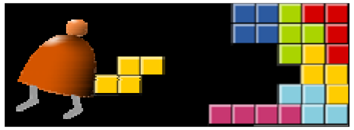
Virtual Machine

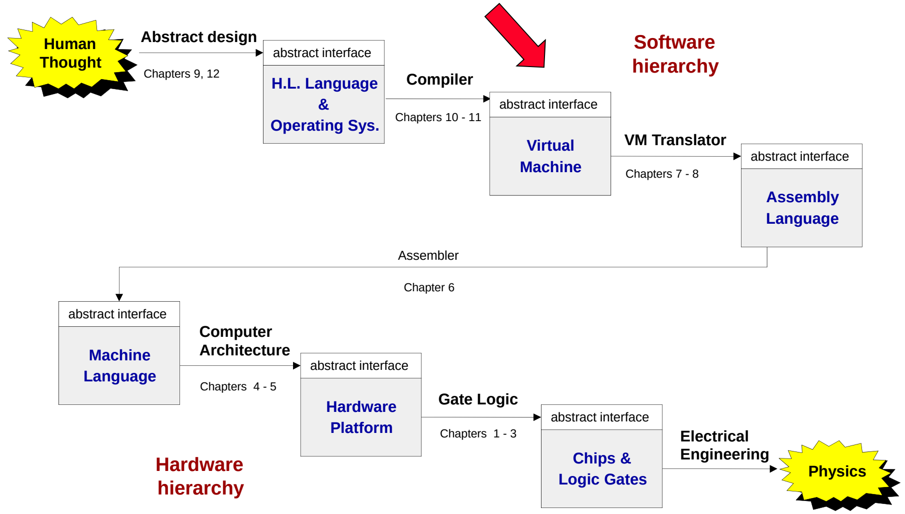
Part I: Stack Arithmetic



Building a Modern Computer From First Principles

www.nand2tetris.org

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.

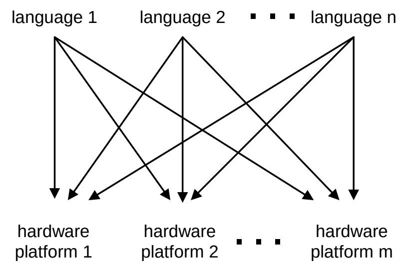
Compiler

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
...
```

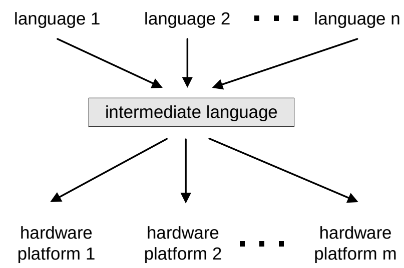

Compilation models

direct compilation:



requires $n \cdot m$ translators

2-tier compilation:

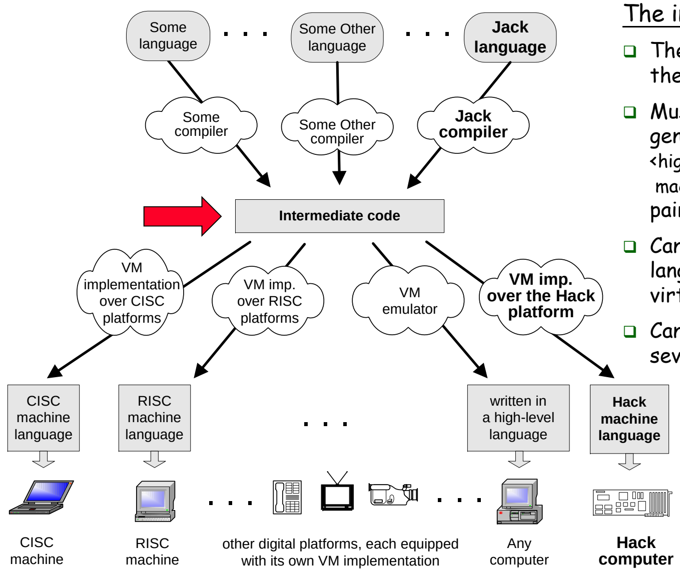


requires $n + m$ translators

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.

