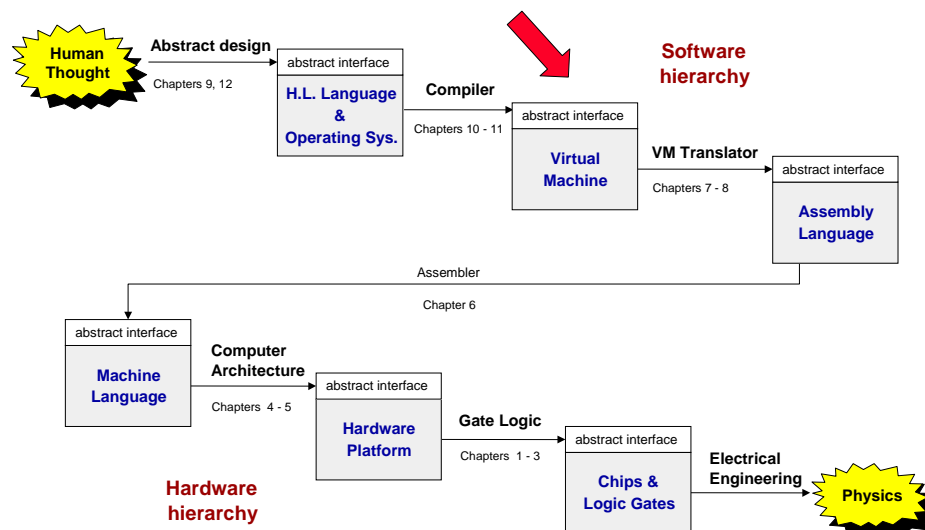


Virtual Machine (Part I)

Shimon Schocken

Spring 2005

Where we are at:



Motivation

```

class Main {
  static int x;

  function void main() {
    // Input and multiply 2 numbers
    var int a, b;
    let a = Keyboard.readInt("Enter a number");
    let b = Keyboard.readInt("Enter a number");
    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 not(j = 0) {
    let result = result + x;
    let j = j - 1;
  }
  return result;
}
    
```

Ultimate goal:

Compile abstract programs into executable code.

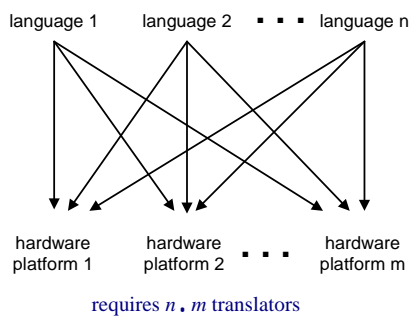
Compiler

```

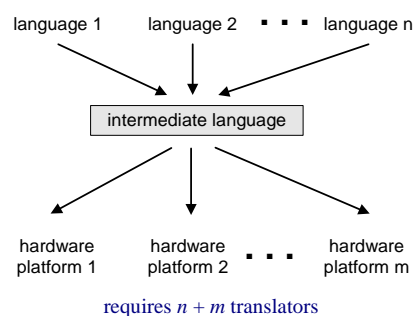
...
@j
M=D
@y
M=0
(LOOP)
@j
D=M
@y
D=D-A
@END
D;JGT
@j
D=M
@temp
M=D+M
@j
M=M+1
@LOOP
0;JMP
(END)
@END
0;JMP
...
    
```

Compilation models

direct compilation:



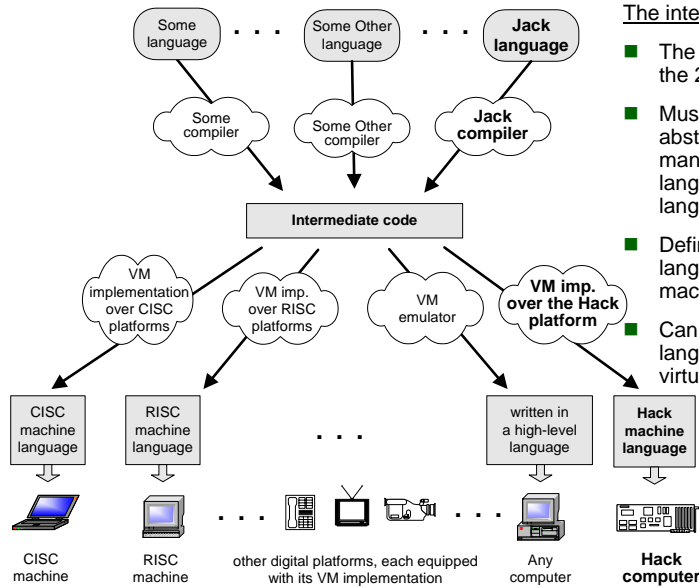
2-tier compilation:



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.

