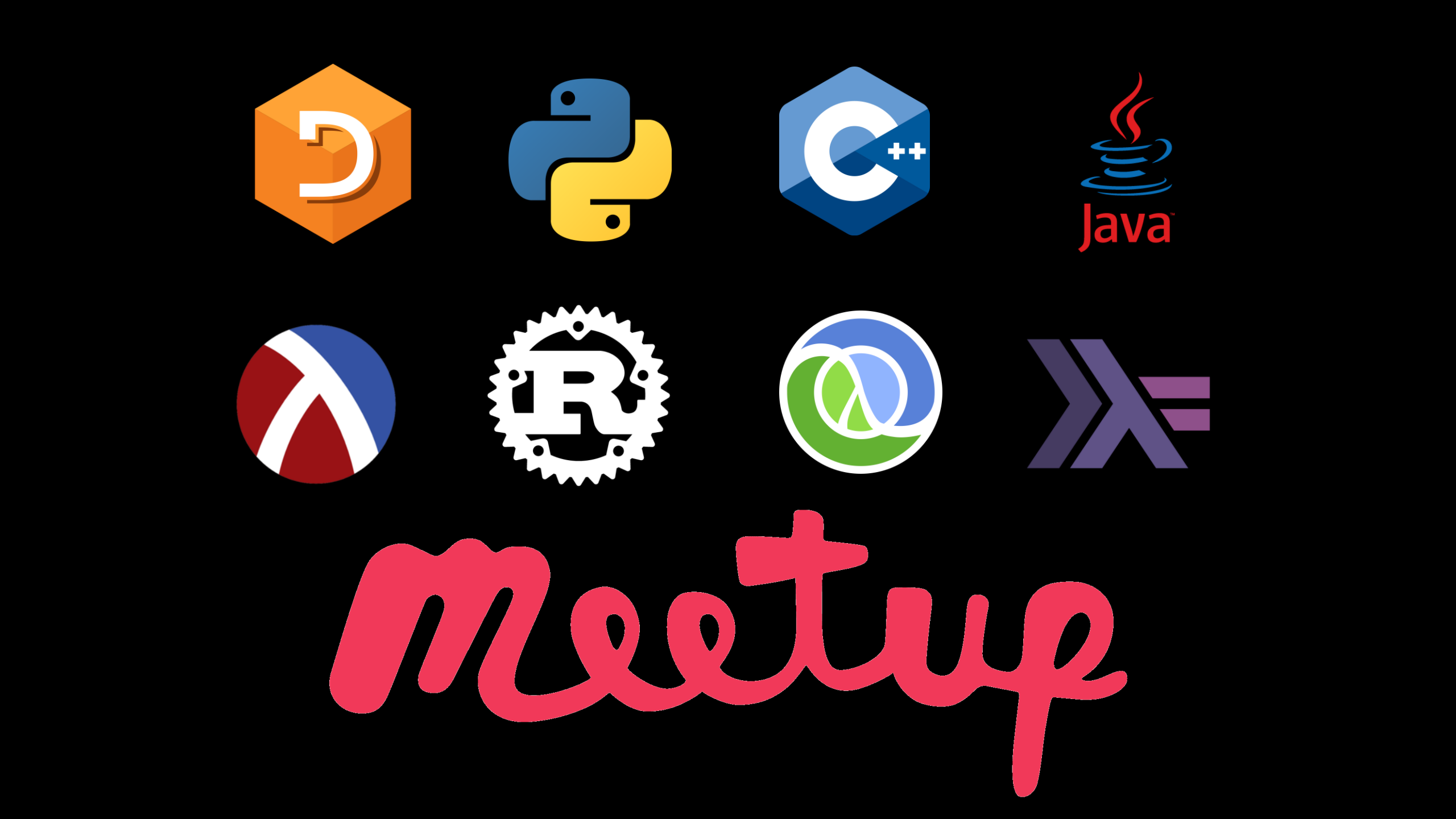
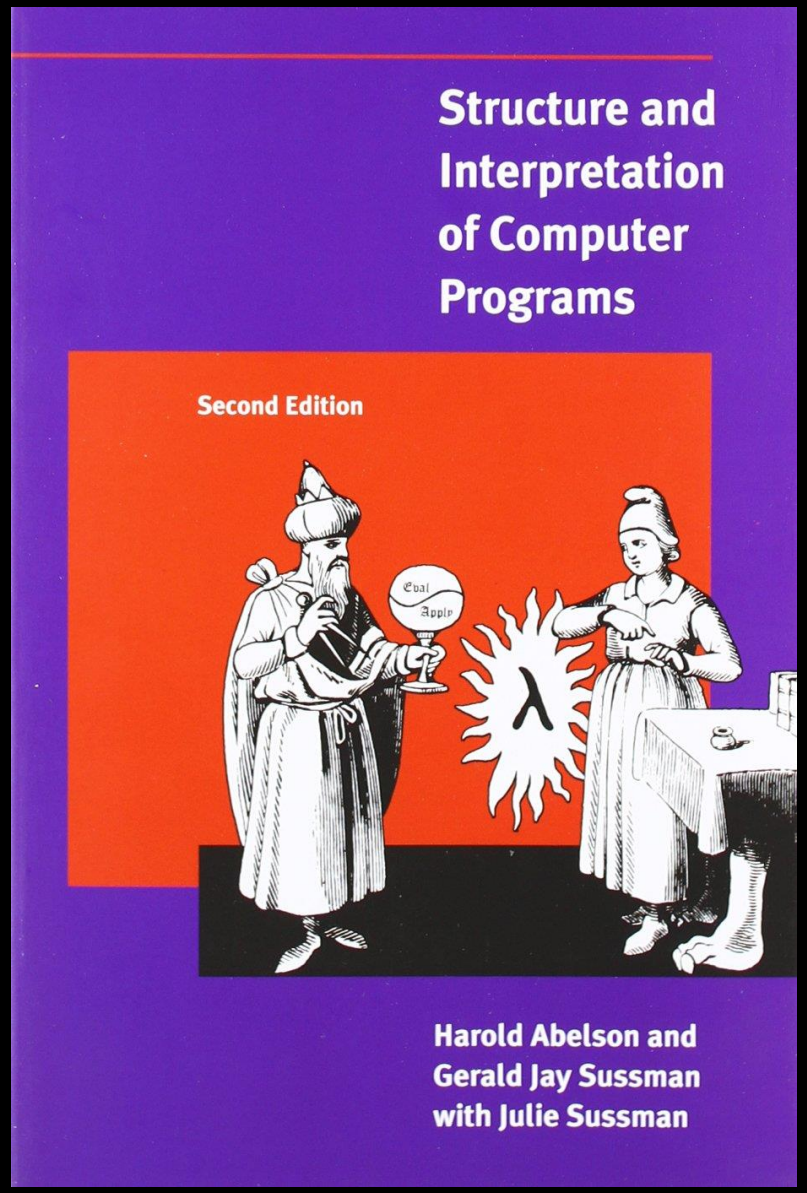