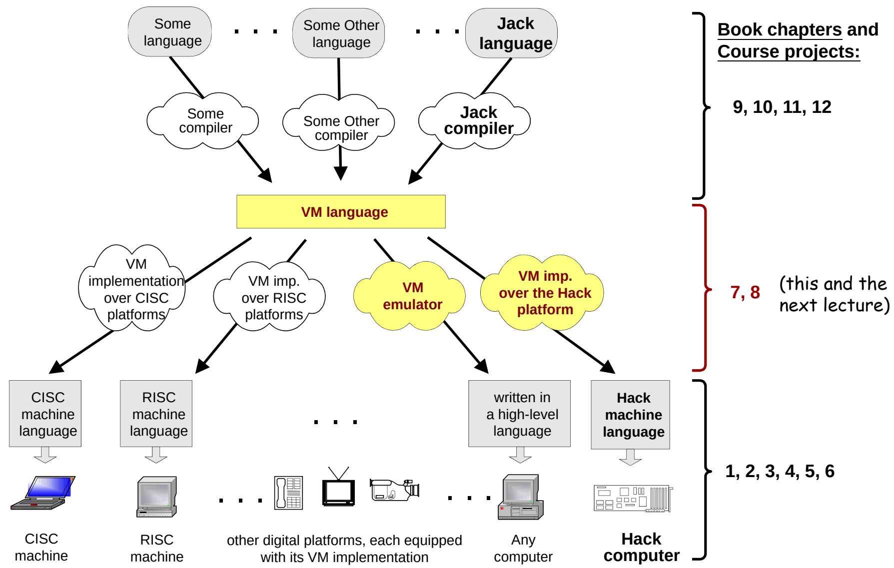
The big picture



The intermediate code:

- The interface between the 2 compilation stages
- Must be sufficiently general to support many <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.

Focus of this lecture (yellow):



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

Goal: Specify and implement a VM model and language:

Arithmetic / Boolean commands

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

This lecture

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)

Next lecture

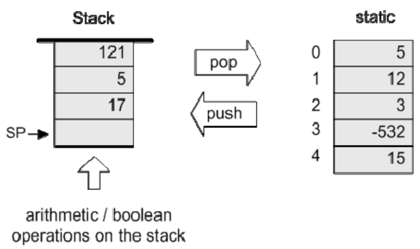
Function calling commands

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

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*

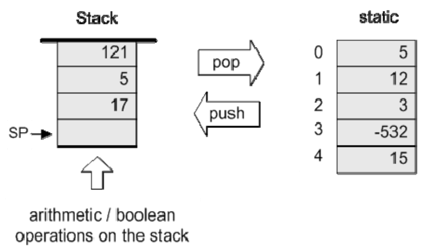
- All operations are done on a stack
- Data is saved in several separate *memory segments*
- All the memory segments behave the same
- One of the memory segments m is called `static`, and we will use it (as an arbitrary example) in the following examples:



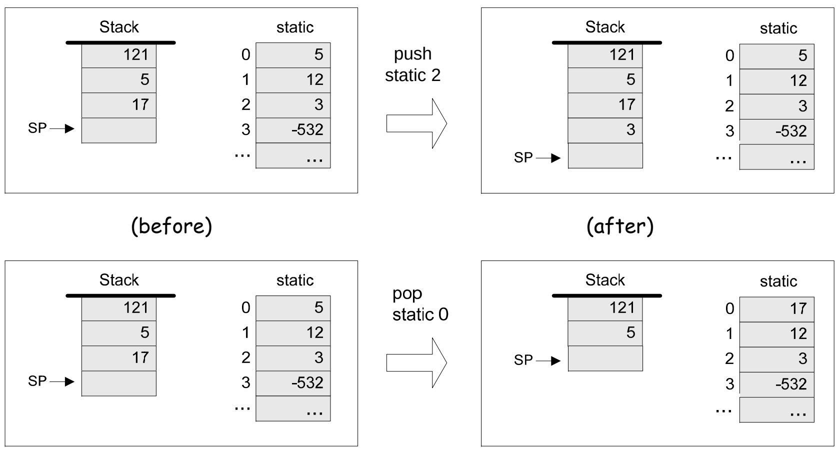
Data types

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 (`0` and `-1`, standing for `true` and `false`)
- a pointer (memory address)



Memory access operations



The stack:

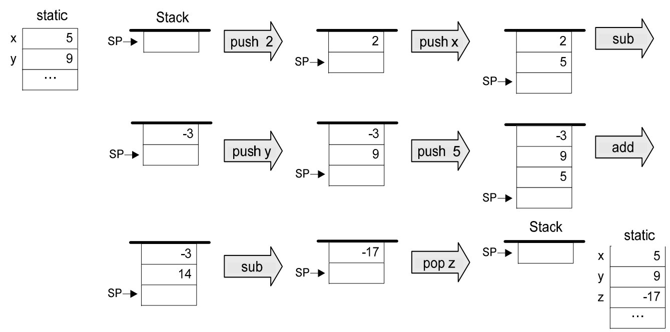
- A classical LIFO data structure
- Elegant and powerful
- Several hardware / software implementation options.

