Chapter 8:
Virtual Machine II:
Program Control

Usage and Copyright Notice:

Copyright 2005 © Noam Nisan and Shimon Schocken


The book web site, www.idc.ac.il/tecs, features 13 such presentations, one for each book chapter. Each presentation is designed to support about 3 hours of classroom or self-study instruction.

You are welcome to use or edit this presentation for instructional and non-commercial purposes.

If you use our materials, we will appreciate it if you will include in them a reference to the book’s web site.

And, if you have any comments, you can reach us at tecs.ta@gmail.com
Virtual Machine

Part II: Program Control
Where we are at:

- **Abstract design**
  - Chapters 9, 12

- **Human Thought**

- **H.L. Language & Operating Sys.**
  - Chapters 10 - 11
  - **Compiler**

- **Virtual Machine**
  - Chapters 7 - 8

- **VM Translator**
  - **Assembly Language**

- **Assembler**
  - Chapter 6

- **Computer Architecture**
  - Chapters 4 - 5

- **Machine Language**

- **Hardware Platform**
  - Chapters 1 - 3

- **Chips & Logic Gates**

- **Gate Logic**

- **Electrical Engineering**

- **Physics**

**Hardware hierarchy**

**Software hierarchy**

The big picture

- Some language
- Some Other language
- Jack language

- Some compiler
- Some Other compiler
- Jack compiler

- VM language

- VM implementation over CISC platforms
- VM imp. over RISC platforms
- VM imp. over the Hack platform

- CISC machine language
- RISC machine language
- Hack machine language

- CISC machine
- RISC machine
- Hack computer

A Java-based emulator is included in the course software suite.

Implemented in Projects 7-8

Chapters 1-6
Chapters 7-8
Chapters 9-13
Lecture plan

**Goal:** Specify and implement a VM model and language

<table>
<thead>
<tr>
<th>Arithmetic / Boolean commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
</tr>
<tr>
<td>sub</td>
</tr>
<tr>
<td>neg</td>
</tr>
<tr>
<td>eq</td>
</tr>
<tr>
<td>gt</td>
</tr>
<tr>
<td>lt</td>
</tr>
<tr>
<td>and</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>not</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory access commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop segment i</td>
</tr>
<tr>
<td>push segment i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program flow commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>label (declaration)</td>
</tr>
<tr>
<td>goto (label)</td>
</tr>
<tr>
<td>if-goto (label)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function calling commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>function (declaration)</td>
</tr>
<tr>
<td>call (a function)</td>
</tr>
<tr>
<td>return (from a function)</td>
</tr>
</tbody>
</table>

**Method:** (a) specify the abstraction (model’s constructs and commands)  
(b) propose how to implement it over the Hack platform.
**Jack source code (example):**

```java
class Foo {
    static int x1, x2, x3;
    method int f1(int x) {
        var int a, b;
        ...
    }
    method void f2(int x, int y) {
        var int a, b, c;
        ...
    }
    function int f3(int u) {
        var int x;
        ...
    }
}

class Bar {
    static int y1, y2;
    function void f1(int u, int v) {
        ...
    }
    method void f2(int x) {
        var int a1, a2;
        ...
    }
}
```

**Following compilation:**

- **VM files**
  - `Foo.vm`
    - `f1` function
    - `f2` function
    - `f3` function
  - `Bar.vm`
    - `f1` function
    - `f2` function

- **VM translator**
  - `static` segment
    - `argument`
    - `local`
    - `this`
    - `that`
    - `pointer`
    - `temp`
    - `constant`

- **Hack machine language code**

- **Compiler**

---

The challenge ahead

\[ x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \]

In order to enable such high-level code we have to know how to handle:

- Arithmetic operations  (previous lecture)
- Boolean operations  (previous lecture)
- Program flow  (this lecture, easy)
- Subroutines  (this lecture, medium/rare)

**In the Jack/Hack platform:** all these abstractions are delivered by the VM level.
Program flow

- label \( c \)
- goto \( c \)
- if-goto \( c \)
  (pop the topmost element from the stack. If it’s not zero, jump)

Implementation (by translation to assembly):

Simple. label declarations and goto directives can be effected directly by assembly commands.
Subroutines

Subroutines = a major programming artifact

- The primitive (given) language can be extended at will by user-defined commands (AKA subroutines / functions / methods ...)