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 5.2

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

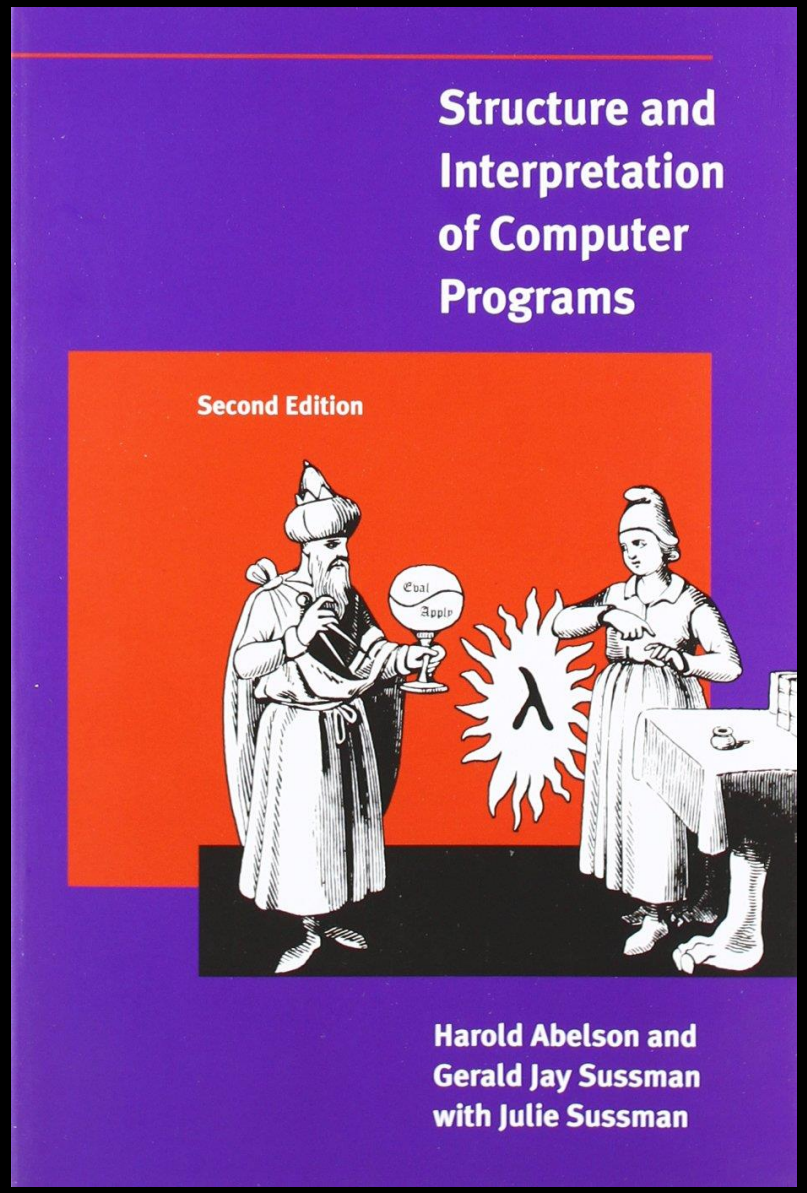
Following

Programming Languages Virtual Meetup

@PLvirtualmeetup

Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

Toronto, CA meetup.com/Programming-La... Joined March 2020



Structure and Interpretation of Computer Programs

Chaper 5.2

In order to gain a good understanding of the design of register machines, we must test the machines we design to see if they perform as expected. One way to test a design is to hand-simulate the operation of the controller, as in Exercise 5.5. But this is extremely tedious for all but the simplest machines. In this section we construct a simulator for machines described in the register-machine language. The simulator is a Scheme program with four interface procedures. The first uses a description of a register machine to construct a model of the machine (a data structure whose parts correspond to the parts of the machine to be simulated), and the other three allow us to simulate the machine by manipulating the model:

`(make-machine ⟨register-names⟩ ⟨operations⟩ ⟨controller⟩)`

> constructs and returns a model of the machine with the given registers, operations, and controller.

