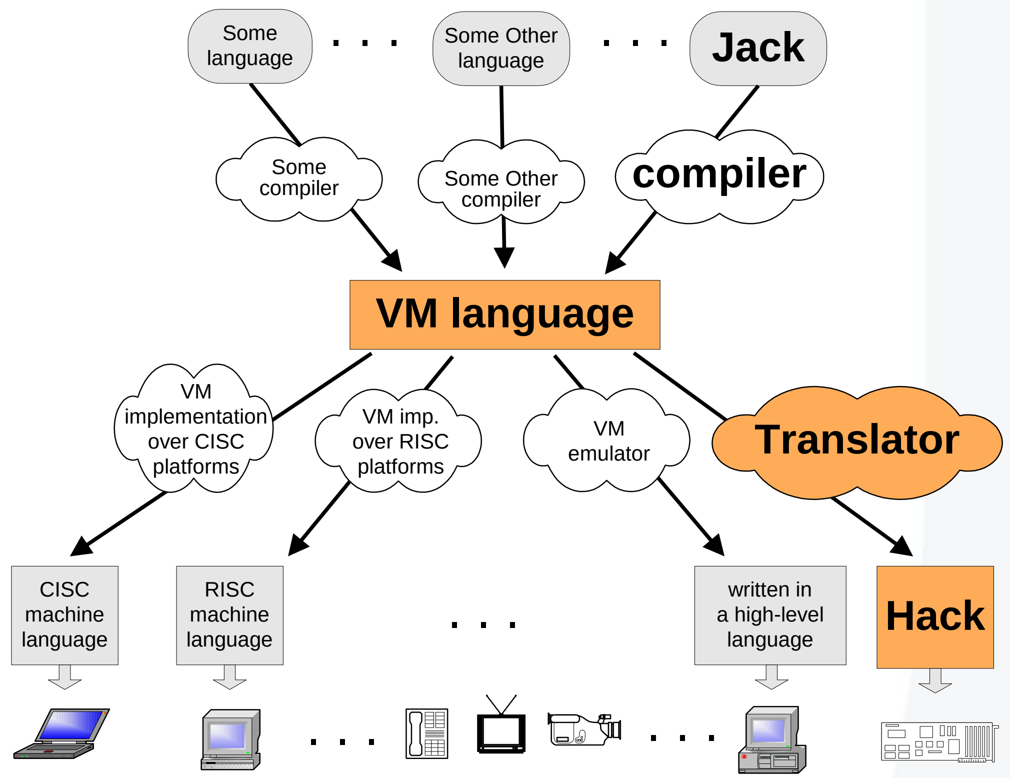
1970's: p-Code

1990's: Java's JVM

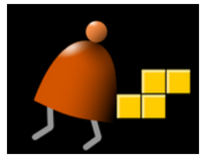
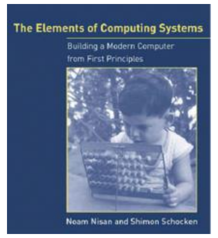
2000's: Microsoft .NET

A full blown VM implementation typically also includes a common software library (can be viewed as a mini, portable OS).

We will build such a mini OS later in the course.



The big picture

Java	Microsoft .net		
JVM	CLR	VM	7, 8
Java	C#	Jack	9
Java compiler	C# compiler	Jack compiler	10, 11
JRE	.NET base class library	Mini OS	12

(Book chapters and Course projects)

This Week

- Review Chapters 6 & 7 of the Text Book (if you haven’t already)
- Assignment 4 Due
- Assignment 5 Available Shortly – Due after Mid-Semester break.
- Review Chapter 8 of the Text Book before week 8.