



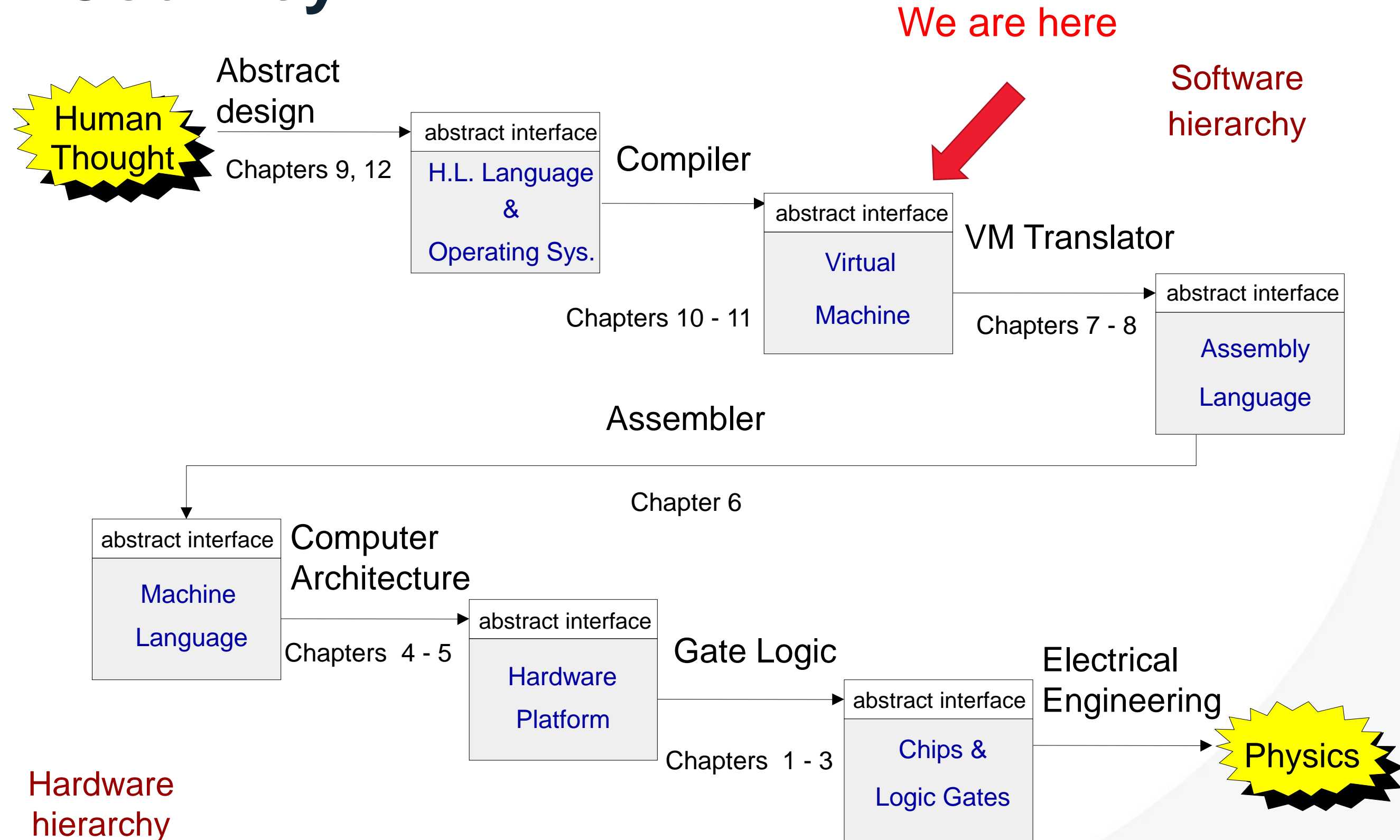
THE UNIVERSITY
of ADELAIDE

Computer Systems

Lecture 11: Virtual Machine and
The Stack



Our Journey

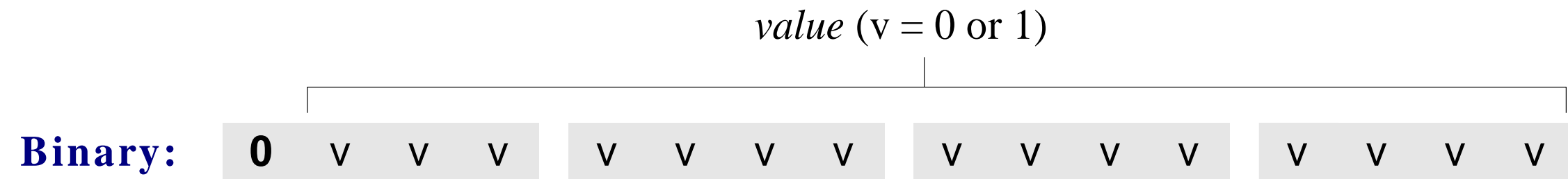


(Abstraction–implementation paradigm)



Review: Assembling A-instructions

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



Translation to binary:

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

Review: Assembling C-instructions

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

Binary:

<i>comp</i>				<i>dest</i>		<i>jump</i>									
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

(when a=0) <i>comp</i>	c1	c2	c3	c4	c5	c6	(when a=1) <i>comp</i>	d1	d2	d3	Mnemonic	Destination (where to store the computed value)
0	1	0	1	0	1	0		0	0	0	null	The value is not stored anywhere
1	1	1	1	1	1	1		0	0	1	M	Memory[A] (memory register addressed by A)
-1	1	1	1	0	1	0		0	1	0	D	D register
D	0	0	1	1	0	0		0	1	1	MD	Memory[A] and D register
A	1	1	0	0	0	0	M	1	0	0	A	A register
!D	0	0	1	1	0	1		1	0	1	AM	A register and Memory[A]
!A	1	1	0	0	0	1	!M	1	1	0	AD	A register and D register
-D	0	0	1	1	1	1		1	1	1	AMD	A register, Memory[A], and D register
-A	1	1	0	0	1	1	-M					
D+1	0	1	1	1	1	1			j1	j2	j3	
A+1	1	1	0	1	1	1	M+1	(out < 0)	(out = 0)	(out > 0)	Mnemonic	Effect
D-1	0	0	1	1	1	0		0	0	0	null	No jump
A-1	1	1	0	0	1	0	M-1	0	0	1	JGT	If out > 0 jump
D+A	0	0	0	0	1	0	D+M	0	1	0	JEQ	If out = 0 jump
D-A	0	1	0	0	1	1	D-M	0	1	1	JGE	If out ≥ 0 jump
A-D	0	0	0	1	1	1	M-D	1	0	0	JLT	If out < 0 jump
D&A	0	0	0	0	0	0	D&M	1	0	1	JNE	If out ≠ 0 jump
D A	0	0	0	0	0	0	D&M	1	1	0	JLE	If out ≤ 0 jump
D A	0	1	0	1	0	1	D M	1	1	1	JMP	Jump