As an example of how these procedures are used, we can define gcd-machine to be a model of the GCD machine of Section 5.1.1 as follows:

```
(define gcd-machine
  (make-machine
   '(a b t)
   (list (list 'rem remainder) (list '= =))
   '(test-b (test (op =) (reg b) (const 0))
            (branch (label gcd-done))
            (assign t (op rem) (reg a) (reg b))
            (assign a (reg b))
            (assign b (reg t))
            (goto (label test-b))
            gcd-done)))
```

```
;; 5.2.1 The Machine Model

(define (make-machine register-names ops controller-text)
  (let ((machine (make-new-machine)))
    (for-each
     (lambda (register-name)
       ((machine 'allocate-register) register-name))
     register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine))
    machine))
```

```
;; Registers

(define (make-register name)
  (let ((contents '*unassigned*))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value) (set! contents value)))
            (else
             (error "Unknown request: REGISTER" message))))
    dispatch))

(define (get-contents register) (register 'get))
(define (set-contents! register value)
  ((register 'set) value))
```

```
;; The stack

(define (make-stack)
  (let ((s '()))
    (define (push x) (set! s (cons x s)))
    (define (pop)
      (if (null? s)
          (error "Empty stack: POP")
          (let ((top (car s)))
            (set! s (cdr s))
            top)))
    (define (initialize)
      (set! s '())
      'done)
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) (pop))
            ((eq? message 'initialize) (initialize))
            (else (error "Unknown request: STACK" message))))
    dispatch))

(define (pop stack) (stack 'pop))
(define (push stack value) ((stack 'push) value))
```

```
;; The basic machine

(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (let ((the-ops
           (list (list 'initialize-stack
                       (lambda () (stack 'initialize)))))
          (register-table
           (list (list 'pc pc) (list 'flag flag))))
      (define (allocate-register name)
        (if (assoc name register-table)
            (error "Multiply defined register: " name)
            (set! register-table
                  (cons (list name (make-register name))
                        register-table)))
        'register-allocated)
```

```
(define (lookup-register name)
  (let ((val (assoc name register-table)))
    (if val
        (cadr val)
        (error "Unknown register:" name))))
(define (execute)
  (let ((insts (get-contents pc)))
    (if (null? insts)
        'done
        (begin
          ((instruction-execution-proc (car insts)))
          (execute)))))
```

```
(define (dispatch message)
  (cond ((eq? message 'start)
         (set-contents! pc the-instruction-sequence)
         (execute))
        ((eq? message 'install-instruction-sequence)
         (lambda (seq)
           (set! the-instruction-sequence seq)))
        ((eq? message 'allocate-register)
         allocate-register)
        ((eq? message 'get-register)
         lookup-register)
        ((eq? message 'install-operations)
         (lambda (ops)
           (set! the-ops (append the-ops ops))))
        ((eq? message 'stack) stack)
        ((eq? message 'operations) the-ops)
        (else (error "Unknown request: MACHINE"
                     message))))
dispatch)))
```

```
(define (start machine) (machine 'start))
(define (get-register-contents machine register-name)
  (get-contents (get-register machine register-name)))
(define (set-register-contents! machine register-name value)
  (set-contents! (get-register machine register-name)
                 value)
  'done)

(define (get-register machine reg-name)
  ((machine 'get-register) reg-name))
```

```
;; 5.2.2 The Assembler

(define (assemble controller-text machine)
  (extract-labels
   controller-text
   (lambda (insts labels)
     (update-insts! insts labels machine)
     insts)))
```

```
(define (extract-labels text receive)
  (if (null? text)
      (receive '() '())
      (extract-labels
       (cdr text)
       (lambda (insts labels)
         (let ((next-inst (car text)))
           (if (symbol? next-inst)
               (if (assoc next-inst labels)
                   (error "repeated label: " next-inst)
                   (receive insts
                            (cons (make-label-entry next-inst
                                                    insts)
                                  labels)))
               (receive (cons (make-instruction next-inst)
                              insts)
                        labels)))))))
```

Exercise 5.8: The following register-machine code is ambiguous, because the label `here` is defined more than once:

```
start
  (goto (label here))
here
  (assign a (const 3))
  (goto (label there))
here
  (assign a (const 4))
  (goto (label there))
there
```

With the simulator as written, what will the contents of register a be when control reaches `there`? Modify the `extract-labels` procedure so that the assembler will signal an error if the same label name is used to indicate two different locations.

```
(define exercise-5-8
  (make-machine
   '(a)
   (list (list 'print display))
   '(start
     (goto (label here))
     here
     (assign a (const 3))
     (goto (label there))
     here
     (assign a (const 4))
     (goto (label there))
     there
     (perform (op print) (reg a)))))

(start exercise-5-8) ; 3
```

```
;; before

(define (extract-labels-before text receive)
  (if (null? text)
      (receive '() '())
      (extract-labels
       (cdr text)
       (lambda (insts labels)
         (let ((next-inst (car text)))
           (if (symbol? next-inst)
               (receive insts
                        (cons (make-label-entry next-inst
                                                insts)
                              labels))
               (receive (cons (make-instruction next-inst)
                              insts)
                        labels)))))))
```

```
;; after

(define (extract-labels-after text receive)
  (if (null? text)
      (receive '() '())
      (extract-labels
       (cdr text)
       (lambda (insts labels)
         (let ((next-inst (car text)))
           (if (symbol? next-inst)
               (if (assoc next-inst labels)
                   (error "repeated label: " next-inst)
                   (receive insts
                            (cons (make-label-entry next-inst
                                                    insts)
                                  labels)))
               (receive (cons (make-instruction next-inst)
                              insts)
                        labels)))))))
```

```
;; 5.2.3 Generating Execution Procedures for Instructions

(define (make-execution-procedure
         inst labels machine pc flag stack ops)
  (cond ((eq? (car inst) 'assign)
         (make-assign inst machine labels ops pc))
        ((eq? (car inst) 'test)
         (make-test inst machine labels ops flag pc))
        ((eq? (car inst) 'branch)
         (make-branch inst machine labels flag pc))
        ((eq? (car inst) 'goto)
         (make-goto inst machine labels pc))
        ((eq? (car inst) 'save)
         (make-save inst machine stack pc))
        ((eq? (car inst) 'restore)
         (make-restore inst machine stack pc))
        ((eq? (car inst) 'perform)
         (make-perform inst machine labels ops pc))
        (else
         (error "Unknown instruction type: ASSEMBLE"
                inst))))
```

```
;; assign instructions

(define (make-assign inst machine labels operations pc)
  (let ((target
         (get-register machine (assign-reg-name inst)))
        (value-exp (assign-value-exp inst)))
    (let ((value-proc
           (if (operation-exp? value-exp)
               (make-operation-exp
                value-exp machine labels operations)
               (make-primitive-exp
                (car value-exp) machine labels))))
      (lambda () ; execution procedure for assign
        (set-contents! target (value-proc))
        (advance-pc pc)))))
```