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`)

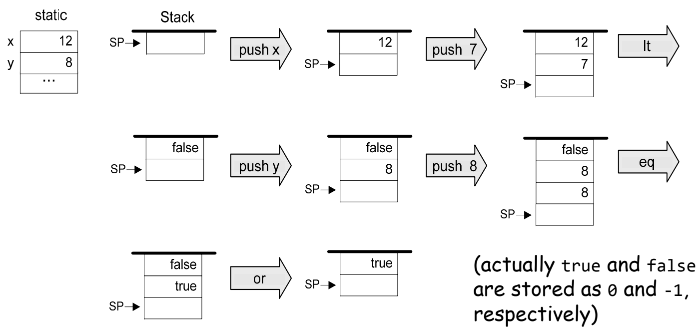


Evaluation of Boolean expressions

VM code (example)

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

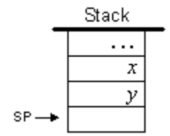
(suppose that
x refers to `static 0`, and
y refers to `static 1`)



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



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`.

Memory segments and memory 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 different roles of the eight memory segments will become relevant when we'll talk about the compiler

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

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

So, we'll now see an example of a VM program

The example includes three new VM commands:

- `function` *functionSymbol* // function declaration
- `label` *labelSymbol* // label declaration
- `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:

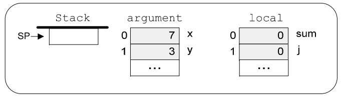
```
push x
push n
gt
if-goto loop          // Note that x, n, and the truth value were removed from the stack.
```


VM programming (example)

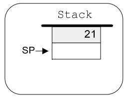
High-level code

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

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



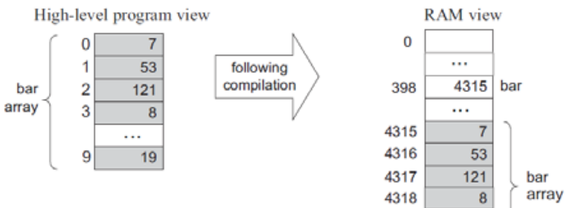
VM code (first approx.)

```
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
```


Handling array



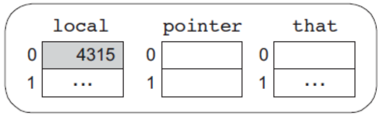
(Actual RAM locations of program variables are run-time dependent, and thus the addresses shown here are arbitrary examples.)

VM code

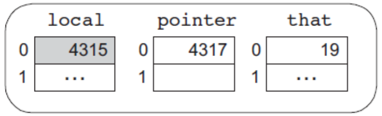
```
/* Assume that the bar array is the first local variable declared in the
   high-level program. The following VM code implements the operation
   bar[2]=19,  i.e., *(bar+2)=19. */
 push local 0      // Get bar's base address
 push constant 2
 add
 pop pointer 1     // Set that's base to (bar+2)
 push constant 19
 pop  that 0       // *(bar+2)=19
 ...
