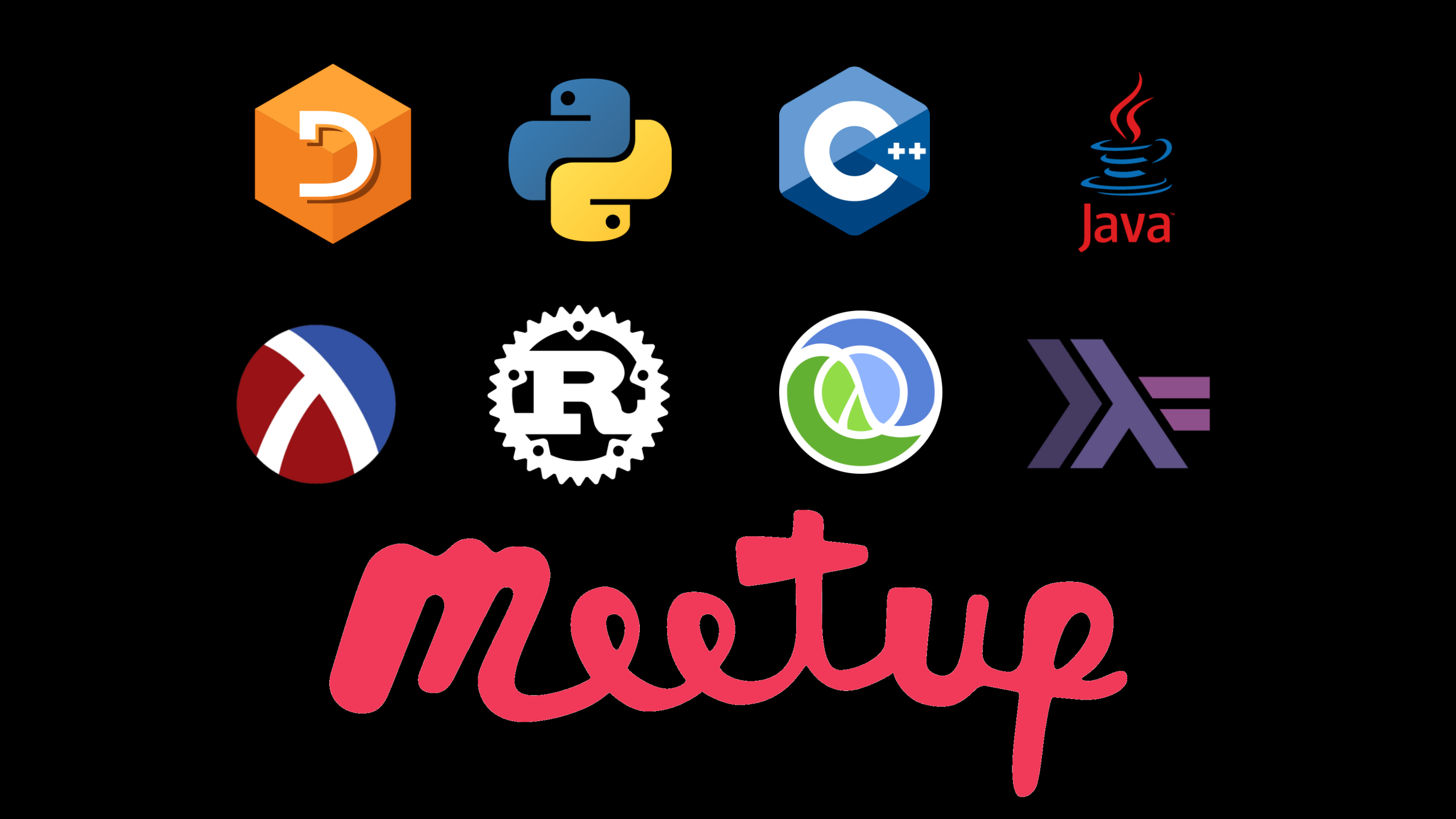
