Virtual Machine (Part II)

Shimon Schocken

Spring 2005

Lecture plan

$$x = (-b + \sqrt{b^2 - 4 \cdot a \cdot c}) / 2a$$

```
if ~(a = 0)
   x = (-b + sqrt(power(b,2) - 4 * a * c)) / (2 * a)
else
   x = - c / b
```

In order to execute this code we have to know how to handle:

- Arithmetic operations (last lecture)
- Boolean operations (last lecture)
- Program flow (this lecture, easy)
- Subroutine calling (this lecture, requires some work ...)

Big point: All these abstractions can delivered at the VM level.

Copyright © Shimon Schocken

Program flow commands

- `label c`
- `goto c`
- `if-goto c`

Program flow

Flow of control structure	*VM pseudo code*

<pre>if (cond)
   s1
else
   s2
...</pre>
	<pre> VM code for computing ~(cond)
	  if-goto L1
	  VM code for executing s1
	  goto L2
	label L1
	  VM code for executing s2
	label L2
	  ...</pre>

<pre>while (cond)
   s1
...</pre>
	<pre>label L1
	  VM code for computing ~(cond)
	  if-goto L2
	  VM code for executing s1
	  goto L1
	label L2
	  ...</pre>

Copyright © Shimon Schocken

Subroutine calling: high level

```
if ~(a = 0)
   x = (-b + sqrt(power(b,2) – 4 * a * c)) / (2 * a)
else
   x = - c / b
```

The most important abstraction delivered by high level languages:

- The basic language can be extended at will by user-defined commands (= *subroutines* / *functions* / *methods* ...)
- The primitive commands and the user-defined commands have the same look-and-feel

"A well-deigned system consists of a collection of black box modules, each executing its effect like magic"
(Steven Pinker, *How The Mind Works*)

Opening up the black box

```
if ~(a = 0)
   x = (-b + sqrt(power(b,2) – 4 * a * c)) / (2 * a)
else
   x = - c / b
```

To handle the **`sqrt(power(b,2)-4*a*c))`** abstraction, someone has to:

- Pass arguments from **`sqrt`** to **`power`**
- Save the state of **`sqrt`** before switching to execute **`power`**
- Allocate space for the local variables of **`power`**
- Jump to execute **`power`**

When **`power`** terminates:

- Return a value from power to **`sqrt`**
- Recycle the memory space occupied by **`power`**
- Reinstate the state of **`sqrt`**
- Jump to execute the code of **`sqrt`** immediately after the spot where we left it.

Copyright © Shimon Schocken

Passing arguments and returning values

```
//x+2
push x
push 2
add
...
```

```
//x^3
push x
push 3
call power
...
```

```
//(x^3+2)^y
push x
push 3
call power
push 2
add
push y
call power
...
```

```
//Power function
//result = first arg
//raised to the power
//of the second arg.
function power
//code omitted
push result
return
```

Call-and-return convention (same for primitive commands and subroutines)

- The caller pushes arguments and calls the callee
- Before the callee terminates, it must push a return value

Net effect

- The arguments are replaced by the return value
- Delivered by the VM implementation
- The VM implementation manages everything away from the programmer's view, using the stack
- One of the most elegant gems in CS

The calling protocol (blue = unfinished magic business)

- `function f nVars`
- `call f nArgs`
- `return`

The caller function view:

- Before calling the function, I must push as many arguments as necessary onto the stack
- Next, I invoke the function using the `call` command
- After the called function returns, the arguments that I pushed before the call have disappeared from the stack, and a return value (that always exists) appears at the top of the stack
- After the called function returns, my `argument`, `local`, `static`, `this`, `that`, and `pointer` memory segments are the same as before the call.

The called function view:

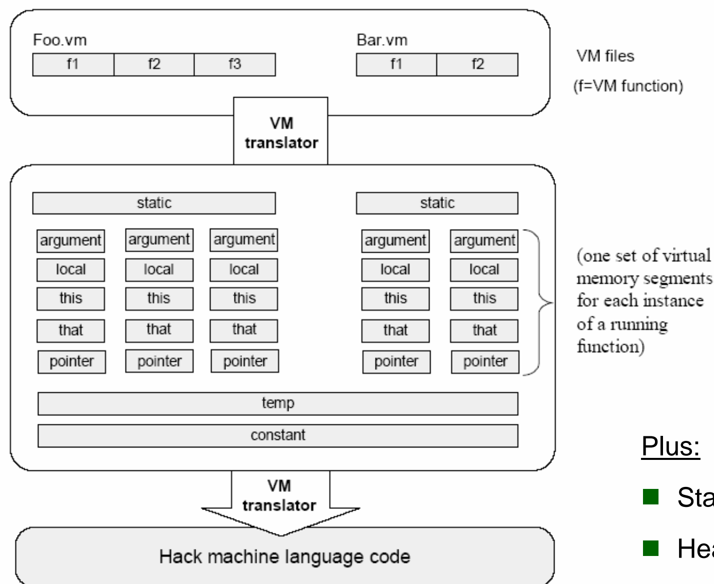
- When I start executing, my argument segment has been initialized with actual argument values passed by the caller
- My local variables segment has been allocated and initialized to zero
- The static segment that I see has been set to the static segment of the VM file to which I belong, and the working stack that I see is empty
- Before returning, I must push a value onto the stack.