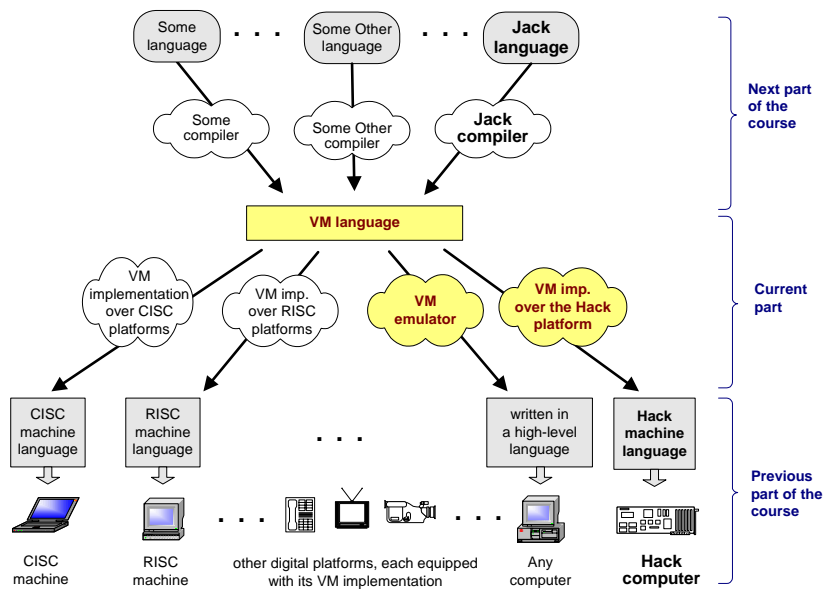
The Hack/Jack platform



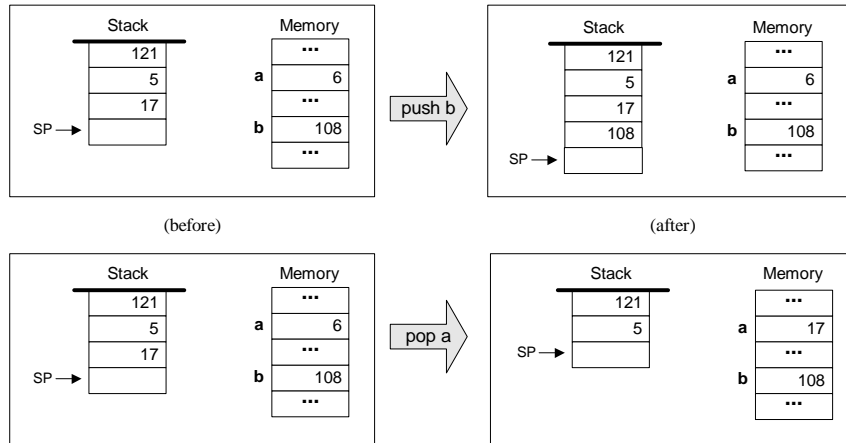
The intermediate code:

- The interface between the 2 compilation stages
- Must be sufficiently abstract to support many (high-level language, machine language) pairs
- Defines a stand-alone language of an abstract machine
- Can be modeled as the language of an abstract virtual machine (VM).

The Hack/Jack platform

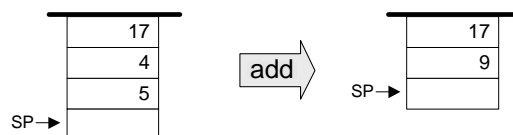


A stack machine - based VM



- Major data structure
- Elegant and powerful
- Implementation options.

