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>call (a function)</td>
</tr>
<tr>
<td>neg</td>
<td>eq</td>
<td>return (from a function)</td>
</tr>
<tr>
<td>gt</td>
<td>lt</td>
<td></td>
</tr>
<tr>
<td>lt</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>or</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>not</td>
<td></td>
</tr>
</tbody>
</table>

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

Segments: local, argument, static, this, that, pointer, temp, constant

Memory Segments Local to a Function

- local
  - where local variables for a function live
  - created on the stack with a new stack frame, and destroyed when the frame is popped
- argument
  - where arguments for a function live
  - like local, these are part of the stack frame
- this, that
  - these move around based on context
  - this corresponds to the current object
  - that is used to access memory in dynamically allocated arrays

Fixed Memory Segments

- pointer – a 2-word memory segment which simply holds the current addresses of this and that
- temp – an 8-word memory segment which can be used for temp storage by VM functions (think registers)
constant: Memory Pseudo-segment

push constant offset:
  • simply pushes offset onto the stack
  • allowed values for offset are all positive (16-bit 2’s complement) integers: 0 – 32767
  • VM supplies these constants, they don’t actually “live” anywhere

static: The Global Memory Segment

• Shared by all functions in a single module
• Offsets are faked using assembly code variables (explained later)

The Stack

Implicit in the functioning of the VM

• For each function call, create a stack frame
  • argument segment
  • Bookkeeping information:
    • Return address (where do we jump after function call)
    • Saved segment base addresses for prior function
      • local
      • argument
      • this, that
  • local segment
  • Working stack memory

The Stack Frame

Memory Map VM → Hack Platform

Big Picture of Programming

Compilation:
  • A Jack application is a set of 1 or more class files (just like .java files).
  • When we apply the Jack compiler to these files, the compiler creates a set of 1 or more .vm files (just like .class files). Each method in the Jack app is translated into a VM function written in the VM language.
  • Thus, a VM file consists of one or more VM functions.

VM Translation:
  • A VM application is a set of 1 or more .vm files
  • When we apply the VM translator to these files, the translator creates a set of 1 or more .asm files
  • All .asm files are “linked” together with a start routine to create a working program.
Programming the VM: Stack Manipulation

- **push** – assembly code to copy word to address given by SP (RAM[0]), increment SP
- **pop** – assembly code to copy word from address given by (*SP – 1), decrement SP

Design decision: where do you push/pop from/to?
- Remember, you have only 2 registers and M
- A holds address of M, pretty much out of the game
- D becomes your best friend
- Consider writing translator functions just for push/pop from/to D (you will get a lot of use out of them!)

Programming the VM: Memory Segments

- For constant segment, just use assembly to load value directly (@val)
  - Then, “D=A”, followed by push from D
- For local, argument, this, that:
  - Load base address from special RAM locations LCL, ARG, THIS, THAT
  - **push**:
    - Compute offset address, copy into A
    - “D=M” followed by push from D
  - **pop** (this one is tricky!):
    - Compute offset address, store somewhere
    - pop to D
    - Copy offset address into A
    - “M=D”

 temp, pointer easier since base address fixed

Programming the VM: Faking the static Segment

- The static memory region is RAM[16..255].
- However, each .vm file gets its own static segment.
- Normally this would be a linking issue, but we aren’t writing a linker.
- Solution: fake static segment using assembly vars; every time you have push/pop static offset:
  - Load variable “filename.offset”
    - E.g., for .vm file foo.vm, push static 3 would start with “@foo.3”
    - Assembler will automatically assign each of these a unique location starting at address 16
    - Note offset isn’t really relative to a base address!

Programming the VM: Arithmetic

- For most operations, this is a simple matter.
- E.g., for binary ops:
  - Pop argument x into D
  - Copy x into one of R13-R15 (not reserved for anything else, use as extra “registers”)
  - Pop argument y into D
  - Load address where x is stored
  - Compute function with dest=D, e.g., “D=D+M”
  - Push result from D

lt, gt, eq are much trickier (next slide)

Programming Hints/Tips

- Start early!!!
- As previously mentioned, create code just for push/pop from/to D register:
  - Obviously useful for push/pop in arithmetic/Boolean ops
  - Can also stage all of your memory accesses through D
- You must not use R0-R12 or RAM[16] and up except as specified by VM memory model.
- However, you have R13-R15 available for general use (e.g., swap space)
- Follow the book’s design
- Especially follow the book’s proposed implementation stages (page 148), testing as you go!