Copyright © Shimon Schocken

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

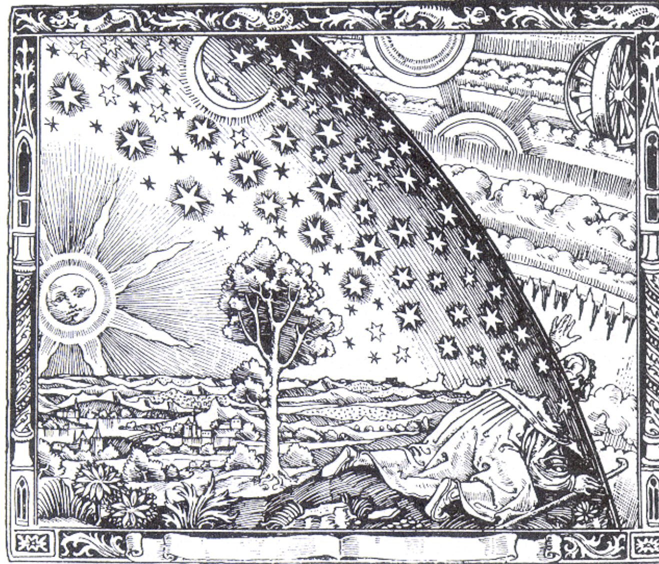


Plus:

- Stack
- Heap

Copyright © Shimon Schocken

Implementation



Implementation view of the calling protocol

- `function f nVars`
- `call f nArgs`
- `return`

When function *g* calls function *f*, I must:

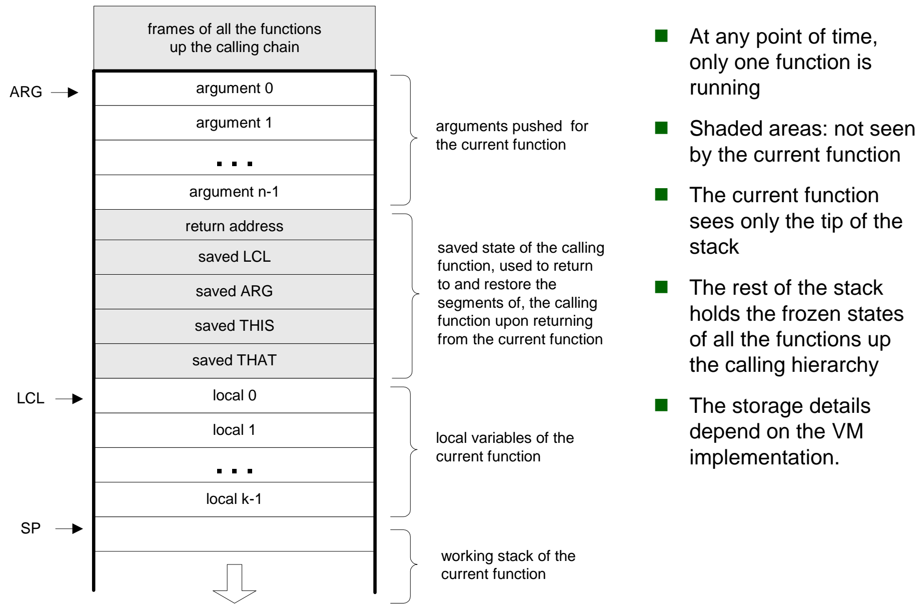
- Save the return address and the segment pointers of *g*
- Allocate, and initialize to 0, as many local variables as needed by *f*
- Set the local and argument segment pointers of *f*
- Transfer control to *f*.

When *f* terminates and control should return to *g*, I must:

- Clear the arguments and other junk from the stack
- Restore the segments of *g*
- Transfer control back to *g* (jump to the saved return address).

