Virtual Machine
Part I: Stack Arithmetic

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

Human Thought

Hardware hierarchy

Machine Language
Chapters 4 - 5

Computer Architecture
Chapter 6

Compiler
Chapters 10 - 11

Virtual Machine
Chapters 7 - 8

VM Translator

Software hierarchy

Assembly Language

H.L. Language & Operating Sys.

Compiler

Assembler
Chapter 6

Assembly

H.L. Language & Operating Sys.

Abstract design

Compiler

Virtual Machine

Assembly Language

Chips & Logic Gates

Electrical Engineering

Physics

Hardware Platform
Chapters 1 - 3

Gate Logic
Motivation

class Main {
    static int x;

    function void main() {
        // Input and multiply 2 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 not(j = 0) {
        let result = result + x;
        let j = j – 1;
    }
    return result;
}

Ultimate goal:
Translate high-level programs into executable code.

Compiler
Compilation models

**direct compilation:**

Language 1 \rightarrow \text{hardware platform 1} \rightarrow \text{hardware platform 2} \rightarrow \cdots \rightarrow \text{hardware platform m} \rightarrow \text{language n}

Requires $n \times m$ translators

**2-tier compilation:**

Language 1 \rightarrow \text{intermediate language} \rightarrow \text{hardware platform 1} \rightarrow \text{hardware platform 2} \rightarrow \cdots \rightarrow \text{hardware platform m} \rightarrow \text{language n}

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 platform.
The big picture

Some language → Some compiler → Intermediate code
Some Other language → Some Other compiler → Intermediate code
Jack language → Jack compiler → Intermediate code

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 many different ways.

Some Other language → Some Other compiler → Intermediate code
Jack language → Jack compiler → Intermediate code

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

CISC machine language
RISC machine language
Any computer
Hack machine language

CISC machine
RISC machine
other digital platforms, each equipped with its own VM implementation

The big picture

- VM languages:
  - VM over CISC platforms
  - VM over RISC platforms
  - VM over the Hack platform

- VM implementations:
  - Over CISC platforms
  - Over RISC platforms
  - Over the Hack platform

- Languages:
  - Some language
  - Some Other language
  - Jack language

- Compilers:
  - Some compiler
  - Some Other compiler
  - Jack compiler

- Machine languages:
  - CISC machine language
  - RISC machine language
  - Hack machine language

- Platforms:
  - CISC machine
  - RISC machine
  - Hack computer
  - Other digital platforms, each equipped with its VM implementation
  - Any computer

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

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

### Arithmetic / Boolean commands
- `add`
- `sub`
- `neg`
- `eq`
- `gt`
- `lt`
- `and`
- `or`
- `not`

### Memory access commands
- `pop x` (variable)
- `push y` (variable or constant)

### Program flow commands
- `label` (declaration)
- `goto` (label)
- `if-goto` (label)

### Function calling commands
- `function` (declaration)
- `call` (a function)
- `return` (from a function)

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

**Important:**

From here till the end of this and the next lecture we describe the VM model used in the Hack-Jack platform.

Other VM models (like JVM/JRE and IL/CLR) are similar in spirit and different in scope and details.

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

- Integer
- Boolean
- Pointer.
Stack arithmetic

- Typical operation:
  - Pops topmost values \( x, y \) from the stack
  - Computes the value of some function \( f(x, y) \)
  - Pushes the result onto the stack

(Unary operations are similar, using \( x \) and \( f(x) \) instead)

- Impact: the operands are replaced with the operation's result
- In general: all arithmetic and Boolean operations are implemented similarly.
Memory access (first approximation)

Stack

<table>
<thead>
<tr>
<th>121</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>17</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>121</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>17</td>
</tr>
</tbody>
</table>

Stack

<table>
<thead>
<tr>
<th>121</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>17</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>121</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>108</td>
</tr>
</tbody>
</table>

push b

Stack

<table>
<thead>
<tr>
<th>121</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>108</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>121</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>108</td>
</tr>
</tbody>
</table>

(before)

(after)
Memory access (first approximation)

- Classical data structure
- Elegant and powerful
- Many implementation options.
Evaluation of arithmetic expressions

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

Memory
x
5
y
9
...