Exercise 5.13: Modify the simulator so that it uses the controller sequence to determine what registers the machine has rather than requiring a list of registers as an argument to `make-machine`. Instead of pre-allocating the registers in `make-machine`, you can allocate them one at a time when they are first seen during assembly of the instructions.

```
;; before

(define (make-machine-before register-names ops controller-text)
  (let ((machine (make-new-machine)))
    (for-each
     (lambda (register-name)
       ((machine 'allocate-register) register-name))
     register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine))
    machine))
```

```
;; after

(define (get-register-names controller-text)
   (map cadr
        (filter (lambda (x) (and (list? x) (eq? (car x) 'assign)))
                controller-text)))

(define (make-machine-after ops controller-text)
  (let ((machine (make-new-machine))
        (register-names (get-register-names controller-text)))
    (for-each
     (lambda (register-name)
       ((machine 'allocate-register) register-name))
     register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine))
    machine))
```

```
;; test

(define gcd-machine-2
  (make-machine-after
   ;'(a b t)
   (list (list 'rem remainder) (list '= =))
   '(test-b (test (op =) (reg b) (const 0))
            (branch (label gcd-done))
            (assign t (op rem) (reg a) (reg b))
            (assign a (reg b))
            (assign b (reg t))
            (goto (label test-b))
            gcd-done)))

(set-register-contents! gcd-machine-2 'a 30) ; done
(set-register-contents! gcd-machine-2 'b 42) ; done
(start gcd-machine-2)                        ; done
(get-register-contents gcd-machine-2 'a)     ; 6
```

Exercise 5.16: Augment the simulator to provide for *instruction tracing*. That is, before each instruction is executed, the simulator should print the text of the instruction. Make the machine model accept `trace-on` and `trace-off` messages to turn tracing on and off.

```
;; Exericise 5.16 (page 721)

;; copy code from http://community.schemewiki.org/?sicp-ex-5.16

(trace-on-instruction gcd-machine)
```

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

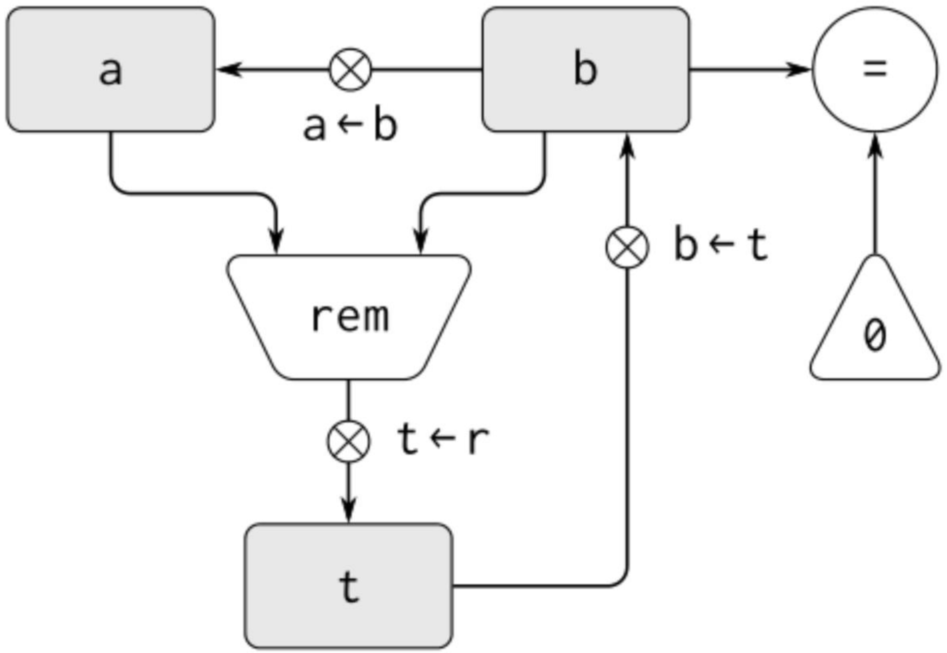


Figure 5.1: Data paths for a GCD machine.

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

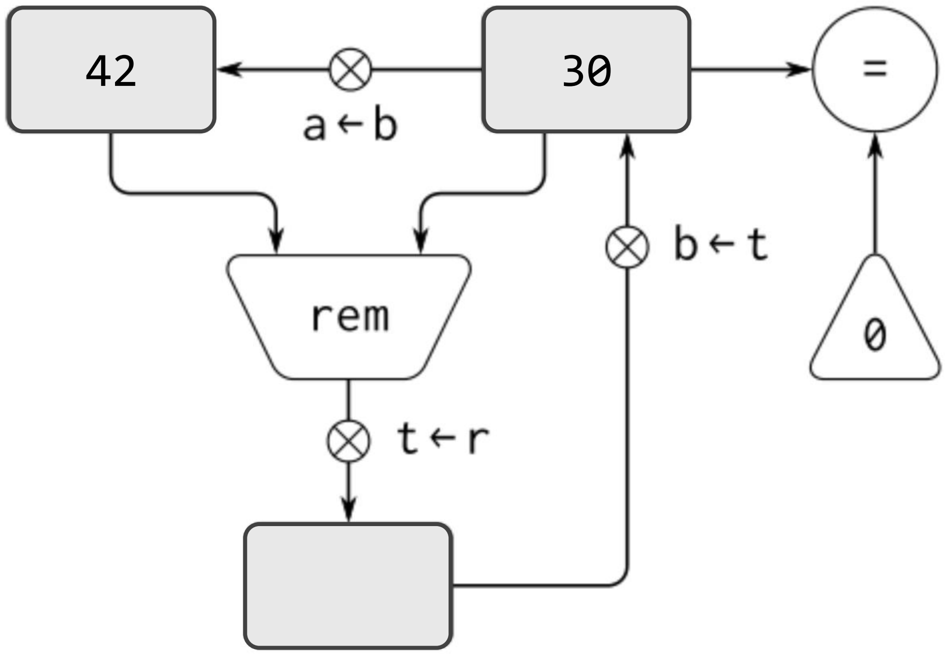


Figure 5.1: Data paths for a GCD machine.