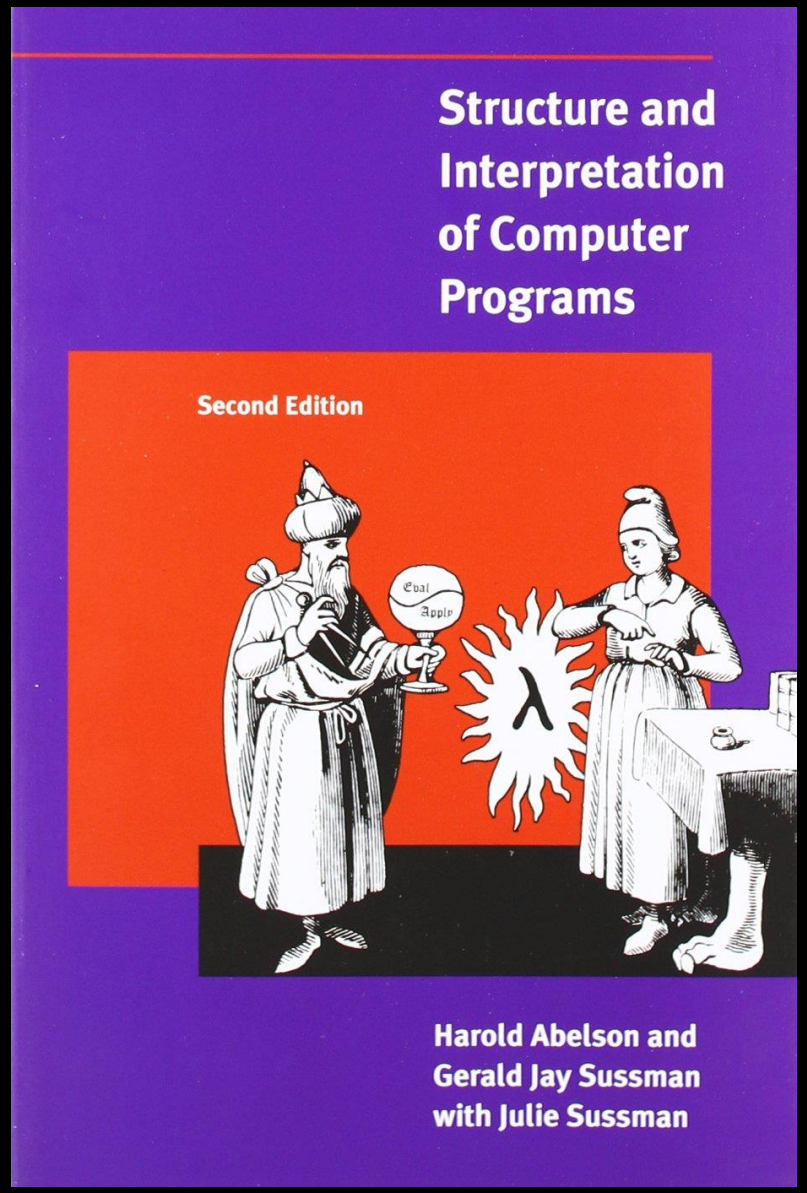