Stack
SP→
push 2
SP→
2
push x
SP→
2
5
sub
push y
SP→
-3
push 5
SP→
-3
9
add
mult
SP→
-3
14
pop d
SP→
-42
Memory
x
5
y
9
...
d
-42
***
Evaluation of Boolean expressions

// if (x<7) or (y=8)
push x
push 7
lt
push y
push 8
eq
or
# Arithmetic and Boolean commands (wrap-up)

<table>
<thead>
<tr>
<th>Command</th>
<th>Return value (after popping the operand/s)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$x + y$</td>
<td>Integer addition</td>
</tr>
<tr>
<td>sub</td>
<td>$x - y$</td>
<td>Integer subtraction</td>
</tr>
<tr>
<td>neg</td>
<td>$-y$</td>
<td>Arithmetic negation</td>
</tr>
<tr>
<td>eq</td>
<td>true if $x = y$ and false otherwise</td>
<td>Equality</td>
</tr>
<tr>
<td>gt</td>
<td>true if $x &gt; y$ and false otherwise</td>
<td>Greater than</td>
</tr>
<tr>
<td>lt</td>
<td>true if $x &lt; y$ and false otherwise</td>
<td>Less than</td>
</tr>
<tr>
<td>and</td>
<td>$x \text{ And } y$</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>or</td>
<td>$x \text{ Or } y$</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>not</td>
<td>Not $y$</td>
<td>Bit-wise</td>
</tr>
</tbody>
</table>

- **Integer addition**: (2's complement)
- **Integer subtraction**: (2's complement)
- **Arithmetic negation**: (2's complement)
Memory access (motivation)

Modern programming languages normally feature the following variable kinds:

- **Class level**
  - Static variables
  - Private variables (AKA “object variables” / “fields” / “properties”)

- **Method level**:
  - Local variables
  - Argument variables

A VM abstraction must support (at least) all these variable kinds.

The memory of our VM model consists of 8 memory segments:

  static, argument, local, this, that, constant, pointer, and temp.
Memory access commands

**Command format:**

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

(Rather than `pop x` and `push y`, as was shown in previous slides, which was a conceptual simplification)

Where *i* is a non-negative integer and *segment* is one of the following:

- **static**: holds values of global variables, shared by all functions in the same class
- **argument**: holds values of the argument variables of the current function
- **local**: holds values of the local variables of the current function
- **this**: holds values of the private ("object") variables of the current object
- **that**: holds array values
- **constant**: holds all the constants in the range 0...32767 (pseudo memory segment)
- **pointer**: used to align `this` and `that` with different areas in the heap
- **temp**: fixed 8-entry segment that holds temporary variables for general use; Shared by all VM functions in the program.
VM programming

- VM programs are normally written by *compilers*, not by humans
- In order to write compilers, it helps to understand the spirit of VM programming. So we will now see how some common programming tasks can be implemented in the VM abstraction:
  - Arithmetic task
  - Object handling task
  - Array handling task

Disclaimer:

These programming examples don't belong here; They belong to the compiler chapter, since expressing programming tasks in the VM language is the business of the compiler (e.g., translating Java programs to Bytecode programs)

We discuss them here to give some flavor of programming at the VM level.

