Chapter 7: Virtual Machine I: Stack Arithmetic

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
Where we are at:

Abstract design
- Human Thought
  - Chapters 9, 12

Software hierarchy
- Compiler
    - Chapters 10 - 11

Hardware hierarchy
- Assembler
  - Machine Language
    - Chapters 4 - 5
- Virtual Machine
  - Abstract interface
- VM Translator
  - Assembly Language
  - Abstract interface

- Computer Architecture
  - Abstract interface
  - Chapters 4 - 5

- Hardware Platform
  - Gate Logic
    - Abstract interface
    - Chapters 1 - 3
  - Chips & Logic Gates
    - Abstract interface
- Electrical Engineering
  - Physics


slide 2
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:
Translate high-level programs into executable code.
Compilation models

**direct compilation:**

- Language 1
- Language 2
- ... Language n

- Hardware platform 1
- Hardware platform 2
- ... Hardware platform m

Requires $n \cdot m$ translators

**2-tier compilation:**

- Language 1
- Language 2
- ... Language n

- Intermediate language

- Hardware platform 1
- Hardware platform 2
- ... Hardware platform m

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

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.

The interface between the two compilation stages must be sufficiently general to support many pairs of high-level languages and machine languages. It can be modeled as the language of an abstract virtual machine (VM) and implemented in many different ways.
The big picture

Some language
Some compiler
VM language
VM implementation over CISC platforms
CISC machine

Some Other language
Some Other compiler

Jack language
Jack compiler
VM emulator
VM imp. over the Hack platform
Hack machine language
Hack computer

Other digital platforms, each equipped with its VM implementation

Written in a high-level language

Any computer

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 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)

**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 \( f(x, y) \)
  - 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)

Memory (before):

- Stack:
  - 121
  - 5
  - 17
- Memory:
  - a: 6
  - b: 108

SP →

Memory (after):

- Stack:
  - 121
  - 5
  - 17
  - 108
- Memory:
  - a: 6
  - b: 108

SP →

push b

---

Memory access (first approximation)

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

// \( d = (2 - x) \times (y + 5) \)
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
### 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 (2's complement)</td>
</tr>
<tr>
<td>sub</td>
<td>( x - y )</td>
<td>Integer subtraction (2's complement)</td>
</tr>
<tr>
<td>neg</td>
<td>( -y )</td>
<td>Arithmetic negation (2's complement)</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>

Stack diagram:

- \( \cdots \)
- \( x \)
- \( y \)
- SP
A typical modern programming language features (at least) the following variable kinds:

- **Class level**
  - Static variables
  - Field variables (object properties)

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

- **And:**
  - Array variables
  - Constants

The language translator, of which the VM is a key component, must support all these variable kinds.
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 vectors:

- **static**: holds global variables, shared by all functions in the same class
- **argument**: holds argument variables of the current function
- **local**: holds local variables of the current function
- **constant**: pseudo segment holding all the constants in the range 0…32767
- **this, that**: general-purpose segments; can be made to represent different areas in the heap
- **pointer**: used to map `this` and `that` on 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, you have to understand the spirit of VM programming. So we will now see some examples:
  - Arithmetic task
  - Object handling task
  - Array 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;
}

VM code (first approx.)

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
Object handling example

/* 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

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

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 1</td>
<td>0 17</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 1</td>
<td>0 17</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

(this 0 is now alligned with RAM[3012])
/* 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
Lecture plan

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

<table>
<thead>
<tr>
<th>Arithmetic / Boolean commands</th>
<th>Program flow commands</th>
<th>Function calling commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>label  (declaration)</td>
<td>function (declaration)</td>
</tr>
<tr>
<td>sub</td>
<td>goto   (label)</td>
<td>call    (a function)</td>
</tr>
<tr>
<td>neg</td>
<td>if-goto (label)</td>
<td>return   (from a function)</td>
</tr>
<tr>
<td>eq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Famous translator-based implementations:**

- JVM (runs bytecode 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:** (a) map the VM constructs on the host RAM, and (b) 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 RAM. 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 \((base + i)\) 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 implementation 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.
Perspective

- 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)

- History of virtual machines (some milestones)
  - 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)

- Road ahead: complete the VM spec. and implementation (chs. 7,8), build a compiler (Ch. 9,10,11), then build a run-time library (ch. 12).