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

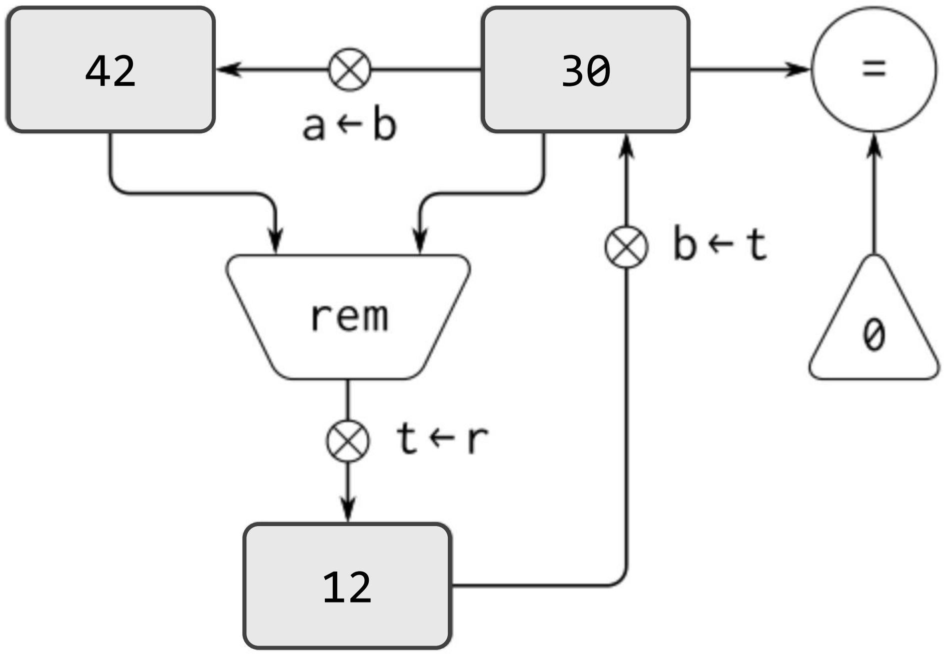


Figure 5.1: Data paths for a GCD machine.

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

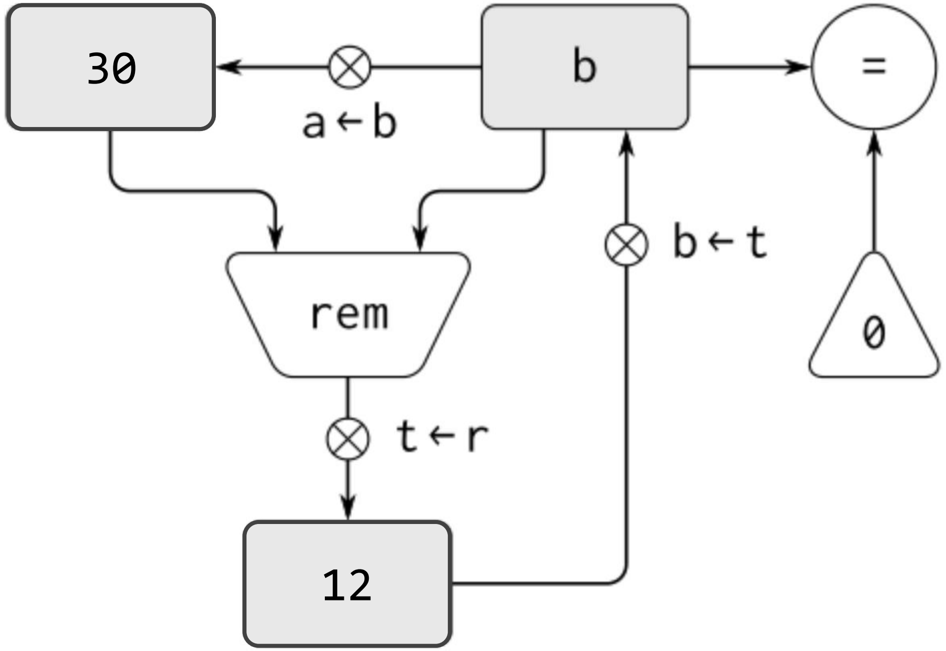


Figure 5.1: Data paths for a GCD machine.

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

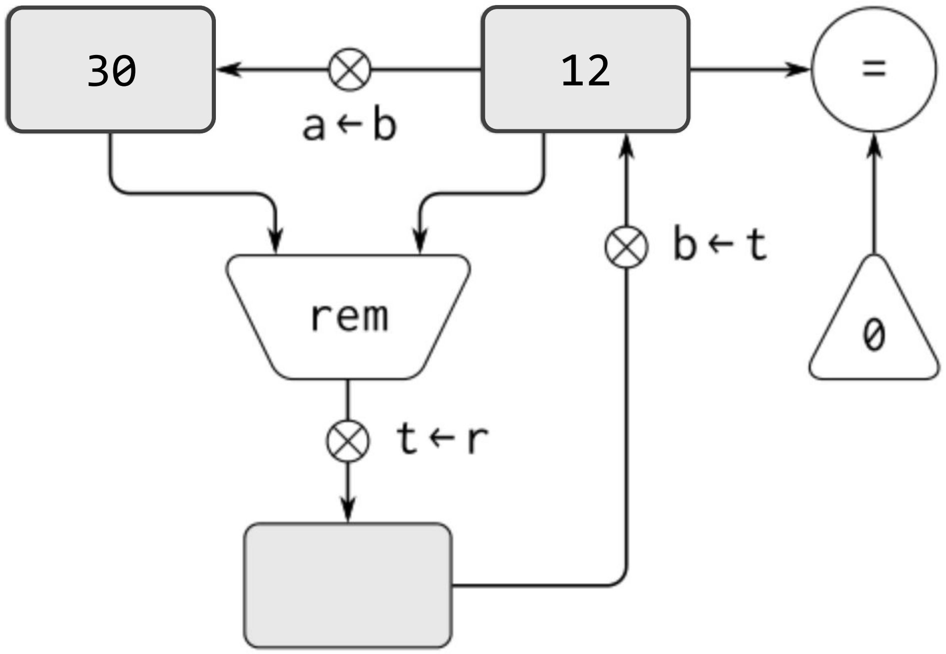


Figure 5.1: Data paths for a GCD machine.