```


Handling array

Virtual memory segments just before the bar[2]=19 operation:

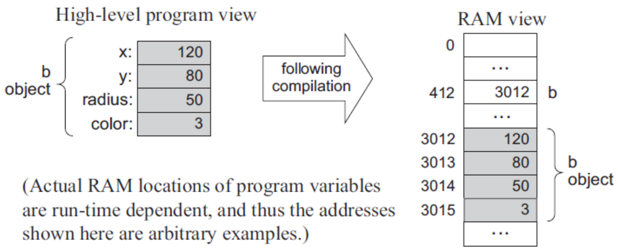


Virtual memory segments just after the bar[2]=19 operation:



(that 0 is now aligned with RAM[4317])

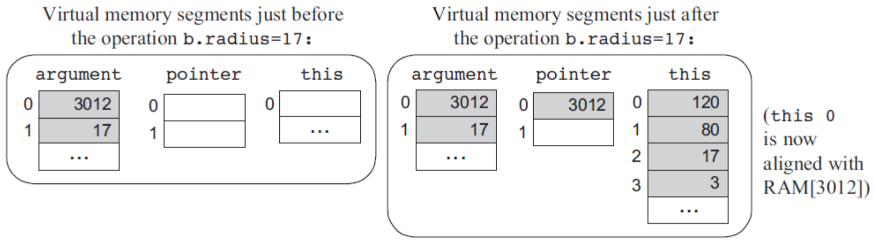
Handling objects



VM code

```
/* Assume that the b object and the r integer were passed to the function as
   its first two arguments. The following code implements the operation
   b.radius=r. */
 push argument 0  // Get b's base address
 pop pointer 0    // Point the this segment to b
 push argument 1  // Get r's value
 pop this 2       // Set b's third field to r
 ...
```


Handling objects



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`).

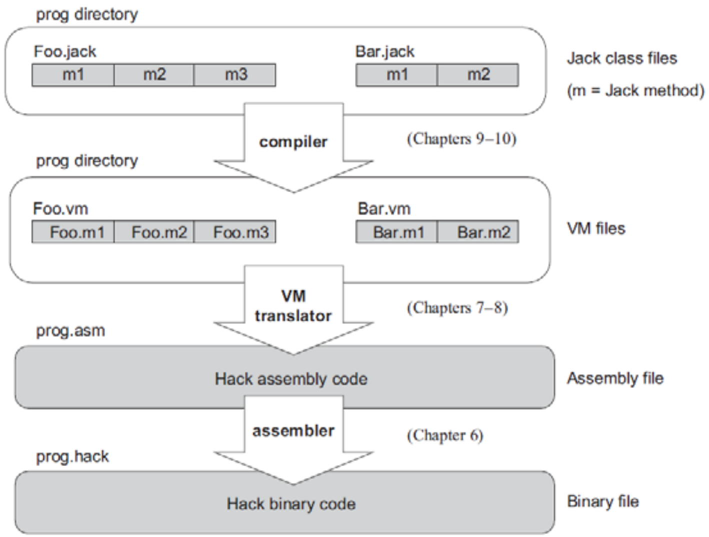


Figure 7.8 Program elements in the Jack-VM-Hack platform.

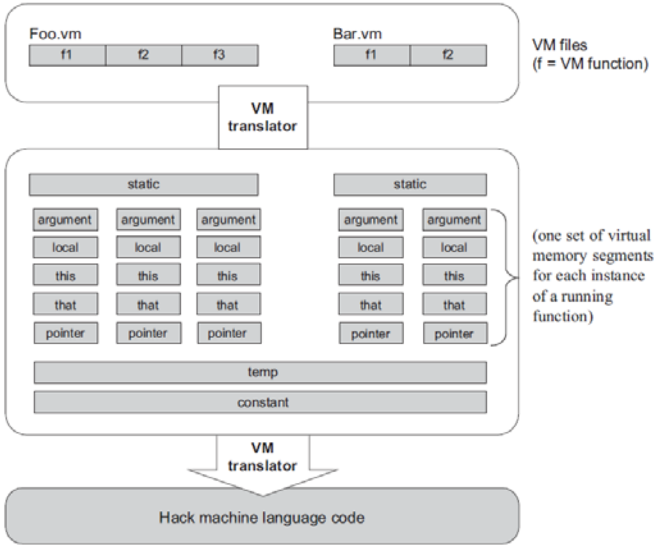


Figure 7.7 The virtual memory segments are maintained by the VM implementation.