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
(LLOOP)
    @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
```



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 Virtual Machine



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.

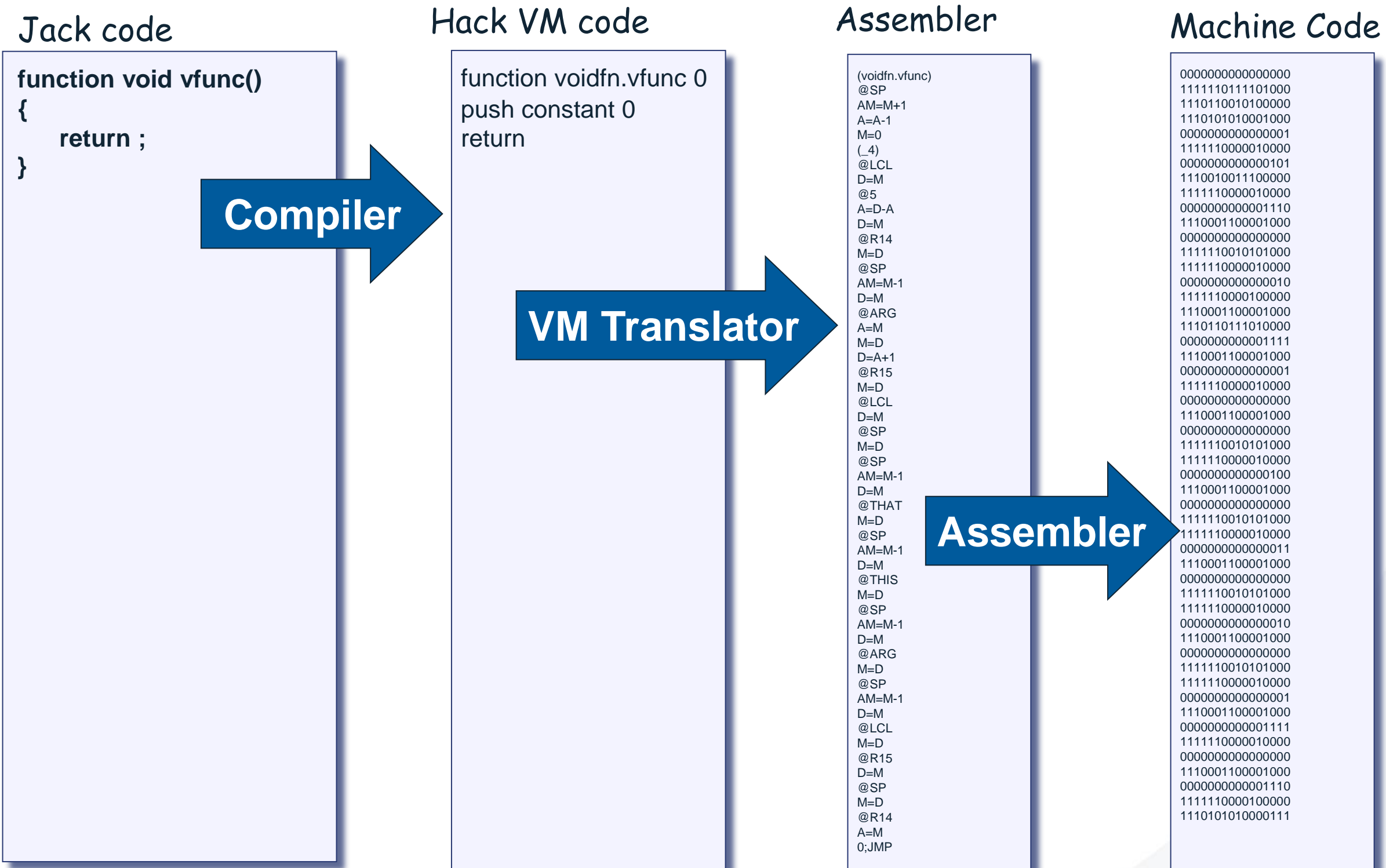
Compiler

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
...
```

