MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

Lecture Notes, November 17 – Register Machines

In order to understand basic concepts and abstractions relating to "low level" computational hardware and how programming languages can be built on top of such hardware, we will use a stylized version of a *register machine* that supports movement of data between primitive storage and computational elements. We will consider both *graphical* and *language based* (or textual) descriptions of such hardware.

Sketch a simple **data flow** diagram illustrating some basic elements:

Sketch a corresponding **control flow** diagram which dictates when data flows through respective elements:

Register machine language

We can also describe or capture such hardware operation in a *register machine language*. We can define a new machine as consisting of:

Register machine - controller language

The soul of the machine is the controller. We use the following controller language to succinctly capture the sequence of operation, as well as the basic connections and data flow in our machine:

```
(assign <reg-name> (reg <reg-name-2>))
(assign <reg-name> (const <constant-value>))
(assign <reg-name> (op <op-name>) <input-1> <input-2> ...)
(assign <reg-name> (label <label-name>))

(perform (op <op-name>) <input-1> <input-2> ...)

(test (op <op-name>) <input-1> <input-2> ...)
(branch (label <label-name>))
(goto (label <label-name>))
(goto (reg <reg-name>))

(save <reg-name>)
(restore <reg-name>)
```

where <input-i> is either (const <constant-value>) or (reg <reg-name>).

Example: An Iterative Process

```
(define (sum-roots sum from to)
  (define (sum-iter sum from to)
    (if (> from to)
        sum
        (sum-iter (+ sum (sqrt from))
                  (+ 1 from)
                  to)))
  (sum-iter 0 from to))
```

A register machine specification for sum-roots:

```
(define-machine sum-roots
 (registers sum from to temp)
 (operations + sqrt >)
 (controller
   sum-roots
     (assign sum (const 0))
   sum-iter
     (test (op >) (reg from) (reg to))
     (branch (label done))
     (assign temp (op sqrt) (reg from))
     (assign sum (op +) (reg sum) (reg temp))
     (assign from (op +) (const 1) (reg from))
     (goto (label sum-iter))
   done))
```

Sketch the data paths for a register machine that computes the sum of roots:

Sketch a control diagram for this machine:

Calling a subroutine

We can extend our language with the ability to store labels in registers, and to `goto` a label stored in a register. Using a `continue` register, a simple subroutine call becomes possible:

```
(controller
 ...
 ;; Contract: input registers from, to
 ;;           output in register sum
 ;; Returns to label in the continue register.
 sum-roots    ;; entry point
  (assign sum (const 0))
 sum-iter
  (test (op >) (reg from) (reg to))
  (branch (label done))
  (assign temp (op sqrt) (reg from))
  (assign sum (op +) (reg sum) (reg temp))
  (assign from (op +) (const 1) (reg from))
  (goto (label sum-iter))
 done
  (goto (reg continue))
  ...)
```

Stacks and Recursive Processes

A stack operates as a LIFO (Last-In, First-Out) queue. This allows us to store away (and retrieve) the contents of registers, and enables us to implement "recursive" hardware:

```
(define (sum-roots from to)
  (if (> from to)
      0                   ;; base case
      (+ (sqrt from)          ;; deferred operation
         (sum-roots (+ 1 from) to))))   ;; recursion
```

With a corresponding machine:

```
(define-machine sum-roots
 (registers continue from to temp)
 (operations + sqrt >)
 (controller
  ;; On entry  -- continue holds return label
  ;;           -- registers from, to hold input values
  ;; On return -- register val holds answer
  sum-roots
    (test (op >) (reg from) (reg to))
    (branch (label base-case))
    ;; Need to recurse, so remember what weUll need
    ;; for the deferred operation...
    (save from)
    (save continue)
```

```
    (assign continue (label do-deferred-operations))
    (assign from (op +) (const 1) (reg from))
    (goto (label sum-roots))  ; recurse
  base-case
    (assign val (const 0))
    (goto (reg continue))
  do-deferred-operations
    (restore continue)  ;; restore in reverse order!
    (restore from)
    (assign temp (op sqrt) (reg from))
    (assign val (op +) (reg val) (reg temp))
    (goto (reg continue))))
```