Lecture plan

Goal: Specify and implement a VM model and language:

Arithmetic / Boolean commands

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

This lecture

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)

Next lecture

Function calling commands

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

Method: (a) specify the abstraction (stack, memory segments, commands)

(b) propose how to implement the abstraction over the Hack platform.

Implementation

VM implementation options:

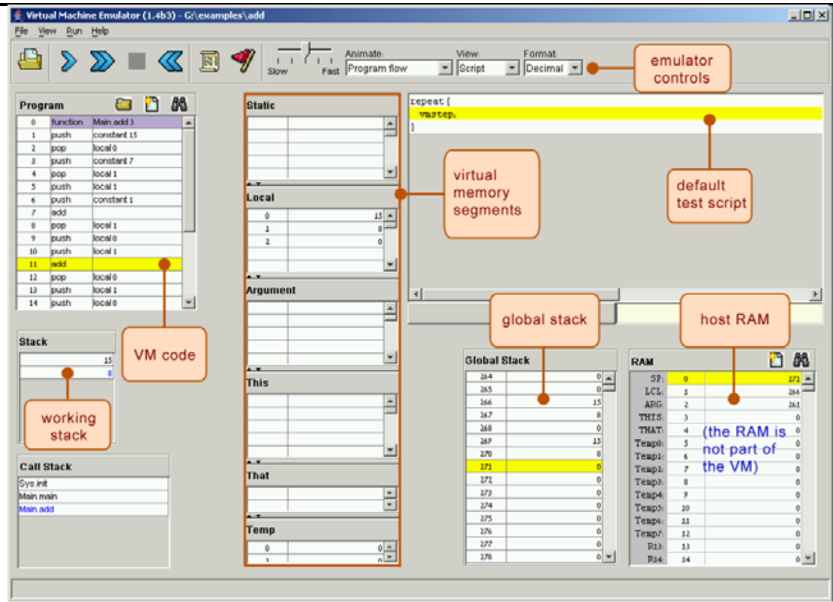
- Software-based (e.g. emulate the VM model using Java)
- Translator-based (e. g. translate VM programs into the Hack machine language)
- Hardware-based (realize the VM model using dedicated memory and registers)

Two well-known translator-based implementations:

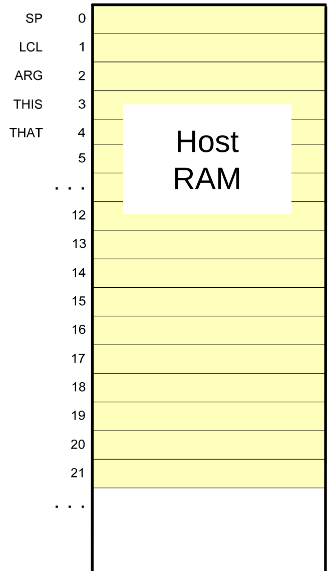
JVM: Javac translates Java programs into bytecode;
The JVM translates the bytecode into
the machine language of the host computer

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

Software implementation: Our VM emulator (part of the course software suite)



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.