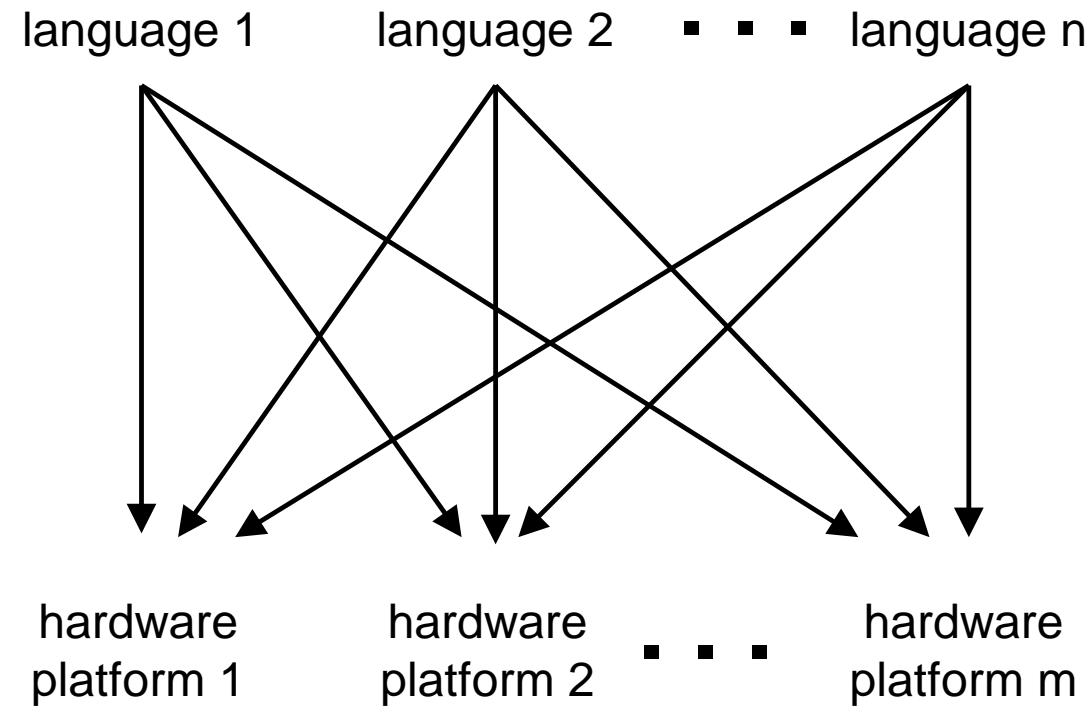


Motivation



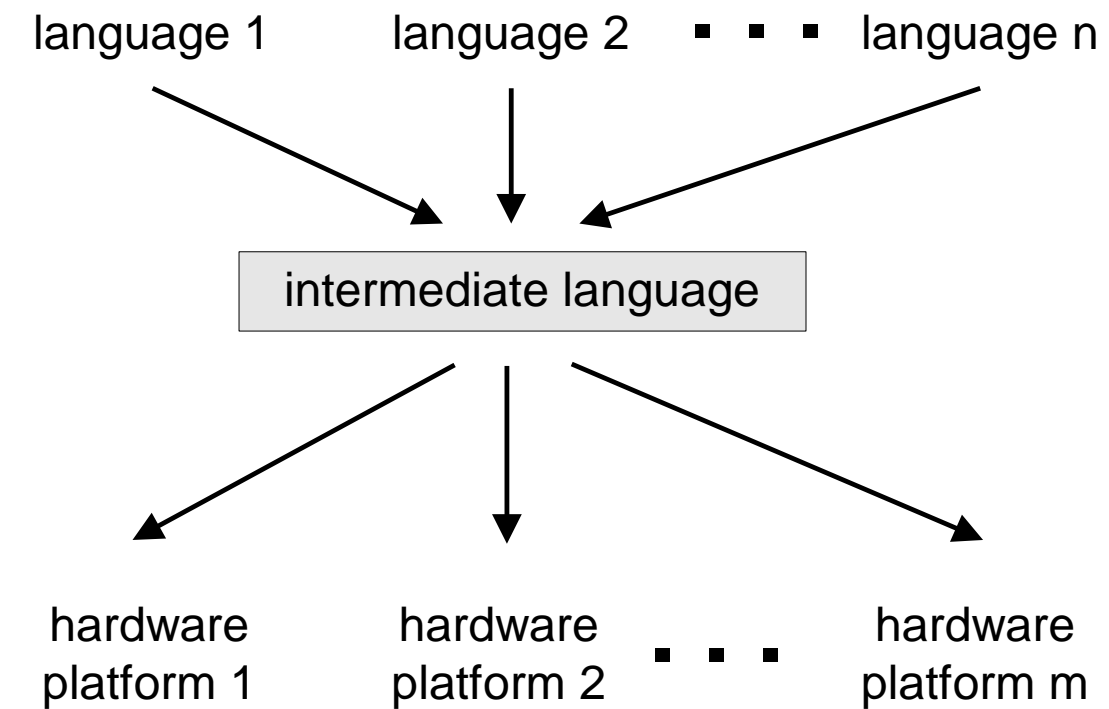
Compilation models

direct compilation:



requires $n \cdot m$ translators

2-tier compilation:

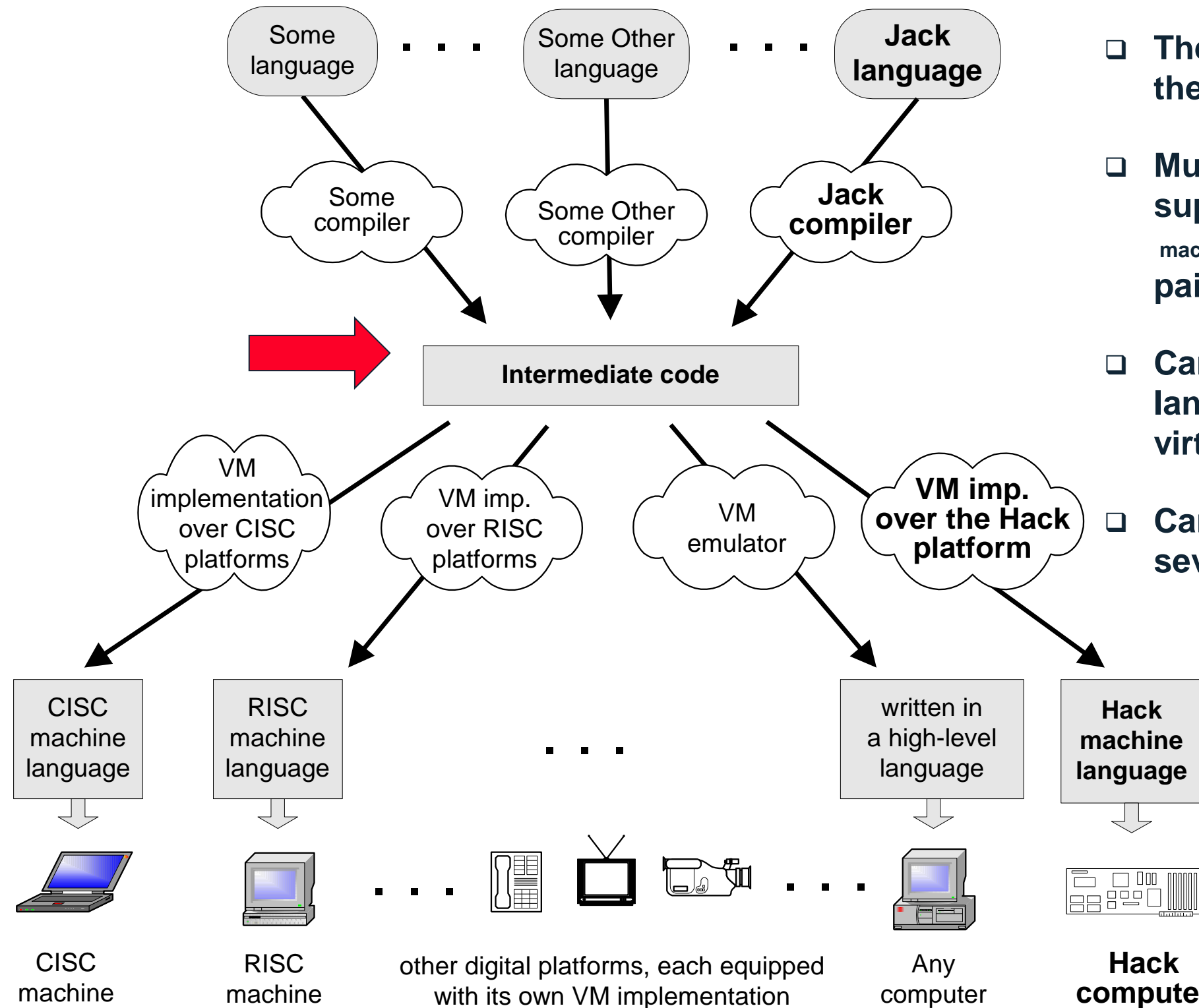


requires $n + m$ translators

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.

The big picture



The intermediate code:

- ❑ The interface between the 2 compilation stages
- ❑ Must be sufficiently general to support many $\langle \text{high-level language, machine-language} \rangle$ 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.