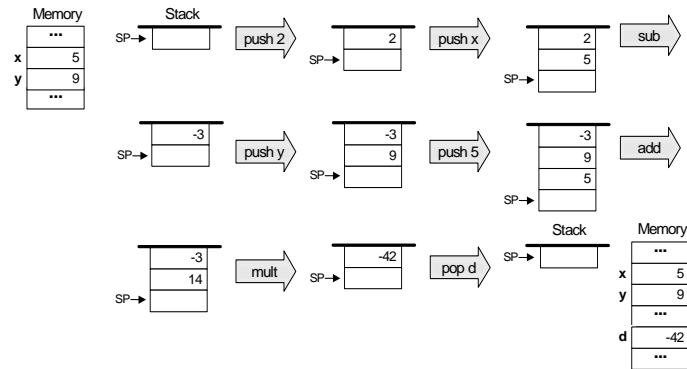
Stack arithmetic



- Typical operation:
 - Pops two arguments from the stack
 - Adds their value
 - Pushes the result onto the stack.
- Impact: the operands are replaced with the operation's result
- In general: same model works for all arithmetic and Boolean operations.

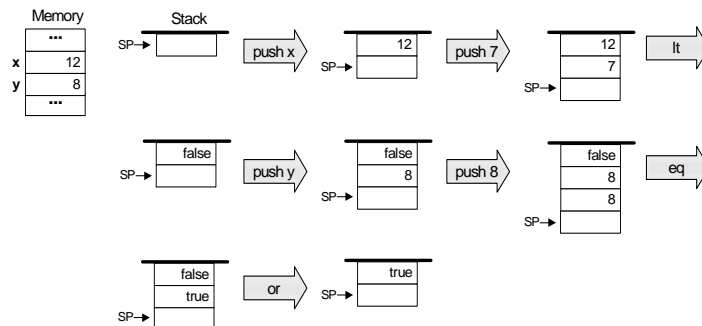
Evaluation of arithmetic expressions

```
// d=(2-x)*(y+5)
push 2
push x
sub
push y
push 5
add
mult
pop d
```



Evaluation of Boolean expressions

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



VM specification: data types

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

- Integer
- Boolean
- Pointer

VM specification: commands

Arithmetic / Boolean commands

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

Memory access commands

`pop segment i`
`push segment i`

Program flow commands

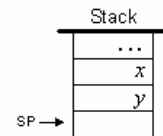
`label` (declaration)
`goto` (label)
`if-goto` (label)

Function calling commands

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

Arithmetic and Boolean commands

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



Memory access commands

(So far, pop/push commands acted on a global memory. This was a conceptual simplification)

Memory segments

- **argument**: stores the current function's arguments
- **local**: stores the function's local variables
- **static**: stores global variables, shared by all functions in the same .vm file
- **constant**: pseudo segment holding all the constants in the range 0...32767
- **this, that**: general-purpose segments; can be made to correspond to different areas in the heap
- **Pointer**: used to align **this** and **that**
- **Temp**: fixed 8-entry segment that holds temporary variables for general use; Shared by all VM functions in the program.

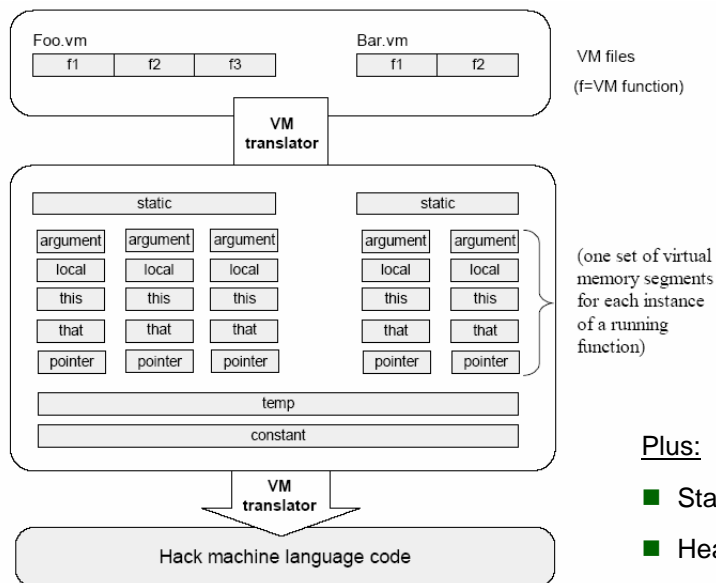
Memory access commands:

```
pop segment i
push segment i
```