- The primitive commands and the user-defined commands have the same look-and-feel

- Perhaps the most important abstraction delivered by programming languages. The challenge: to make the implementation of this abstraction as transparent as possible:

  “A well-deigned system consists of a collection of black box modules, each executing its effect like magic”
  (Steven Pinker, How The Mind Works)
Subroutines usage at the VM level (pseudo code)

Call-and-return convention

- The caller pushes the arguments, calls the callee, then waits for it to return
- Before the callee terminates (returns), it must push a return value
- At the point of return, the callee’s resources are recycled, and the caller’s state is re-instated

**Caller’s net effect:** the arguments were replaced by the return value (just like with primitive operations)

Behind the scene

- Recycling and re-instating subroutine resources and states is a major headache
- The VM implementation should manage it “like magic”
- The magic is stack-based, and is considered a great CS gem.
Subroutine commands

- **function g nVars**
  (Here starts a function called $g$, which has $nVars$ local variables)

- **call g nArgs**
  (Invoke function $g$ for its effect; $nArgs$ arguments have been pushed onto the stack)

- **Return**
  (Terminate execution and return control to the calling function)

**Implementation:** Next few slides.
Aside: The VM emulator (Java-based, included in the course software suite)
The calling protocol

The caller's 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.
  - All my memory segments (argument, local, static, ...) are the same as before the call.

The callee's 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.

Blue = function writer's responsibility
Black = black box magic, supplied by the VM implementation.
VM implementation view of the calling protocol

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

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

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

- Clear the arguments and other junk from the stack
- Restore the segments of $f$
- Transfer control back to $f$
  (jump to the saved return address).
The VM implementation housekeeping storage = the stack

- Remember: at any typical point of time, some functions are waiting, and only the current function is running.
- Shaded areas: irrelevant to the current function.
- The current function sees only the top of the stack (AKA working stack).
- The rest of the stack holds the frozen states of all the functions up the calling hierarchy.
- Physical 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;
}
Behind the scene:

```javascript
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;
}
```
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

---

frames of all the functions up the calling chain

ARG →

- argument 0
- argument 1
- . . .
- argument n-1
- return address
- saved LCL
- saved ARG
- saved THIS
- saved THAT

LCL →

- local 0
- local 1
- . . .
- local k-1

SP →

---
Implementing the \textbf{function} $f_k$ \textbf{command}

\begin{Verbatim}
\textbf{function} $f_k$
(declaring a function $f$ that has $k$ local variables)

(\textbf{f}) \hfill // Declare a label for the function entry
repeat $k$ times: \hfill // $k =$ number of local variables
  \textbf{PUSH} 0 \hfill // Initialize all of them to 0
\end{Verbatim}
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)
```
One more detail: bootstrapping

- A high-level Jack program (AKA application) is a set of class files. By convention, one class must be called `Main`, and this class must have at least one function called `main`. The contract is such that when we tell the computer to execute the program, the function `Main.main` starts running.

**Implementation:**

- After the program is compiled, each class file is translated into a `.vm` file.
- From the host platform’s standpoint, the operating system is also a set of `.vm` files (AKA “libraries”) that co-exist alongside the user’s `.vm` files.
- One of the OS libraries is called `Sys.vm`, which includes a function called `init`. This function starts with some OS initialization code (explained in Ch. 12), then it does `call f` and enters an infinite loop; if the code originates from Jack, `f` is `Main.main`.
- Thus, to bootstrap, the VM implementation has to effect (e.g. in assembly), the following operations:

  ```
  SP = 256         // initialize the stack pointer to 0x0100
  call Sys.init    // the initialization function
  ```
VM implementation over the Hack platform

- Extends the VM implementation proposed in the last lecture (Chapter 7)
- The result: a big assembly program with lots of agreed-upon symbols:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP, LCL, ARG, THIS, THAT</td>
<td>These predefined symbols point, respectively, to the stack top and to the base addresses of the virtual segments local, argument, this, and that.</td>
</tr>
<tr>
<td>R13 - R15</td>
<td>These predefined symbols can be used for any purpose.</td>
</tr>
<tr>
<td>Xxx.j</td>
<td>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.</td>
</tr>
<tr>
<td>functionName$label</td>
<td>Each label b command in a VM function f should generate a globally unique symbol “f$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”.</td>
</tr>
<tr>
<td>(FunctionName)</td>
<td>Each VM function f should generate a symbol “f” that refers to its entry point in the instruction memory of the target computer.</td>
</tr>
<tr>
<td>return-address</td>
<td>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.</td>
</tr>
</tbody>
</table>
Proposed API

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeInit</td>
<td>--</td>
<td>--</td>
<td>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.</td>
</tr>
<tr>
<td>writeLabel</td>
<td>label (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the label command.</td>
</tr>
<tr>
<td>writeGoto</td>
<td>label (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the goto command.</td>
</tr>
<tr>
<td>writeIf</td>
<td>label (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the if-goto command.</td>
</tr>
<tr>
<td>writeCall</td>
<td>functionName (string)</td>
<td>numArgs (int)</td>
<td>--</td>
</tr>
<tr>
<td>writeReturn</td>
<td>--</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the return command.</td>
</tr>
<tr>
<td>writeFunction</td>
<td>functionName (string)</td>
<td>numLocals (int)</td>
<td>--</td>
</tr>
</tbody>
</table>
Perspective

Benefits of the VM approach

- 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
- Code mobility: Internet
- 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
  - New programming languages can be implemented easily using simple compilers

Benefits of managed code:

- Security
- Array bounds, index checking, ...
- Add-on code
- Etc.

VM Cons

- Performance.