Virtual Machine
Part II: Program Control

Usage and Copyright Notice:

Copyright 2005 © Noam Nisan and Shimon Schocken


We provide both PPT and PDF versions.

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 as you see fit 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.

If you have any questions or comments, you can reach us at tecs.ta@gmail.com
Where we are at:

**Abstract design**
- Chapters 9, 12

**Software hierarchy**
- Compiler (Chapters 10 - 11)
- VM Translator (Chapters 7 - 8)
- Assembly Language

**Hardware hierarchy**
- Machine Language (Chapters 4 - 5)
- Hardware Platform (Chapters 1 - 3)
- Chips & Logic Gates
- Electrical Engineering (Physics)

**Human Thought**
- Chapter 6
The big picture

- VM language
  - VM implementation over CISC platforms
  - VM implementation over RISC platforms
  - VM emulator
    - A Java-based emulator is included in the course software suite
    - Implemented in Projects 7-8
  - VM imp. over the Hack platform
    - Written in a high-level language
    - Hack machine language
  - Some language
    - Some compiler
    - Some Other language
    - Some Other compiler
    - Jack language
      - Jack compiler
- CISC machine language
  - CISC machine
- RISC machine language
  - RISC machine
- Other digital platforms, each equipped with its VM implementation
- Any computer

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

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

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

**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)
Program structure and translation path (on the Hack-Jack 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;
        ...
    }
}
```

Jack source code:

```java
class Foo {
    static staticsList;
    method f1(argsList) {
        var localsList;
        ...
    }
    method f2(argsList) {
        var localsList;
        ...
    }
    function f3(argsList) {
        var localsList;
        ...
    }
}

class Bar {
    static staticsList;
    function f1(argsList) {
        ...
    }
    method f2(argsList) {
        var localsList;
        ...
    }
}
```

In general
Program structure and translation path (on the Hack-Jack platform)

**Jack source code:**

```java
class Foo {
    static staticsList;
    method f1(argsList) {
        var localsList;
        ...}
    method f2(argsList) {
        var localsList;
        ...}
    function f3(argsList) {
        var localsList;
        ...}
}

class Bar {
    static staticsList;
    function f1(argsList) {
        ...}
    method f2(argsList) {
        var localsList;
        ...}
}
```

**Following compilation:**

- **Foo.vm**
  - f1
  - f2
  - f3

- **Bar.vm**
  - f1
  - f2

**VM files**

- **VM translator**
  - static
    - argument
    - local
    - this
    - that
    - pointer
  - argument
  - local
  - this
  - that
  - pointer

- **VM translator**
  - argument
  - local
  - this
  - that
  - pointer

- **Hack machine language code**
  - temp
  - constant

- **One file**

*(one set of virtual segments for each instance of a running function)*

Elements of Computing Systems, Nisan & Schocken, MIT Press, [www.idc.ac.il/tecs](http://www.idc.ac.il/tecs), Chapter 8: *Virtual Machine, Part II*
The challenge ahead

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

To translate such high-level code to VM code, we have to know how to handle:

- **Arithmetic operations** (last lecture)
- **Boolean operations** (last lecture)
- **Program flow** (this lecture, *easy*)
- **Subroutines** (this lecture, *less easy*)

**In the Jack/Hack platform:** all these abstractions are delivered by the VM level (rather than by the compiler).
Program flow

- label c
- goto c
- if-goto c  // pop the topmost stack element;
  // If it's not zero, jump

Example:

```plaintext
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
```

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-designed 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), the callee 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
- Some behind-the-scene agent (the VM or the compiler) should manage it “like magic”
- In our implementation, 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 function-call-and-return protocol

**The caller’s view:**

- Before calling the function, I must push as many arguments as needed 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 exiting the function, I must push a value onto the stack and then return.

**Function**  
function g nVars  
call g nArgs  
return

Blue = function writer’s responsibility

Black = black box magic, supplied by the VM implementation

In other words, we have to worry about the “black operations” only.
VM implementation view of the function-call-and-return protocol

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

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

- At any 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.

- 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
  - arguments pushed for the current function
  - saved state of the calling function, used to return to and restore the segments of, the calling function upon returning from the current function
  - local variables of the current function
  - working stack of the current function

At any 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);
        j=j+1;
    }
    return result;
}

function mult(x, y) {
    vars sum, j;
    sum=0; j=y;
    while j>0 {
        sum=sum+x;
        j=j+1;
    }
    return sum;
}
```

```
call fact(4)

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

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

function mult(x, y) {
    vars sum, j;
    sum=0; j=y;
    while j>0 {
        sum=sum+x;
        j=j+1;
    }
    return sum;
}
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```

```
call fact(4)
```
Behind the scene:

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

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

```
function mult(x, y) {
    vars sum, j;
    sum = 0; j = y;
    while j > 0 {
        sum = sum + x;
        j = j + 1;
    }
    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

If the VM is implemented as a program that translates VM code to assembly code, the translator should generate the above logic in assembly.
Implementing the **function f_k** command

**function f_k**

(declaring a function f that has k local variables)

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

- If the VM is implemented as a program that translates VM code to assembly code, the translator should generate the above logic in assembly.

---

If the VM is implemented as a program that translates VM code to assembly code, the translator should generate the above logic in assembly.
Implementing the `return` command

(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)

If the VM is implemented as a program that translates VM code to assembly code, the translator should generate the above logic in assembly.
One more detail: bootstrapping

- A high-level jack program (AKA application) is a set of class files. By a Jack convention, one class must be called Main, and this class must have at least one function, called main. The contract: 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, which includes a method called init. The Sys.init function starts with some OS initialization code (we’ll deal with this later, when we discuss the OS), then it does call f and enters an infinite loop; If the application was written in the Jack language, then by convention call f should be call Main.main.
- Thus, to bootstrap, the VM implementation has to effect (e.g. in assembly), the following operations:

```plaintext
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$label</td>
<td>Each label ( b ) command in a VM function ( f ) should generate a globally unique symbol ( &quot;f$\text{label}&quot; ) where ( &quot;f&quot; ) is the function name and ( &quot;\text{label}&quot; ) is the label symbol within the VM function’s code. When translating \text{goto} ( b ) and \text{if-goto} ( b ) VM commands into the target language, the full label specification ( &quot;f$\text{label}&quot; ) must be used instead of ( &quot;\text{label}&quot; ).</td>
</tr>
<tr>
<td>(FunctionName)</td>
<td>Each VM function ( f ) should generate a symbol ( &quot;f&quot; ) 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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>writeInit</code></td>
<td>--</td>
<td>--</td>
<td>Writes the assembly code that affects the VM initialization, also called bootstrap code. This code must be placed at the beginning of the output file.</td>
</tr>
<tr>
<td><code>writeLabel</code></td>
<td>label (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>label</code> command.</td>
</tr>
<tr>
<td><code>writeGoto</code></td>
<td>label (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>goto</code> command.</td>
</tr>
<tr>
<td><code>writeIf</code></td>
<td>label (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>if-goto</code> command.</td>
</tr>
<tr>
<td><code>writeCall</code></td>
<td>functionName (string) numArgs (int)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>call</code> command.</td>
</tr>
<tr>
<td><code>writeReturn</code></td>
<td>--</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>return</code> command.</td>
</tr>
<tr>
<td><code>writeFunction</code></td>
<td>functionName (string) numLocals (int)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the given function command.</td>
</tr>
</tbody>
</table>
Perspective

Benefits of the VM approach

- Code transportability: compiling for different platforms requires 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.