VM program structure

- A VM *program* is a collection of one or more .vm Files (classes)
- Each .vm file is a collection of one or more *functions* (methods)
- Each function is a collection of VM *commands*
- Each VM *command*
 - appears in a separate line
 - Has 0, 1, or 2 arguments
 - May have in-line // comments.

Program structure and memory segments



Some motivation

VM programming examples:

- Arithmetic task
- Array handling task
- Object handling task.

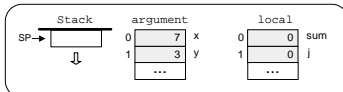
(These example don't belong to this lecture, and are given here for motivation only.)

Arithmetic example

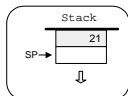
High-level code

```
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:



Pseudo VM code

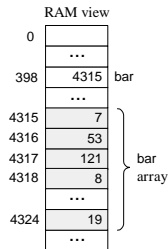
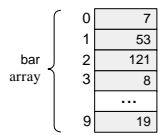
```
function mult(x,y)  
  push 0  
  pop result  
  push y  
  pop j  
  label loop  
  push j  
  push 0  
  eq  
  if-goto end  
  push result  
  push x  
  add  
  pop result  
  push j  
  push 1  
  sub  
  pop j  
  goto loop  
  label end  
  push result  
  return
```

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
```

Array handling example

High-level program view

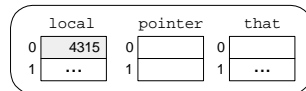


(Actual RAM locations of program variables are run-time dependent, and thus the addresses shown here are arbitrary examples.)

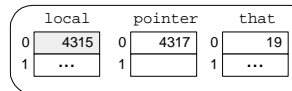
```
/* Assume that bar is the
first local variable declared
in the high-level program. The
code below implements
bar[2]=19, or *(bar+2)=19. */
```

```
// Get bar's base address:
push local 0
push constant 2
add
// Set that's base to (bar+2):
pop pointer 1
push constant 19
// *(bar+2)=19:
pop that 0
```

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



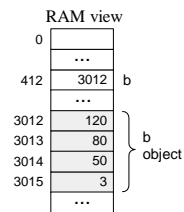
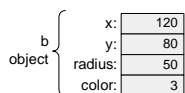
Virtual memory segments
Just after the bar[2]=19 operation:



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

Object handling example

High level program view

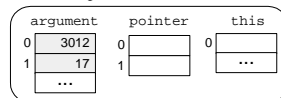


(Actual RAM locations of program variables are run-time dependent, and thus the addresses shown here are arbitrary examples.)

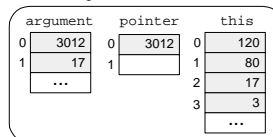
```
/* Assume that b and r were
passed to the function as
its first two arguments.
The following code
implements the operation
b.radius=r. */
```

```
// Get b's base address:
push argument 0
// Point the this seg. to b:
pop pointer 0
// Get r's value
push argument 1
// Set b's third field to r:
pop this 2
```

Virtual memory segments just before
the operation b.radius=17:



Virtual memory segments just after
the operation b.radius=17:



(this 0
is now
aligned with
RAM[3012])

The VM language – summary

- Arithmetic / Boolean commands (`add`, `sub`, `eq`, ...)
- Memory commands (`pop`, `push`)
- Program flow commands (`label`, `goto`, `if-goto`)
- Function calling commands (`function`, `call`, `return`)