Copyright © Shimon Schocken

The global stack – the VM implementation's housekeeping memory



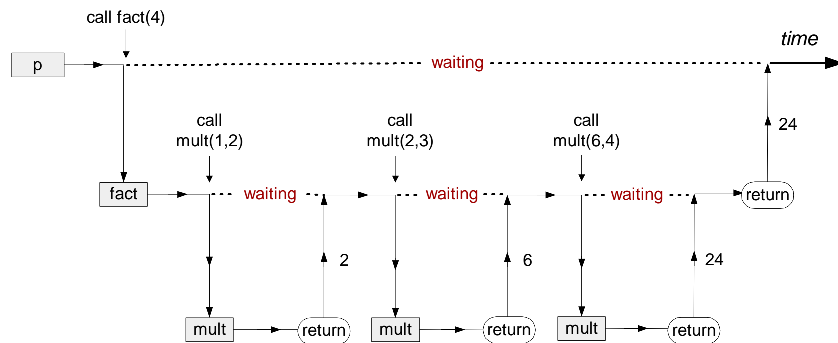
- At any point of time, only one function is running
- Shaded areas: not seen by the current function
- The current function sees only the tip of the stack
- The rest of the stack holds the frozen states of all the functions up the calling hierarchy
- The storage details depend on the VM implementation.

Example: a typical calling scenario

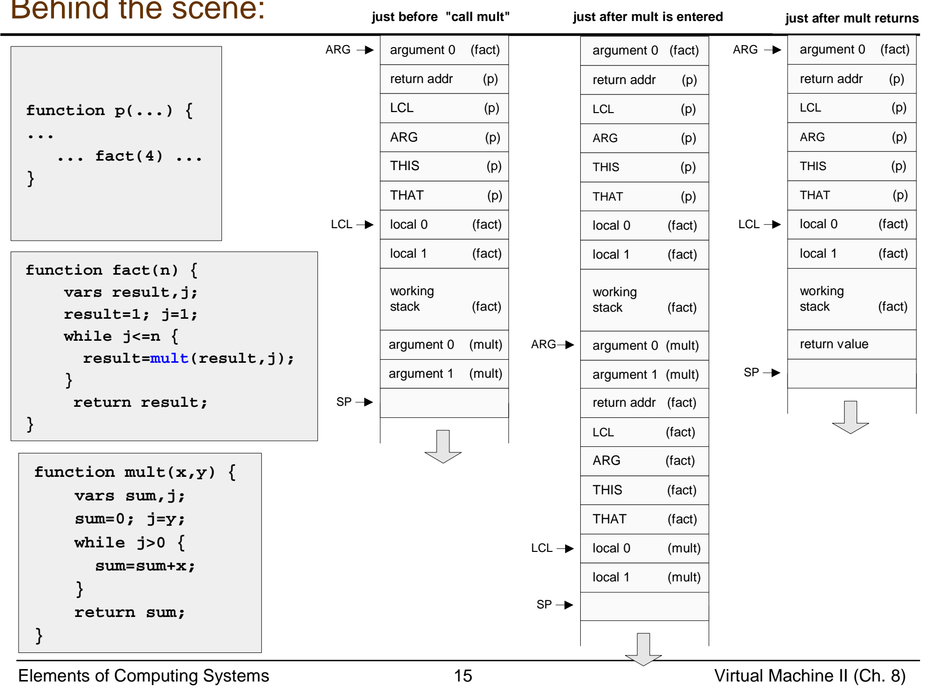
```
function p(...) {
...
   ... fact(4) ...
}
```

```
function fact(n) {
    vars result,j;
    result=1; j=1;
    while j<=n {
      result=mult(result,j);
    }
     return result;
}
```

```
function mult(x,y) {
    vars sum,j;
    sum=0; j=y;
    while j>0 {
      sum=sum+x;
    }
    return sum;
}
```



Copyright © Shimon Schocken

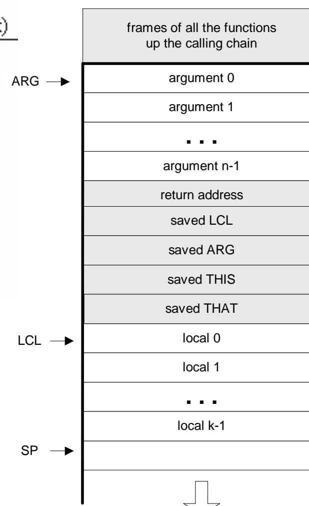


Implementing the `call f n` command

```
call f n
```

(calling a function `f` after `n` arguments have been pushed onto the stack)

```
push return-address   // (Using the label declared below)
push LCL              // Save LCL of the calling function
push ARG              // Save ARG of the calling function
push THIS             // Save THIS of the calling function
push THAT             // Save THAT of the calling function
ARG = SP-n-5          // Reposition ARG (n = number of args)
LCL = SP              // Reposition LCL
goto f                // Transfer control
(return-address)      // Declare a label for the return-address
```