Software implementation: Hack VM emulator

The screenshot shows the Virtual Machine Emulator (1.4b3) interface. The title bar indicates the file path is G:\examples\add. The menu bar includes File, View, Run, and Help. The toolbar contains icons for file operations and execution controls, along with a slider for animation speed (Slow to Fast) and dropdowns for View (Program flow, Script) and Format (Decimal). The main window is divided into several panels:

- Program:** A list of instructions. Instruction 11, 'add', is highlighted in yellow. An orange callout box labeled 'VM code' points to this instruction.
- Static:** A table for static variables.
- Local:** A table for local variables. An orange callout box labeled 'virtual memory segments' points to this table.
- Argument:** A table for arguments.
- This:** A table for 'this' pointer.
- That:** A table for 'that' pointer.
- Temp:** A table for temporary variables.
- Global Stack:** A table showing memory addresses and values. An orange callout box labeled 'global stack' points to this table.
- RAM:** A table showing memory addresses and values. An orange callout box labeled 'host RAM' points to this table. A blue note '(the RAM is not part of the VM)' is present next to the RAM table.
- Stack:** A table showing the current stack state. An orange callout box labeled 'working stack' points to this table.
- Call Stack:** A list of active function calls: Sys.init, Main.main, and Main.add.
- emulator controls:** A set of buttons and sliders for controlling the emulator's execution.
- default test script:** A text area containing the script 'repeat { vmstep; }'. An orange callout box labeled 'default test script' points to this area.

Lecture plan

Goal: Specify and implement a VM model and language:

<u>Arithmetic / Boolean commands</u>	<u>Program flow commands</u>
add	label (declaration)
sub	goto (label)
neg	if-goto (label)
eq	
gt	
lt	
and	
or	
not	
<u>Memory access commands</u>	<u>Function calling commands</u>
pop x (pop into x, which is a variable)	function (declaration)
push y (y being a variable or a constant)	call (a function)
	return (from a function)

This week

Next week

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

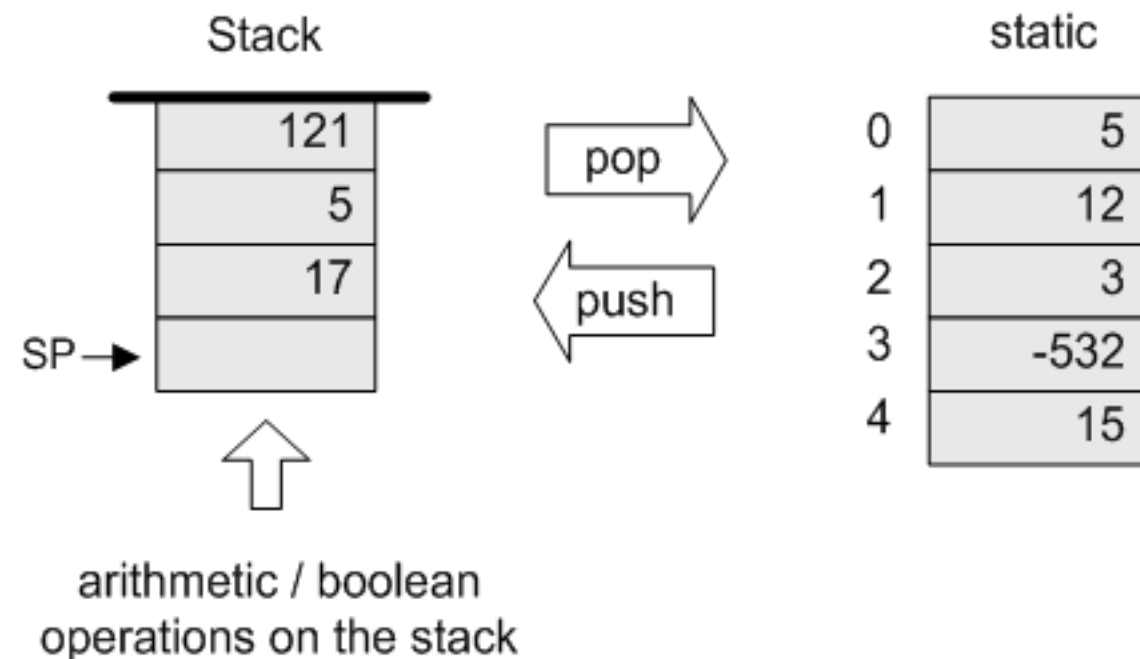
Our VM model is *stack-oriented*

All operations are done on a stack

Data is saved in several separate *memory segments*

All the memory segments behave the same

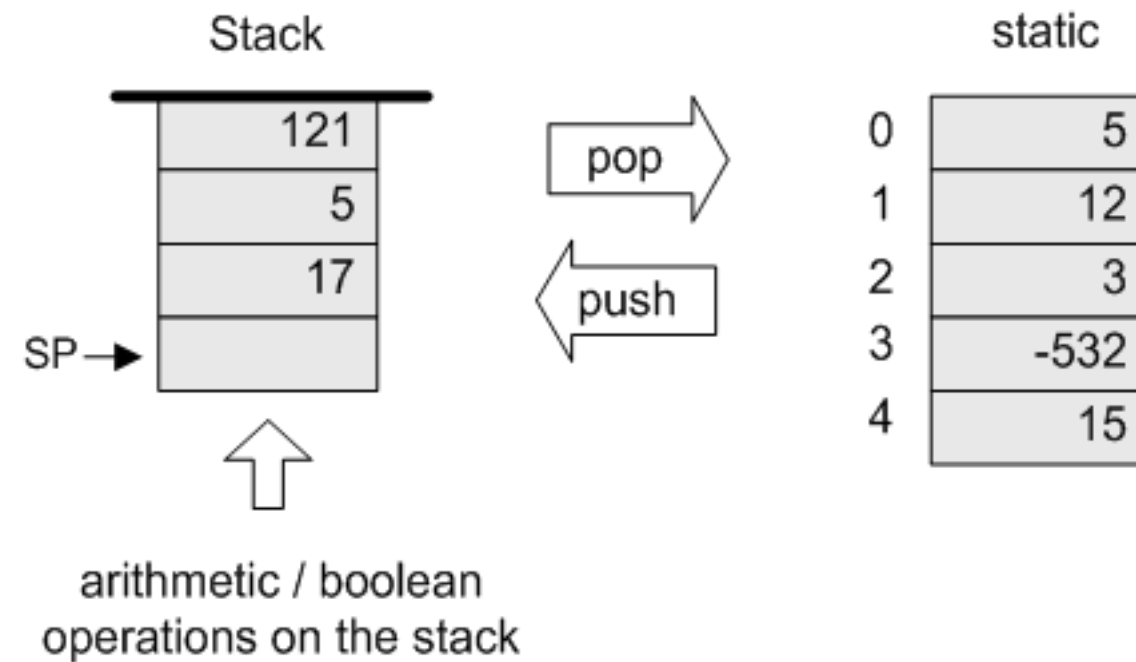
One of the memory segments *m* is called *static*, and we will use it (as an arbitrary example) in the following examples:



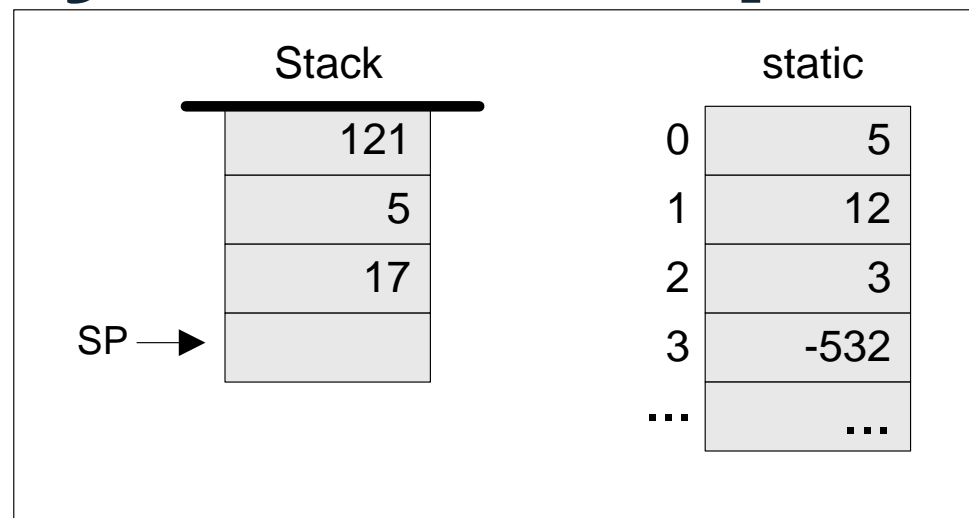
Data types

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)

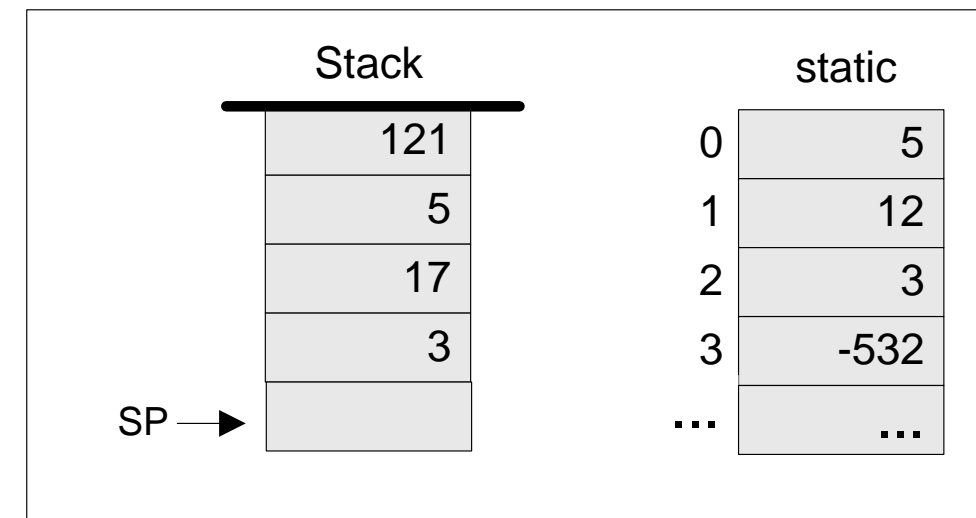
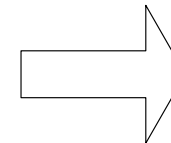


Memory access operations

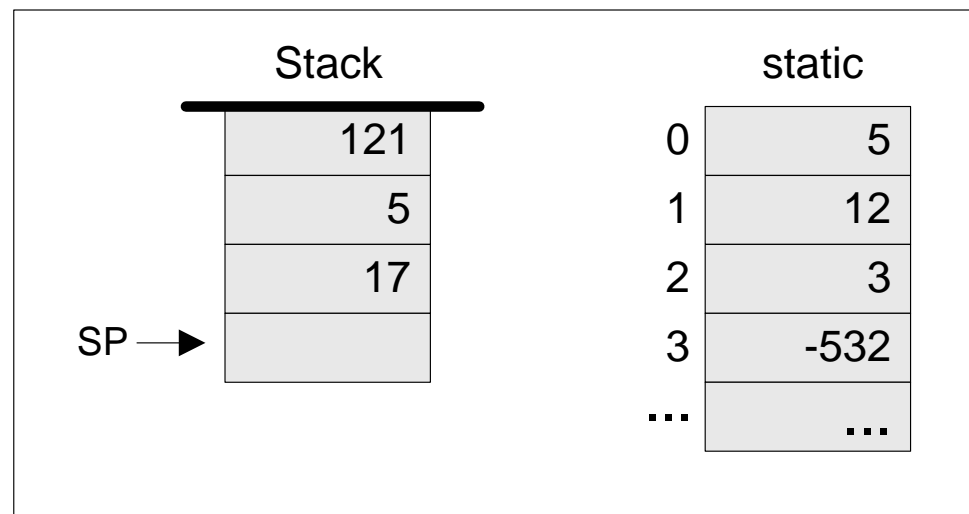


(before)

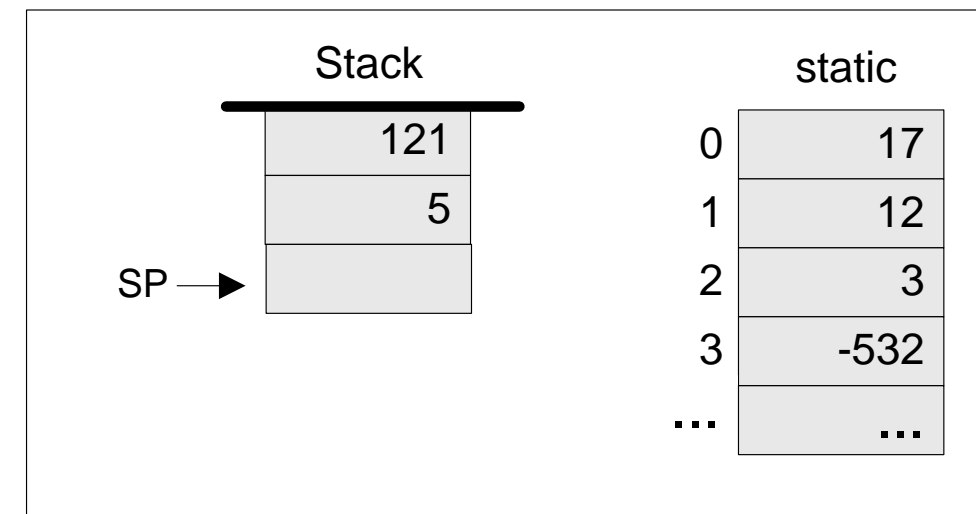
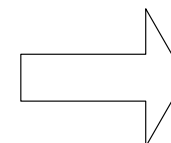
push
static 2



(after)



pop
static 0



The stack:

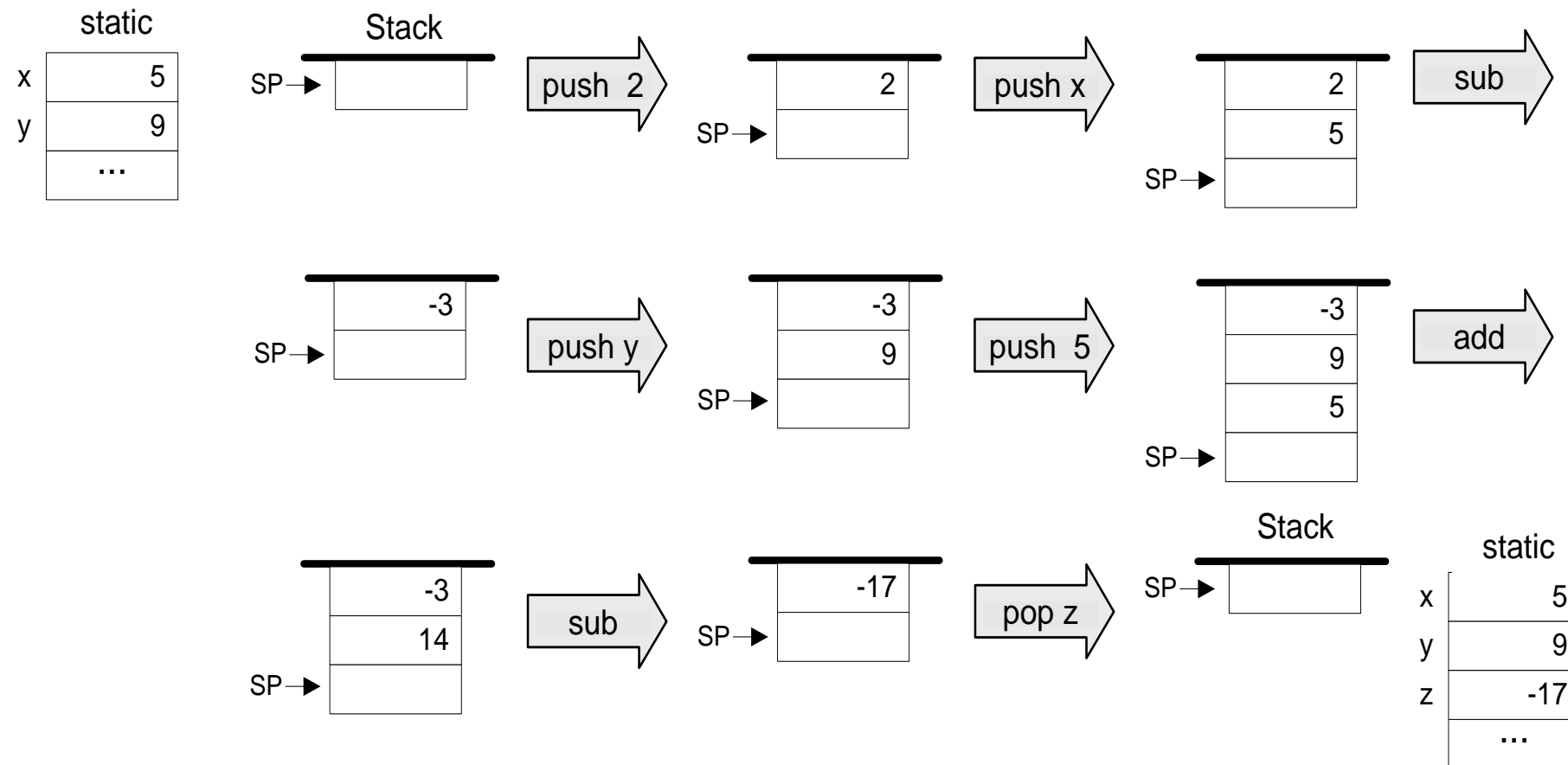
- A classical LIFO data structure
- Elegant and powerful
- Several hardware / software implementation options.

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)



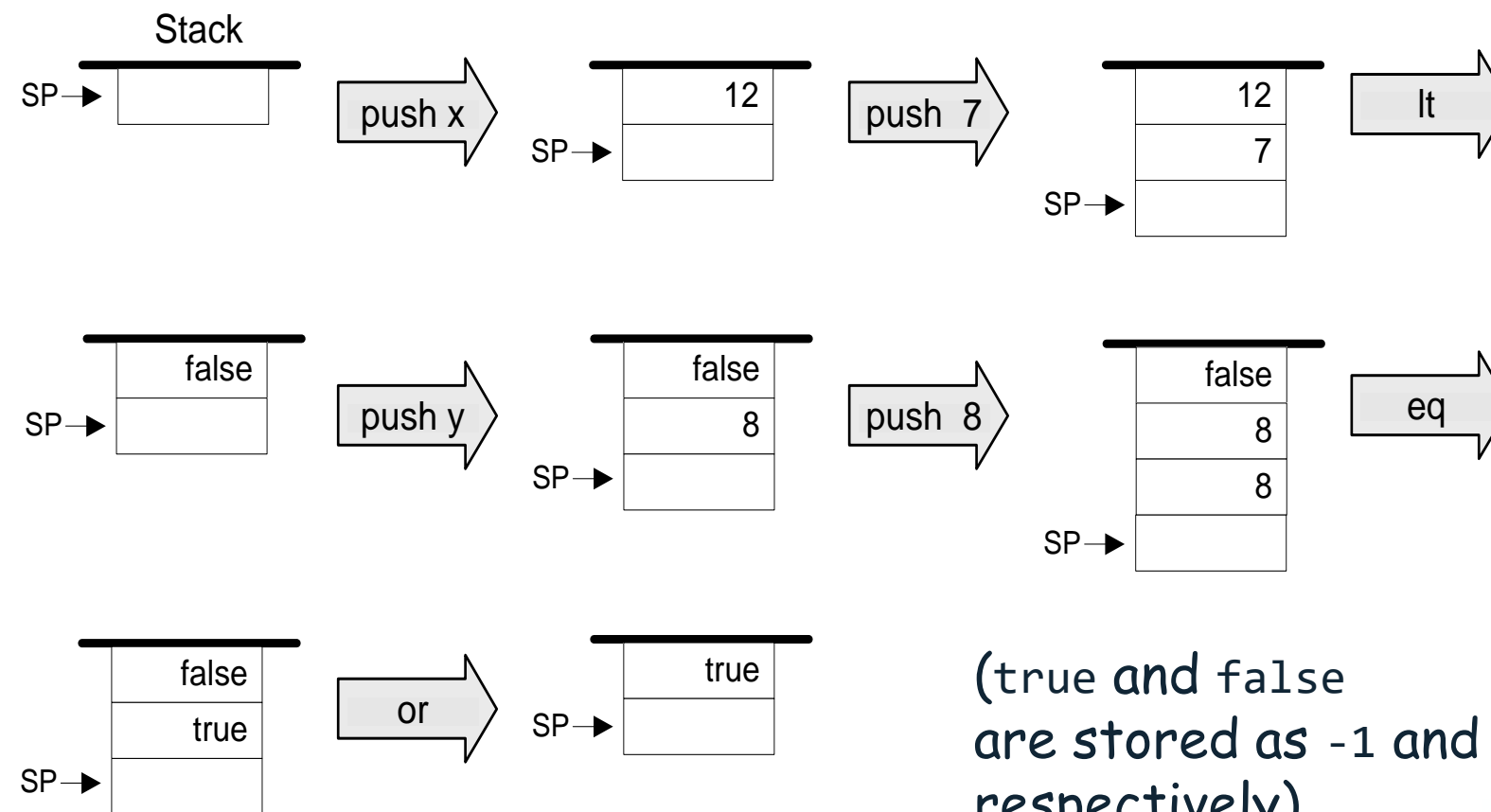
Evaluation of Boolean expressions

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)

static	
x	12
y	8
	...



