CSCI 410
13 – The Hack Virtual Machine
Part I: Stack and Memory Operations

The Hack Virtual Machine in a Nutshell

- All operations are done on a stack (LIFO)
- Data is saved in several separate memory segments (RA)
- All the memory segments behave the same
- E.g., one memory segment is called static

Stack and Memory

- All operations are done on a stack (LIFO)
- Data is saved in several separate memory segments (RA)
- All the memory segments behave the same
- E.g., one memory segment is called static

VM features a single 16-bit data type that can be used as:
- an integer value (16-bit 2's complement: -32768, ..., 32767)
- a Boolean value (0 and -1, standing for false and true)
- a pointer (memory address)

Types

Memory Access Operations

- Push items to the stack from a memory segment:
  push segment offset

- Pop items off the stack and store in memory:
  pop segment offset

As mentioned previously, all memory segments behave identically (more or less). The segment static will be used for examples. We'll also use the pseudo-segment constant.

Credit
Most of the slides in this lecture come from www.nand2tetris.org
Push & Pop

Arithmetic/Boolean Operations

- Binary: add, sub, eq, lt, gt, and, or
  - Pop two items from stack
  - Apply binary operation
  - Push result back on stack

- Unary: neg, not
  - Pop one item from stack
  - Apply unary operation
  - Push result back on stack

Arithmetic/Boolean Summary

<table>
<thead>
<tr>
<th>Command</th>
<th>Return value (after popping operands)</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>$\neg x$</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>$x$ and $y$</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>or</td>
<td>$x$ or $y$</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>not</td>
<td>$\neg x$</td>
<td>Bit-wise</td>
</tr>
</tbody>
</table>

Arithmetic Example

Initial

<table>
<thead>
<tr>
<th>$\text{Initial}$</th>
<th>$\text{VM Code}$</th>
<th>$\text{Final}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>push constant 4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>push static 0</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>push constant 10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>push static 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>push constant 200</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>push static 2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>push static 3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>push constant 10</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>push static 4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>push constant 200</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>push static 1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>push constant 2</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>push static 3</td>
<td></td>
</tr>
</tbody>
</table>

VM Code

<table>
<thead>
<tr>
<th>Stack</th>
<th>static</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>177</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Final

<table>
<thead>
<tr>
<th>Stack</th>
<th>static</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: constant is a “segment” which simply provides the constants 0 .. 32767.

The Memory Segments

A VM program is designed to provide an interim abstraction of a program written in some high-level language.

Modern OO languages typically feature the following variable kinds:

- Class level:
  - Static variables (class-level variables)
  - Private variables (aka “instance variables” / “fields” / “properties”)

- Method level:
  - Local variables
  - Argument variables

When translated into the VM language:

- StaticVars $\rightarrow$ static
- PrivateVars $\rightarrow$ this
- LocalVars $\rightarrow$ local
- Arguments $\rightarrow$ argument

Other segments (to be explored later)
VM Programming Preview

VM programs are normally written by compilers, not by humans. However, compilers are written by humans ...

In order to write or optimize a compiler, it helps to first understand the spirit of the compiler’s target language – the VM language

So, we’ll now see an example of a VM program

The example includes three new VM commands:

- `function name #locals  // function declaration`
- `label symbol   // label declaration`
- `if-goto symbol  // control`

---

**Example VM Code**

### 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 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 result
  push x
  add
  pop result
  push y
  push 0
  label loop
  push j
  push 1
  sub
  push j
  goto loop
  label end
  push local 0
  return
}
```

---

Just after `mult(7,3)` returns:

```
x
y
```

Just after `mult(7,3)` is entered:

```
SP
SP
21
7
3
0
```

...