CSCI 410
15 – The Hack Virtual Machine
Part 3: Program Flow and Function Calls

The Hack Virtual Machine in a Nutshell

<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</td>
<td>function (declaration)</td>
</tr>
<tr>
<td>sub</td>
<td>goto</td>
<td>(a function)</td>
</tr>
<tr>
<td>neg</td>
<td>if-goto</td>
<td>return</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>ge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>re</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory access commands

<table>
<thead>
<tr>
<th>pop segment offset</th>
<th>push segment offset</th>
</tr>
</thead>
</table>

Program Flow in the Hack VM

- **label name**
  - Names a location in the current function
  - Must be unique within the function
- **goto name**
  - Unconditionally jump to label name
- **if-goto name**
  - Pops top value in stack
  - If **not**, jumps to label name

Functions

- **Language should be extensible by user-defined ops**
- **User-defined ops should behave like built-in ops**

Example code (high-level language):

```plaintext
// Compute \( x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \) if \(-a \neq 0\)
\( x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \)
else
\( x = -\frac{c}{b} \)
```

Example code (high-level language):

```plaintext
function mult 1
  push constant 0
  pop local 0
  label loop
  push argument 0
  push constant 0
  eq
  if-goto end
  push argument 0
  push 1
  sub
  pop argument 0
  push local 0
  add
  goto loop
  push local 0
  return
```

Credit

Some of the slides in this lecture come from www.nand2tetris.org
Functions in the Hack VM

Primitives and subroutines follow same rules:
- Caller pushes the argument(s) and calls function
- Function removes the argument(s) from the stack, and pushes result on the stack

VM subroutine call and return conventions

Calling code, aka “callee” (example)
```
// computes (7 + 2) * 3
push constant 7
push constant 2
add
push constant 3
call mult
push constant 5
sub...
```

Called code, aka “caller” (example)
```
// computes (7 + 2) * 3
push constant 0
pop local 0 // result (local 0) = 0
label loop
push argument 0
push constant 0
eq
if
- goto end // if arg0 == 0, jump to end
push argument 0
push 1
sub
pop argument 0  // arg0 --
push argument 1
push local 0
add
pop local 0 // result = arg0
goto loop
label end
push local 0 // push result
return
```

Called code, aka “callee” (example)
```
function mult 1
push constant 0
pop local 0 // result (local 0) = 0
label loop
push argument 0
push constant 0
eq
if
- goto end // if arg0 == 0, jump to end
push argument 0
push 1
sub
pop argument 0  // arg0 --
push argument 1
push local 0
add
pop local 0 // result = arg0
goto loop
label end
push local 0 // push result
return
```

Called Function’s View of Stack (Start of Call)
```
function name nLocals
```

Called Function’s View of Stack (Start of Call)
```
```
```

Caller’s View of Stack (Before)
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Called Function’s View of Stack (Right Before return)

- Stack leftovers
- Return value

Return value is on top of stack.

Implementation

- Previous slides describe things from VM’s point of view
- Now, we turn to how flow control and functions are mapped onto the Hack hardware platform

Implementing VM on Hack Platform

The challenge:
- Done:
  - Manage memory ops (push/pop from/to segments)
  - Handle arithmetic, Boolean operations
- To do:
  - Manage program flow with goto, if-goto and label
    - (Almost) direct translation to assembly code
    - Just have to discuss how to translate labels
  - Make function call protocol work on Hack platform
  - Add in initialization code

The Stack Frame

Argument 0
Argument 1
Argument n-1
Base of working stack

Managed by VM: not "visible" to running VM program. (E.g., no way to access in VM code.)

Local 0
Local 1
Local k-1
**label**

- The label VM command defines a unique location in a function's code
- This has to be translated into a **globally** unique location in assembly code
- Solution: function-name$label-name
  - Function names are unique globally
  - Labels are unique per function

Since all VM code resides in functions, VM translator just needs to keep track of current function name.

**goto, if-goto**

Almost too trivial to discuss...

goto: unconditional jump to label

if-goto: conditional jump on not-equal to zero

---

**call**

Do the following steps (translated to assembly code):
1. Push the return address (use label declared below)
2. Push LCL
3. Push ARG
4. Push THIS
5. Push THAT
6. Set ARG = SP – nargs – 5
7. Set LCL = SP
8. Goto name
9. Set return address label (must be globally unique!)

---

**Stack (call)**

BEFORE | AFTER
--- | ---
... | ...
Argument 0 | Argument 0
Argument 1 | Argument 1
... | ...
Argument nargs - 1 | Argument nargs - 1
Return address | Return address
Saved LCL | Saved LCL
Saved ARG | Saved ARG
Saved THIS | Saved THIS
Saved THAT | Saved THAT

---

**function**

Do (in assembly):
1. Set label for function name
2. Push 0 nlocals times

---

**Stack (function)**

BEFORE | AFTER
--- | ---
... | ...
Argument 0 | Argument 0
Argument 1 | Argument 1
... | ...
Argument nargs - 1 | Argument nargs - 1
Return address | Return address
Saved LCL | Saved LCL
Saved ARG | Saved ARG
Saved THIS | Saved THIS
Saved THAT | Saved THAT
Local 0 | Local 0
Local 1 | Local 1
Local nlocals - 1 | Local nlocals - 1
This is the most difficult of the three! Do:
1. Store address in LCL to a temp location (e.g., in R13) – call this “FRAME”
2. Store return address (stored in FRAME – 5) in another temp location (e.g., R14) – call this “RET”
3. Pop return value and relocate to location held in ARG
4. Set SP = ARG + 1
5. Restore pointers:
   • THAT = address stored in FRAME – 1
   • THIS = address stored in FRAME – 2
   • ARG = address stored in FRAME – 3
   • LCL = address stored in FRAME – 4
6. Goto RET

Hack program always starts at ROM[0]:
What do we put there?
• RAM[SP] = 256 // set start location of stack
• call Sys.init // VM defined start function // (supplied by programmer)

• Provided to you: 5 test “programs”
  • Get ProgramFlow programs working first (goto, etc.)
  • Then do FunctionCalls programs (one at a time!)
• ProgramFlow, first FunctionCalls programs do not include Sys.init
  • Translate single .vm file to .asm
  • Test using .tst script in CPUEmulator (test script does initialization for you)
• Last 2 are multi-file (including Sys.vm)
• Take your time!

Include comments in emitted assembly code
• Write a function for this in your translator, use it e.g., to note start of VM commands
• Why? You will likely need to debug in part by staring at your assembly code.
• When you fail a test:
  • Figure out what the test code is doing
  • Figure out what the .vm code is supposed to do
  • Figure out what your code was expected to do
  • Compare your output to expected output to see where expectations differed from reality
  • Together with incremental development, this should help you narrow down your search to a single VM command