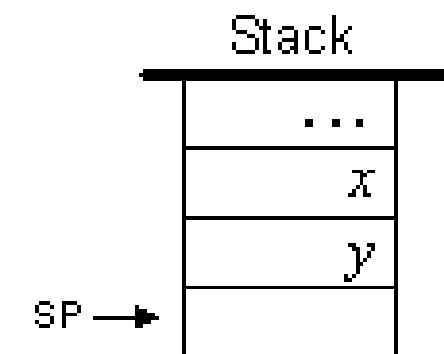
(true and false
are stored as -1 and 0,
respectively)



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	$x \text{ And } y$	Bit-wise
or	$x \text{ Or } y$	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 object-oriented 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.



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

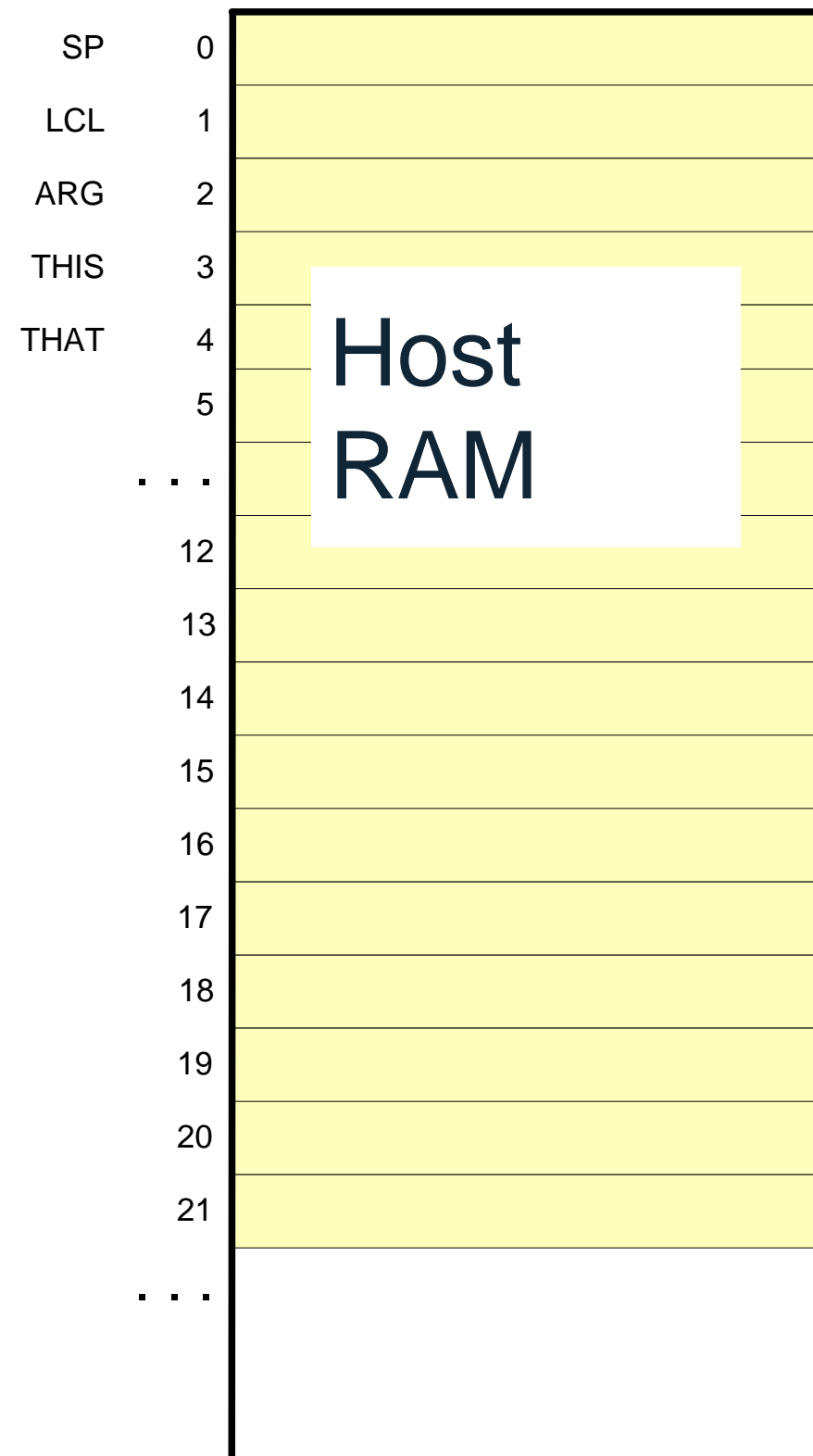
Notes:

(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.

VM implementation on the Hack platform



The stack: 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

static, constant, temp, pointer:

Global memory segments, all functions see the same four segments

local, argument, this, that:

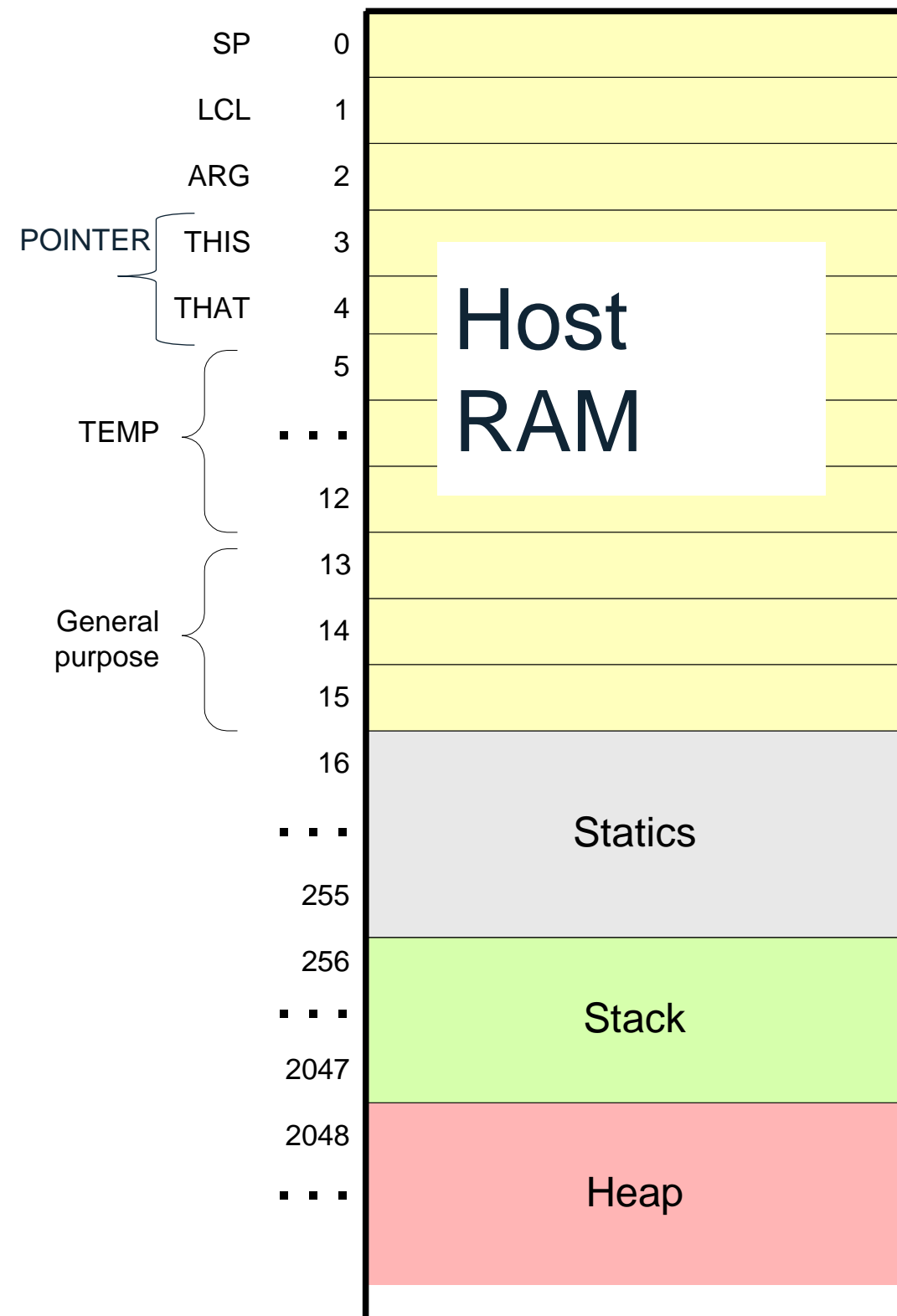
these segments are local at the function level; each function sees its own, private copy of each one of these four segments

The challenge:

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 $\text{RAM}[\text{segmentBase} + i]$

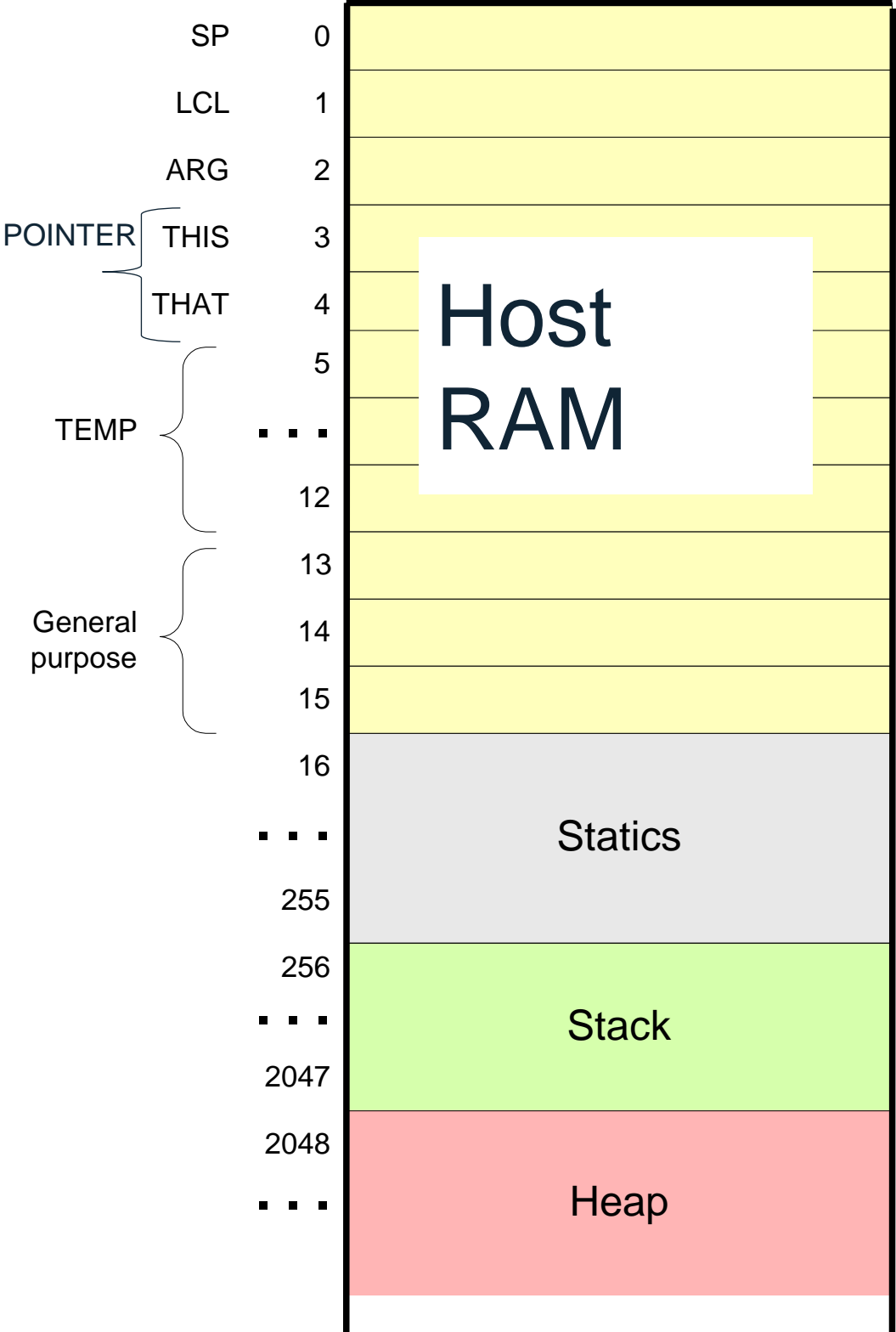
constant: a truly virtual segment:

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 Bob.vm)
- ❑ 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

- `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

- pop local 2

@SP

AM=M-1

D=M

@LCL

A=M+1

A=A+1

M=D



VM Translator Parsing

- eq
@SP
AM=M-1
D=M
@SP
AM=M-1
D=M-D
@labelTrue
D;JEQ
D=0
@labelFalse
0;JMP
(labelTrue)
D=-1
(labelFalse)
@SP
A=M
M=D
@SP
M=M+1

This Week

- Review Chapters 6 & 7 of the Text Book (if you haven't already)
- Assignment 4 due this Friday
- Assignment 5 Available Shortly – Due week 8.
- Review Chapter 8 of the Text Book before week 7.