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

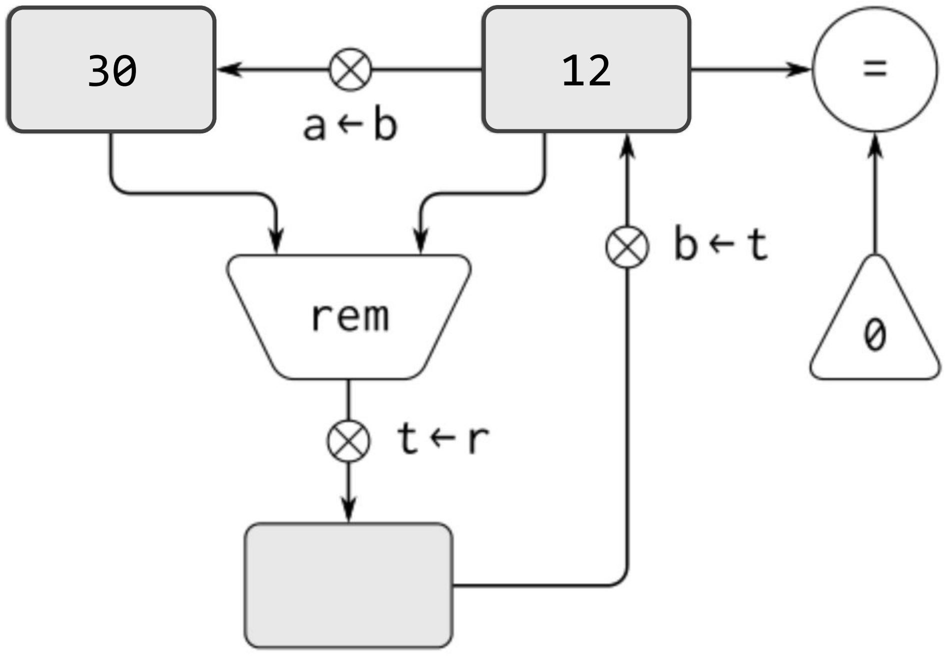


Figure 5.1: Data paths for a GCD machine.

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

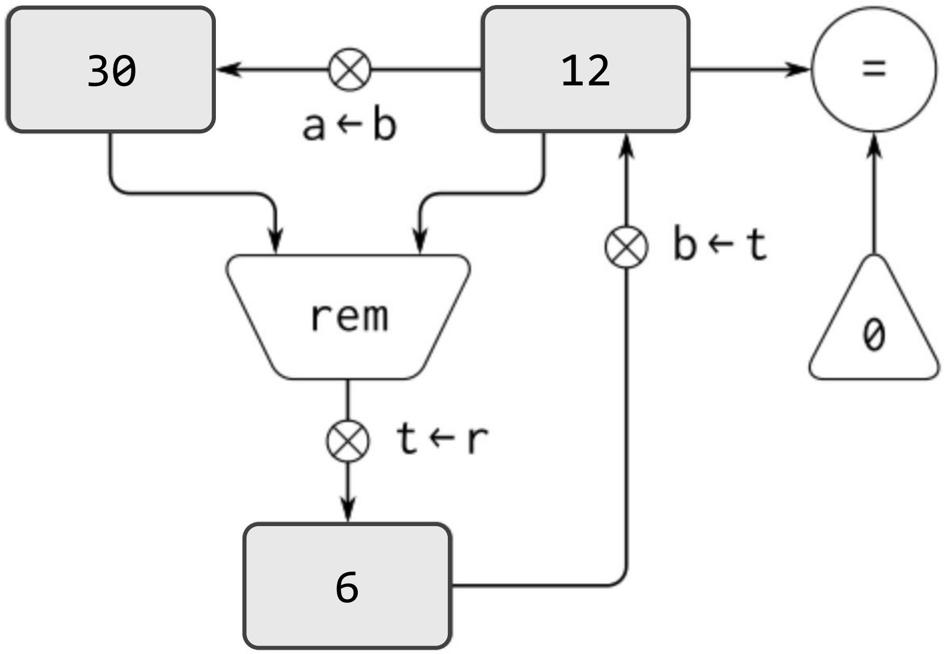


Figure 5.1: Data paths for a GCD machine.

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

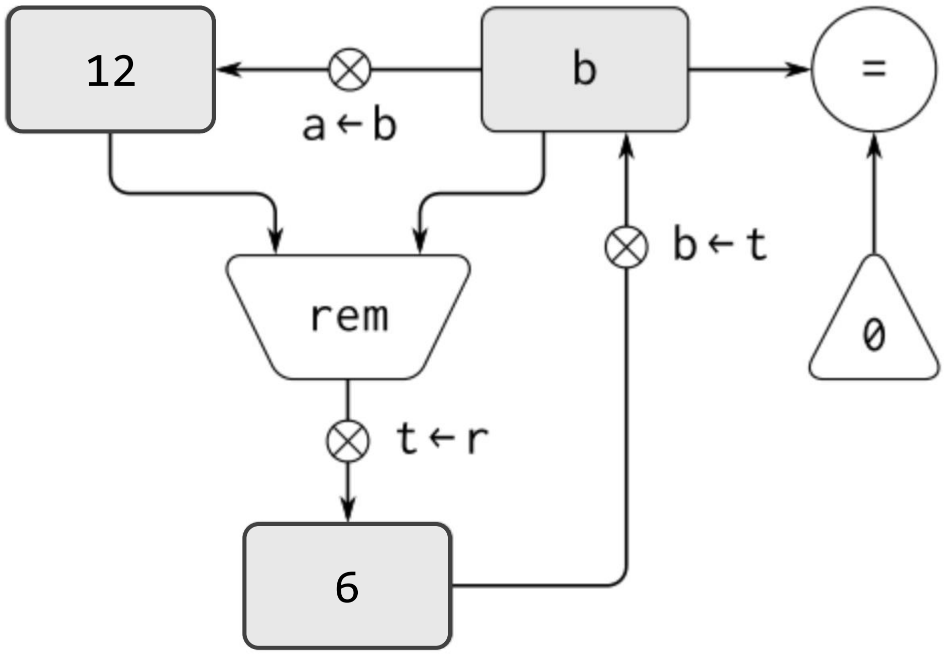


Figure 5.1: Data paths for a GCD machine.