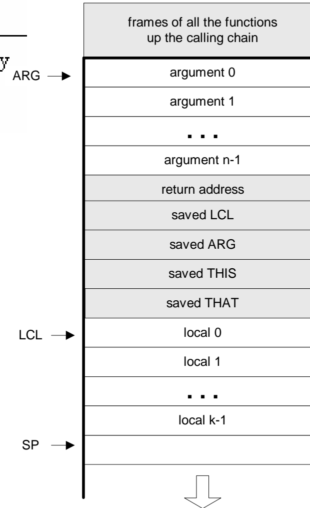
Copyright © Shimon Schocken

Implementing the `function f k` command

function f k

(declaring a function f that has k local variables)

```
(f)                    // Declare a label for the function entry
    repeat k times:    // k = number of local variables
    PUSH 0             // Initialize all of them to 0
```

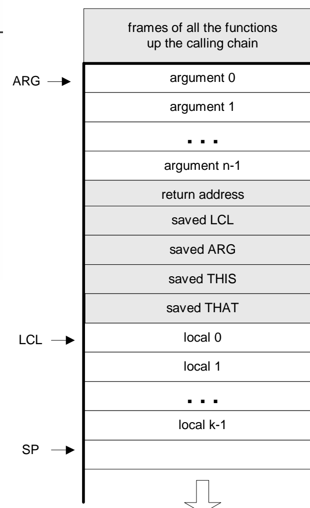


Implementing the `return` command

return

(from a function)

```
FRAME=LCL           // FRAME is a temporary variable
RET=*(FRAME-5)      // Put the return-address in a temp. variable
*ARG=pop()          // Reposition the return value for the caller
SP=ARG+1            // Restore SP of the caller
THAT=*(FRAME-1)     // Restore THAT of the caller
THIS=*(FRAME-2)     // Restore THIS of the caller
ARG=*(FRAME-3)      // Restore ARG  of the caller
LCL=*(FRAME-4)      // Restore LCL of the caller
goto RET            // Goto return-address (in the caller's code)
```



Copyright © Shimon Schocken

Bootstrap code

```
SP = 256          // initialize the stack pointer to 0x0100
Call Sys.init     // an initialization function
```

- `Sys.init` should `call f` and then enter an infinite loop

In the Hack/Jack platform:

- `Sys.init` is part of the OS
- `f` is `Main.main`

VM implementation over the Hack platform: special symbols

Symbol	*Usage*
SP, LCL, ARG, THIS, THAT	These predefined symbols point, respectively, to the stack top and to the base addresses of the virtual segments local, argument, this, and that.
R13 - R15	These predefined symbols can be used for any purpose.
Xxx.j	Each static variable j in a VM file Xxx.vm is translated into the assembly symbol Xxx.j. In the subsequent assembly process, these symbolic variables will be allocated RAM space by the Hack assembler.
functionName$label	Each label b command in a VM function f should generate a globally unique symbol "f$b" where "f" is the function name and "b" is the label symbol within the VM function's code. When translating goto b and if-goto b VM commands into the target language, the full label specification "f$b" must be used instead of "b".
(FunctionName)	Each VM function f should generates a symbol "f" that refers to its entry point in the instruction memory of the target computer.
return-address	Each VM function call should generate and insert into the translated code a unique symbol that serves as a return address, namely the memory location (in the target platform's memory) of the command following the function call.

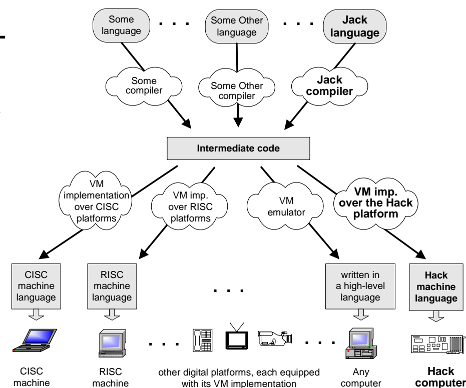
Copyright © Shimon Schocken

Proposed API

CodeWriter: Translates VM commands into Hack assembly code. The routines listed here should be added to the `CodeWriter` module API given in chapter 7.

Routine	Arguments	Returns	Function
`writeInit`	--	--	Writes the assembly code that effects the VM initialization, also called *bootstrap code*. This code must be placed at the beginning of the output file.
`writeLabel`	`label` (string)	--	Writes the assembly code that is the translation of the `label` command.
`writeGoto`	`label` (string)	--	Writes the assembly code that is the translation of the `goto` command.
`writeIf`	`label` (string)	--	Writes the assembly code that is the translation of the `if-goto` command.
`writeCall`	`functionName` (string) `numArgs` (int)	--	Writes the assembly code that is the translation of the `call` command.
`writeReturn`	--	--	Writes the assembly code that is the translation of the `return` command.
`writeFunction`	`functionName` (string) `numLocals` (int)	--	Writes the assembly code that is the trans. of the given `function` command.

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

Copyright © Shimon Schocken