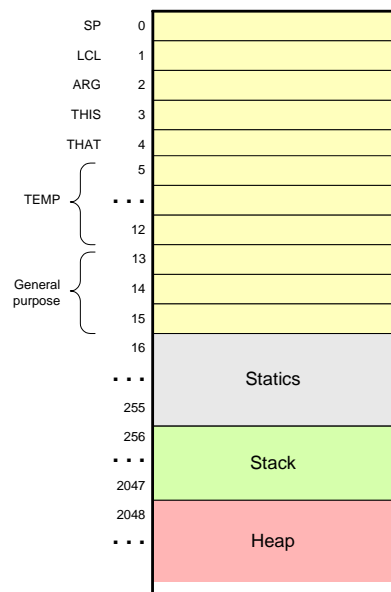
VM implementation options:

- Hardware-based (e.g. Hack)
- Software-based (emulation)

Famous VM implementations:

- JRE (runs bytecode)
- CLR (runs IL programs)

VM implementation on the Hack platform



- **Local, argument, this, that:** mapped directly on the host RAM. The base addresses of these segments are kept in the registers `LCL`, `ARG`, `THIS`, `THAT`. Access to the i -th entry of a segment results in accessing the $(base + i)$ word in the RAM
- **static:** static variable number j in a VM file f is represented as the assembly language symbol `f.j`.
- **constant:** truly a virtual segment. Access to `constant i` results in supplying the constant i
- **pointer, temp:** see the book.

Parser module (proposed design)

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 <code>hasMoreCommands()</code> 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 argument 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.

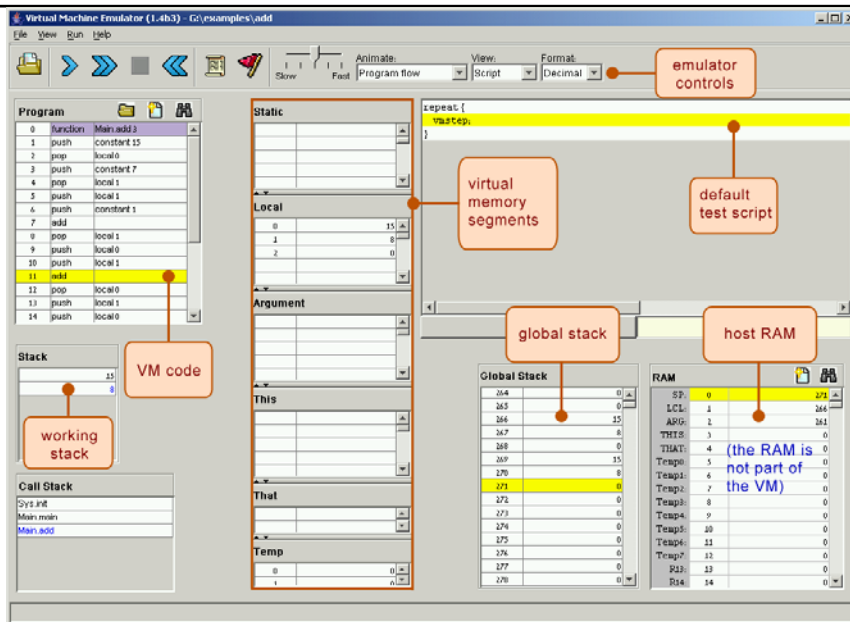
CodeWriter module (proposed design)

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 chapter 8.

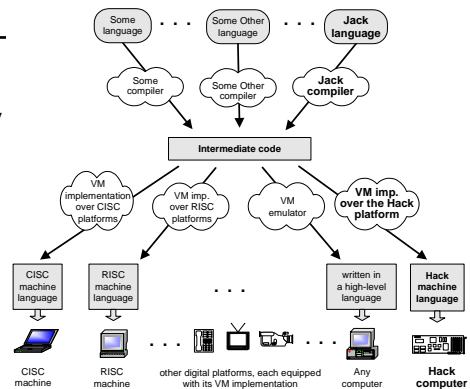
The VM emulator



Perspective

Pros

- Code transportability: compiling for different platforms require replacing only the VM implementation
- Language inter-operability: code of multiple languages can be shared using the same VM
- Common software libraries
- Modularity:
 - Improvements in the VM implementation are shared by all compilers above it
 - Every new digital device with a VM implementation gains immediate access to an existing software base
- Simple compilers
- Managed code:
 - Security
 - Monitoring (BI, BPM)



Cons

- Performance

History

- P-code
- Java
- .Net