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

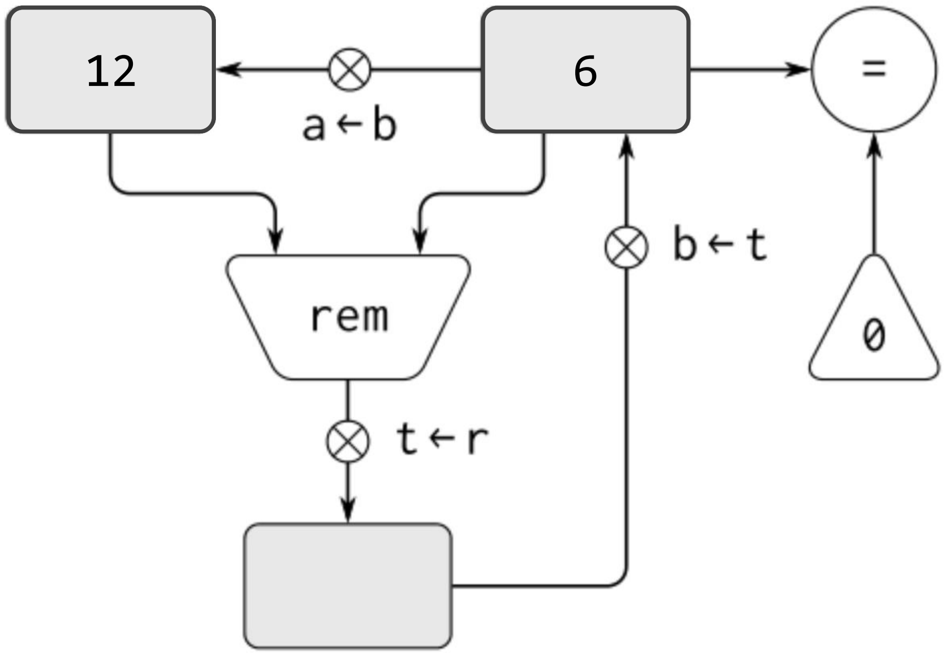


Figure 5.1: Data paths for a GCD machine.

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

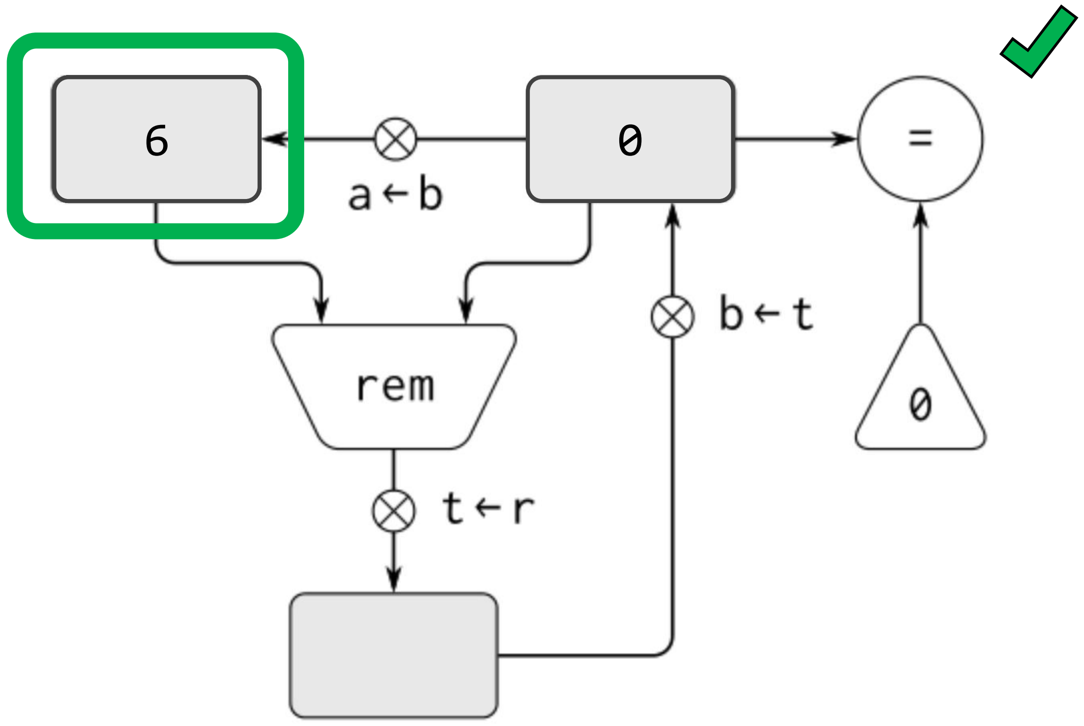


Figure 5.1: Data paths for a GCD machine.

```
;; output
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (label test-b))
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
```

