Exercise 5.4.

Specify register machines that implement each of the following procedures. For each machine, write a controller instruction sequence and draw a diagram showing the data paths.

a. Recursive exponentiation:

```
(define (expt b n)
  (if (= b 0)
      1
      (* b (expt b (- n 1)))))
```

b. Iterative exponetiation:

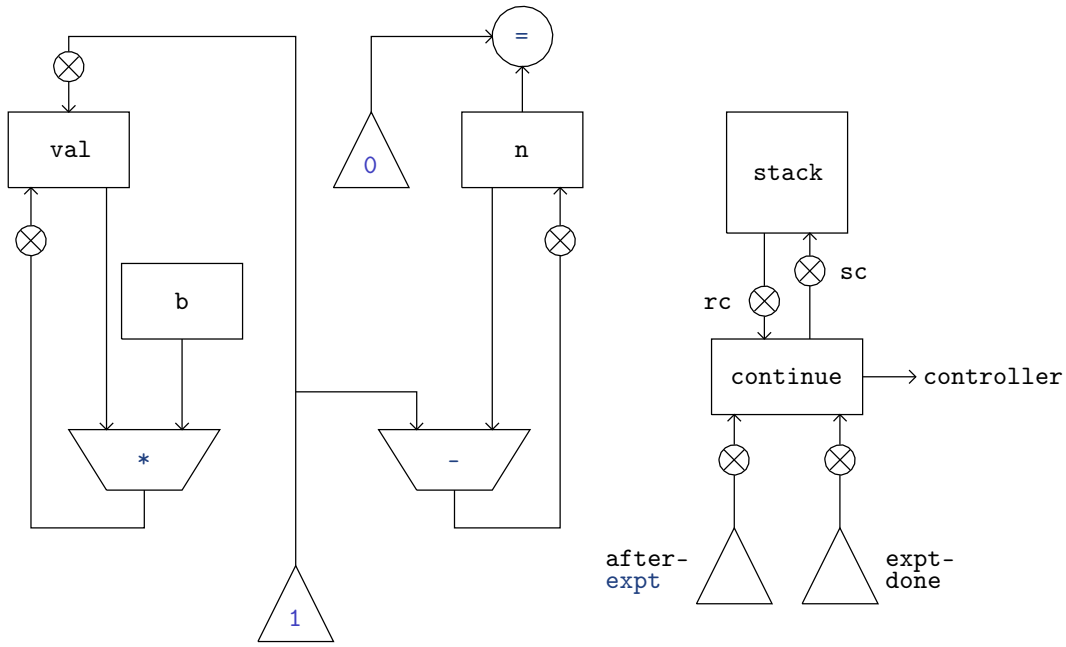
```
(define (expt b n)
  (define (expt-iter counter product)
    (if (= counter 0)
        product
        (expt-iter (- counter 1) (* b product))))
  (expt-iter n 1))
```

Answer.

a. Figure 1 shows the data paths and controller for a machine that implements the recursive exponentiation procedure. The machine has a stack and four registers, called `b`, `n`, `val` and `continue`. Like the `factorial` machine, we omit the register-assignment buttons to simplify the data path diagram. To operate the machine, we put in registers `b` and `n` the number whose exponent we wish to compute and start the machine. When the machine reaches `expt-done`, the computation is finished and the answer will be found in the `val` register. Unlike the `factorial` machine, we only save contents of `continue` register before each recursive call and restored upon return from the call. The `n` and `val` registers are not saved before the recursive call, because their old contents are no longer useful after the subroutine returns.

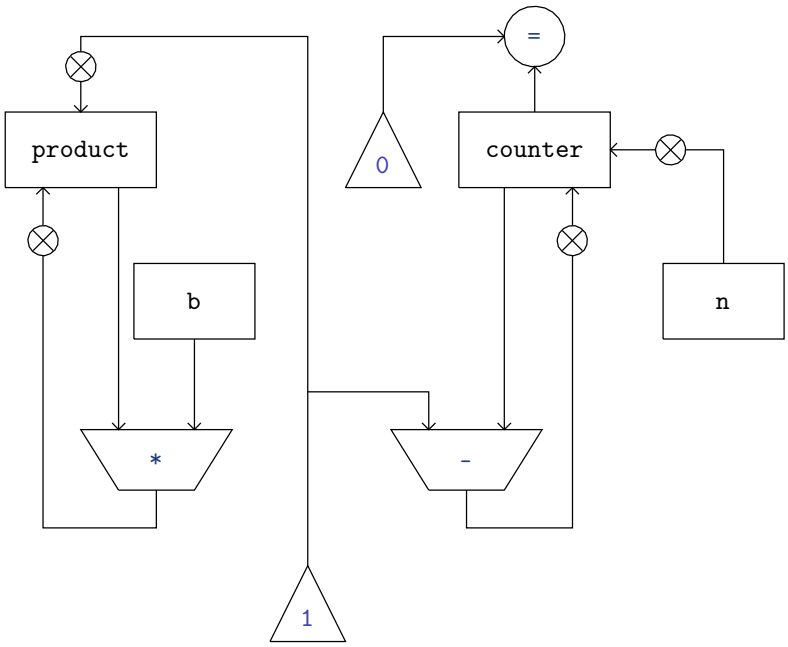
b. Figure 2 shows the data paths and controller for a machine that implements the iterative exponentiation procedure. The machine has four registers, called `b`, `n`, `counter` and `product`. We still put in registers `b` and `n` to employ the the machine to compute the exponent of a number. When the `counter` register hits 0, the computation is finished and the answer will be found in `product` register. The iterative exponentiation machine turn out to be much simpler than the recursive one, both on the data paths diagram and controller sequence. This divergence of complexity on machine construction illustrates the reason why procedures that generate iterative process work more efficient than those that generate recursive process.

Email address: informlarry@gmail.com




```
(controller
   (assign continue (label expt-done))        ; set up final return address
 expt-loop
   (test (op =) (reg n) (const 0))
   (branch (label base-case))
   ;; Set up for the recursive call by saving continue.
   ;; Set up continue so that the computation will continue
   ;; at after-expt when the subroutine returns.
   (save continue)
   (assign n (op -) (reg n) (const 1))
   (assign continue (label after-expt))
   (goto (label expt-loop))
 after-expt
   (restore continue)
   (assign val (op *) (reg b) (reg val))      ; val now contains b·b^(n-1)
   (goto (reg continue))                      ; return to caller
 base-case
   (assign val (const 1))                     ; base case: b^0 = 1
   (goto (reg continue))                      ; return to caller
 expt-done)
```

Figure 1. A recursive exponentiation machine.




```
(controller
   (assign counter (reg n))
   (assign product (const 1))
 expt-loop
   (test (op =) (reg counter) (const 0))
   (branch (label expt-done))
   (assign counter (op -) (reg counter) (const 1))
   (assign product (op *) (reg b) (reg product))
   (goto (label expt-loop))
 expt-done)
```

Figure 2. A iterative exponentiation machine.