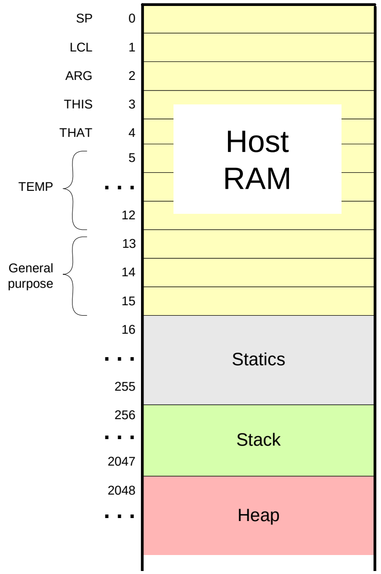
The tip of this stack if the working stack of the current function

`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`

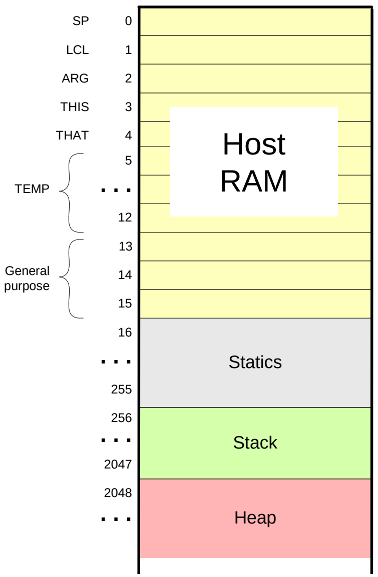
`static`: mapped on `RAM[16 ... 255]`;
each segment reference `static` *i* appearing in a VM file named `f` is compiled to the assembly language symbol `f.i` (recall that the assembler further maps such symbols to the RAM, from address 16 onward)

`local,argument,this,that`: these method-level segments are mapped somewhere from address 2048 onward, in an area called "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 a virtual segment:
access to `constant` *i* is implemented by supplying the constant *i*.

`pointer`: discussed later.

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 implementation on the Hack platform

- **`push constant 1`**

```
@7
D=A
@SP
A=M
M=D
@SP
M=M+1
```

- **`add`**

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

- **`pop local 2`**

```
@LCL
D=M
@2
D=D+A
@R15
M=D
@SP
AM=M-1
D=M
@R15
A=M
M=D
```

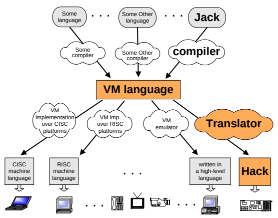
Proposed VM translator implementation: Parser module

Parser: Handles the parsing of a single `.vm` file, and encapsulates access to the input code. It reads VM commands, parses them, and provides convenient access to their components. In addition, it removes all white space and comments.			
Routine	**Arguments**	**Returns**	**Function**
Constructor	Input file / stream	--	Opens the input file/stream and gets ready to parse it.
`hasMoreCommands`	--	boolean	Are there more commands in the input?
`advance`	--	--	Reads the next command from the input and makes it the current command. Should be called only if `hasMoreCommands` is true. Initially there is no current command.
`commandType`	--	`C_ARITHMETIC, C_PUSH, C_POP, C_LABEL, C_GOTO, C_IF, C_FUNCTION, C_RETURN, C_CALL`	Returns the type of the current VM command. `C_ARITHMETIC` is returned for all the arithmetic commands.
`arg1`	--	string	Returns the first arg. of the current command. In the case of `C_ARITHMETIC`, the command itself (`add`, `sub`, etc.) is returned. Should not be called if the current command is `C_RETURN`.
`arg2`	--	int	Returns the second argument of the current command. Should be called only if the current command is `C_PUSH`, `C_POP`, `C_FUNCTION`, or `C_CALL`.

Proposed VM translator implementation: CodeWriter module

CodeWriter: Translates VM commands into Hack assembly code.			
Routine	**Arguments**	**Returns**	**Function**
Constructor	Output file / stream	--	Opens the output file/stream and gets ready to write into it.
`setFileName`	fileName (string)	--	Informs the code writer that the translation of a new VM file is started.
`writeArithmetic`	command (string)	--	Writes the assembly code that is the translation of the given arithmetic command.
`WritePushPop`	command (`C_PUSH` or `C_POP`), segment (string), index (int)	--	Writes the assembly code that is the translation of the given `command`, where `command` is either `C_PUSH` or `C_POP`.
`Close`	--	--	Closes the output file.
Comment: More routines will be added to this module in the next lecture / chapter 8.			

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)