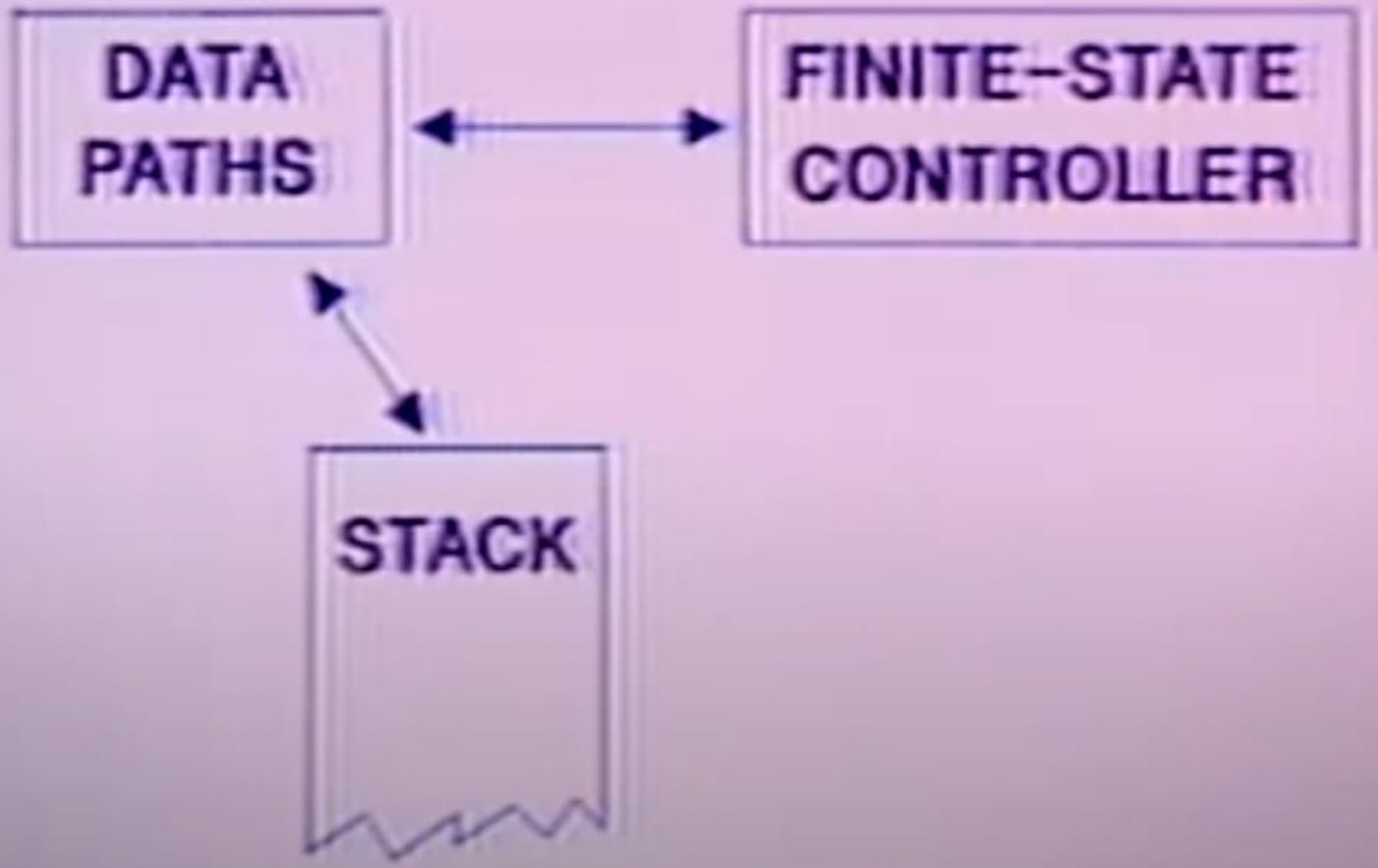
Structure & Interpretation of Computer Programs

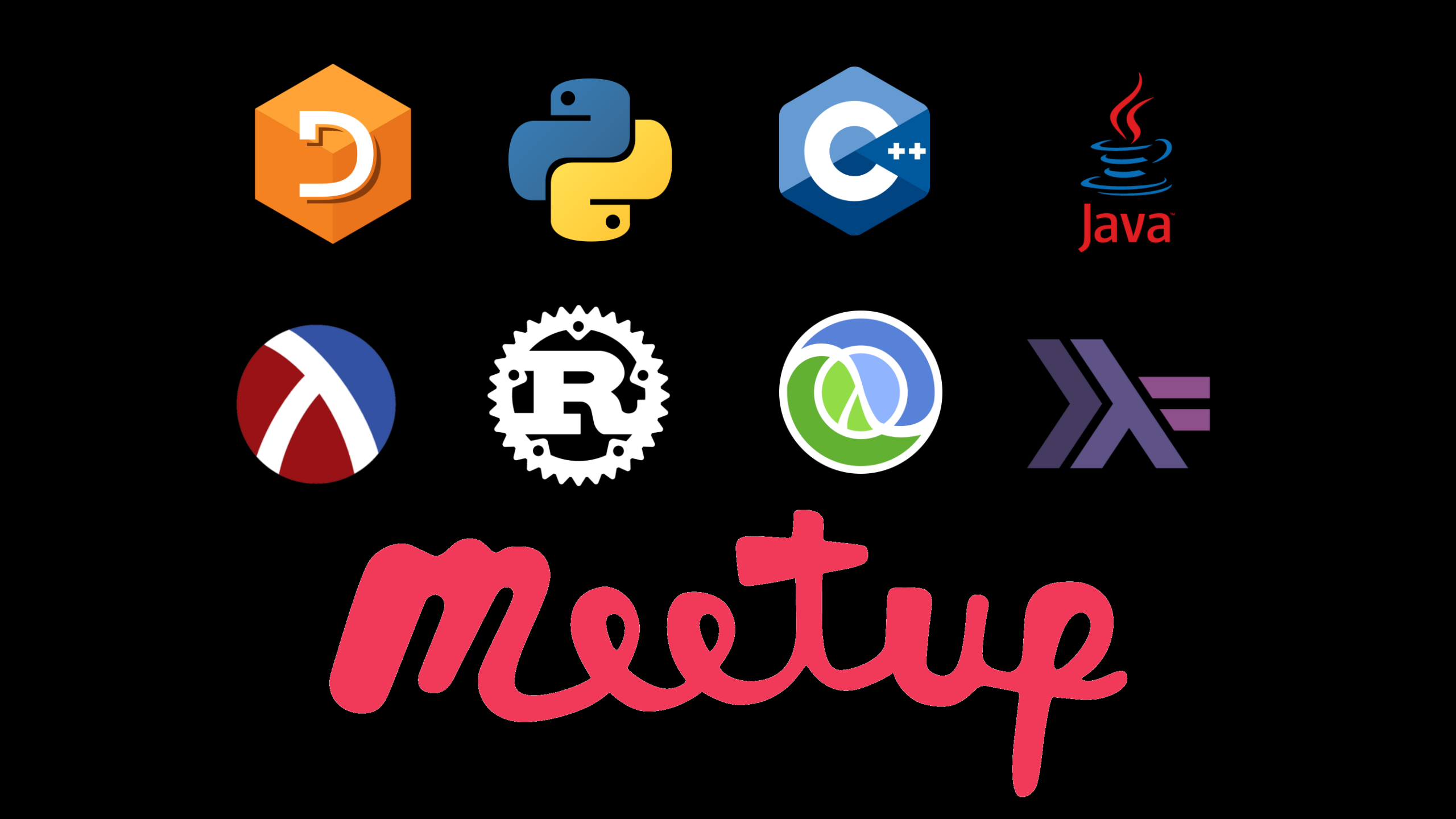
Harold Abelson

Gerald Jay Sussman


Escher Pict.
Digital
Logic
Query
LISP
Metacirc.
LISP
LISP

ORGANIZATION OF REGISTER MACHINE




Java
meetup