(One can safely skip from here to slide 21)
### High-level code

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

### VM code (first approx.)

```c
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

```c
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
```
/* 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:

<table>
<thead>
<tr>
<th>argument</th>
<th>pointer</th>
<th>this</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 3012</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 17</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>argument</th>
<th>pointer</th>
<th>this</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 3012</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 17</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

3012 120

3013 80

3014 50

3015 3

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

(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

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

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

<table>
<thead>
<tr>
<th>Arithmetic / Boolean commands</th>
<th>Program flow commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>label</td>
</tr>
<tr>
<td>sub</td>
<td>goto</td>
</tr>
<tr>
<td>neg</td>
<td>if-goto</td>
</tr>
<tr>
<td>eq</td>
<td></td>
</tr>
<tr>
<td>gt</td>
<td></td>
</tr>
<tr>
<td>lt</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td></td>
</tr>
</tbody>
</table>

**Memory access commands**

| pop segment i | push segment i |

**Function calling commands**

<table>
<thead>
<tr>
<th>function</th>
<th>call</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a function)</td>
<td>(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.
Implementation

VM implementation options:

- Software-based (emulation)
- Translator-based (e.g., to the Hack language)
- Hardware-based (CPU-level)

Well-known translator-based implementations:

- JVM (runs bytecode programs in the Java platform)
- CLR (runs IL programs in the .NET platform)
Our VM emulator (part of the course software suite)
VM implementation on the Hack platform

The challenge: (i) map the VM constructs on the host RAM, and (ii) given this mapping, figure out how to implement each VM command using assembly commands that operate on the RAM.

- **local, argument, this, that**: mapped on the heap. The base addresses of these segments are kept in LCL, ARG, THIS, THAT. Access to the \(i\)-th entry of a segment is implemented by accessing the segment's (base + \(i\)) word in the RAM.

- **static**: static variable number \(j\) in a VM file \(f\) is implemented by the assembly language symbol \(f.j\) (and recall that the assembler maps such symbols to the RAM starting from address 16).

- **constant**: truly a virtual segment. Access to constant \(i\) is implemented by supplying the constant \(i\).

- **pointer, temp**: see the book.

Exercise: given the above game rules, write the Hack commands that implement, say, push constant 5 and pop local 2.
## 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.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Input file / stream</td>
<td>--</td>
<td>Opens the input file/stream and gets ready to parse it.</td>
</tr>
<tr>
<td>hasMoreCommands</td>
<td>--</td>
<td>boolean</td>
<td>Are there more commands in the input?</td>
</tr>
<tr>
<td>advance</td>
<td>--</td>
<td>--</td>
<td>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.</td>
</tr>
<tr>
<td>commandType</td>
<td>--</td>
<td>C_ARITHMETIC, C_PUSH, C_POP, C_LABEL, C_GOTO, C_IF, C_FUNCTION, C_RETURN, C_CALL</td>
<td>Returns the type of the current VM command. C_ARITHMETIC is returned for all the arithmetic commands.</td>
</tr>
<tr>
<td>arg1</td>
<td>--</td>
<td>string</td>
<td>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.</td>
</tr>
<tr>
<td>arg2</td>
<td>--</td>
<td>int</td>
<td>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.</td>
</tr>
</tbody>
</table>
**CodeWriter module** (proposed design)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Output file / stream</td>
<td>--</td>
<td>Opens the output file/stream and gets ready to write into it.</td>
</tr>
<tr>
<td>setFileName</td>
<td>fileName (string)</td>
<td>--</td>
<td>Informs the code writer that the translation of a new VM file is started.</td>
</tr>
<tr>
<td>writeArithmetic</td>
<td>command (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the given arithmetic command.</td>
</tr>
<tr>
<td>WritePushPop</td>
<td>Command (C_PUSH or C_POP), segment (string), index (int)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the given command, where command is either C_PUSH or C_POP.</td>
</tr>
<tr>
<td>Close</td>
<td>--</td>
<td>--</td>
<td>Closes the output file.</td>
</tr>
</tbody>
</table>

**Comment:** More routines will be added to this module in chapter 8.
In this lecture we began the process of building a compiler.

Modern compiler architecture:
- Front end (translates from high level language to a VM language)
- Back end (implements the VM language on a target 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 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 road ahead

<table>
<thead>
<tr>
<th>Tasks:</th>
<th>Conceptually similar to:</th>
<th>And to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete the VM specification and implementation (chapters 7,8)</td>
<td>JVM</td>
<td>CLR</td>
</tr>
<tr>
<td>Introduce Jack, a high-level programming language (chapter 9)</td>
<td>Java</td>
<td>C#</td>
</tr>
<tr>
<td>Build a compiler for it (chapters 10,11)</td>
<td>Java compiler</td>
<td>C# compiler</td>
</tr>
<tr>
<td>Finally, build a mini-OS, i.e. a run-time library (chapter 12).</td>
<td>JRE</td>
<td>.NET base class library</td>
</tr>
</tbody>
</table>