Virtual Machine Structure

Lecture 20

Basics of the MJ Virtual Machine

Word addressed (in many other machines we are forever shifting by 2 bits to get from words to bytes or back.)

All instructions (appear to) fit in a single word. All integers fit in a single word. Everything else is "pointed to" and all pointers fit in a single word.

A minimum set of operations for MJ, but these could be expanded easily.

All arithmetic operations use a stack.

Why a stack?

- The usual alternative is: a pile of registers.

- Why use registers in IC?
  - Many, (all?) current architectures have registers.
  - If you want to control efficiency, you need to know how to save/restore/spill registers.
  - It is not too hard, if you have enough of them.

- Why use a stack?
  - Some architectures historically were stack-dependent because they had few registers (like 4, or 16..).
  - Some current architectures use a stack e.g. for floats in Pentium, 8 values.
  - Minimize complexity for code generation.

Why a stack? Are registers more work for IC?

- Generating code to load data into registers initially seems more complicated,
- Not by much: the compiler can keep track of which register has a value [perhaps by keeping a stack of variable-value pairs while generating code],
- And you did this in CS61c, but in your head, probably.

- With a finite number of registers there is always the possibility of running out: "spill" to a stack? Or…
- (New architectures with 128 registers or more make running out unlikely but then what?: perhaps "error, expression too complicated, compiler fails"?).
- Opportunity to optimize: rearrange expressions to use minimum number of registers. Good CS theory problem related to graph coloring. (In practice, registers are finicky, aligned, paired, special purpose,…)

Instructions: stack manipulation

```
pushi x   push immediate the constant x on the top of the stack
used only for literals.  Same as iconst.  e.g. (iconst 43)
Only 24 bits for x(?). (larger consts in 2 steps??)

pusha x   push address. pushes the address of x on stack.
e.g. pusha ="hello world".  We assume the assembler will
find some place for x.
    Same as sconst. e.g. (sconst "hello")

pop  pops top of stack; value is lost

dup  pushes duplicate of top of stack
swap guess ☺
```

A simple call / the thinking..

Consider a method

```
 public int function F(){return 3; /*here*/
                        }
```

How might we compile F() ? Set up a label L001 for location /*here*/.
Save it on the stack.
Push the address of F on the stack.
Execute a (callj 0) to call F, a function of 0 args
Execute an (args 0) /* get params, here none {what about THIS}*/
the stack looks like

    L001
    3

Execute a (return). Which jumps to L001, leaving 3 on the stack.
(exit 0)

A simple call/ the program

```
public int function F(){return 3; /*here*/
                        }
```

```
 (save L001)
 (lvar  1 0)  // magic... get address of f on stack.. Details follow
 (callj 0)    // call function of 0 args
L001:         // label to return to
 (exit 0)
```

The program f looks like

```
  (args 0)  // collect 0 arguments into environment..
  (pushi 3)
  (return)
```

More fun, less work, look at SCAM

```
(setf *fact-test* (compile-scam
  '( (define (main)
      (print (factorial 5)))
    (define (factorial n)
      (if n
        (* n (factorial (- n 1)))
        1)))))
```

More fun, less work, look at SCAM

```
(setf *fact-test* (compile-scam
  '( (define (main)
       (print (factorial 5)))
     (define (factorial n)
       (if n
          (* n (factorial (- n 1)))
          1)))))

(pprint-code *fact-test*)
(run-vm (make-vm-state :code (assemble
                                   *fact-test*)))
```