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 5.1

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

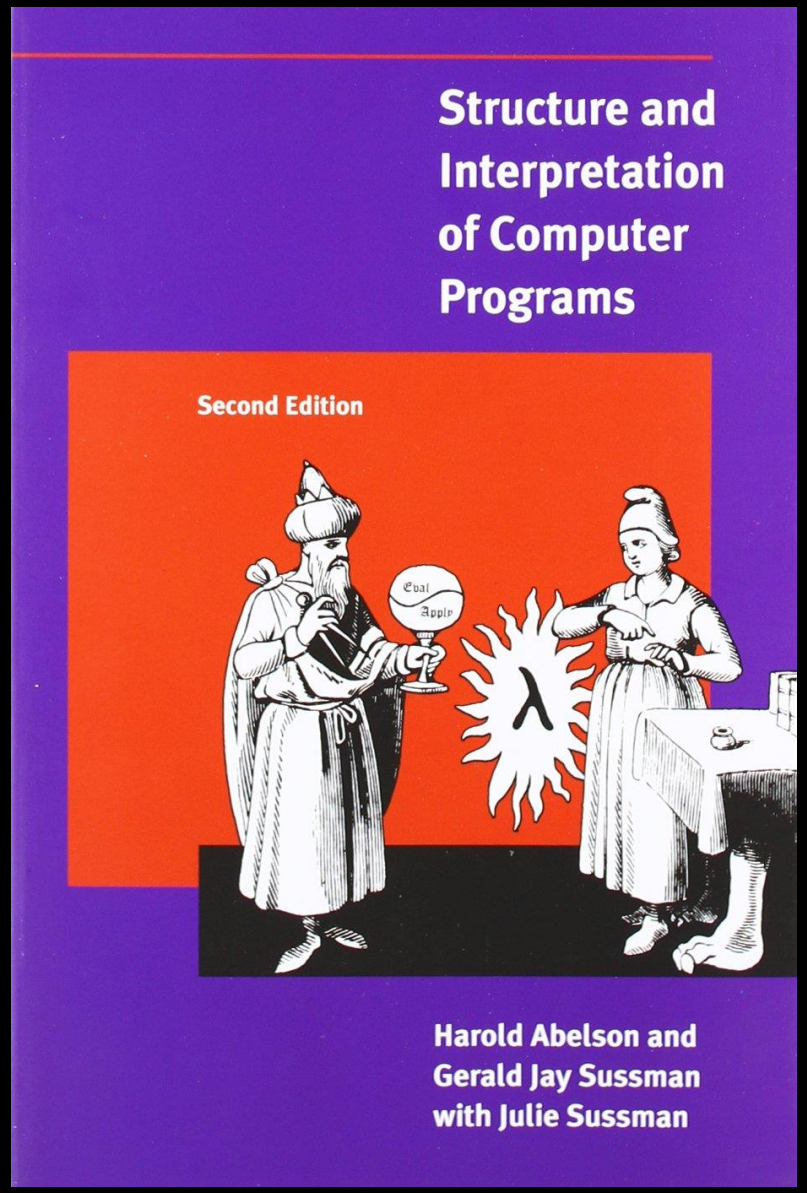
Following

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



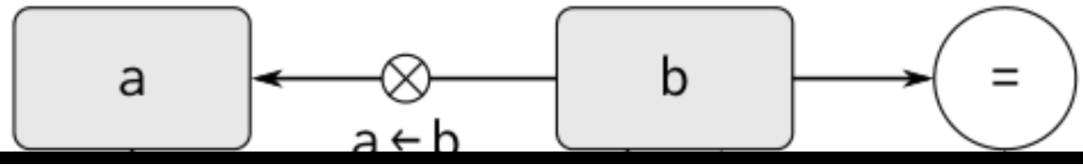
Structure and Interpretation of Computer Programs

Chaper 5.1

We began this book by studying processes and by describing processes in terms of procedures written in Lisp. To explain the meanings of these procedures, we used a succession of models of evaluation: the substitution model of Chapter 1, the environment model of Chapter 3, and the metacircular evaluator of Chapter 4. Our examination of the metacircular evaluator, in particular, dispelled much of the mystery of how Lisp-like languages are interpreted. But even the metacircular evaluator leaves important questions unanswered, because it fails to elucidate the mechanisms of control in a Lisp system

In this chapter we will describe processes in terms of the step-by-step operation of a traditional computer. Such a computer, or *register machine*, sequentially executes *instructions* that manipulate the contents of a fixed set of storage elements called *registers*. A typical register-machine instruction applies a primitive operation to the contents of some registers and assigns the result to another register. Our descriptions of processes executed by register machines will look very much like "machine-language" programs for traditional computers. However, instead of focusing on the machine language of any particular computer, we will examine several Lisp procedures and design a specific register machine to execute each procedure.

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```



register a). The source of data for a register can be another register (as in the a←b assignment), an operation result (as in the t←r assignment), or a constant (a built-in value that cannot be changed, represented in a data-path diagram by a triangle containing the constant).

An operation that computes a value from constants and the contents of registers is represented in a data-path diagram by a trapezoid containing a name for the operation. For example, the box marked rem

Figure 5.1: Data paths for a GCD machine.

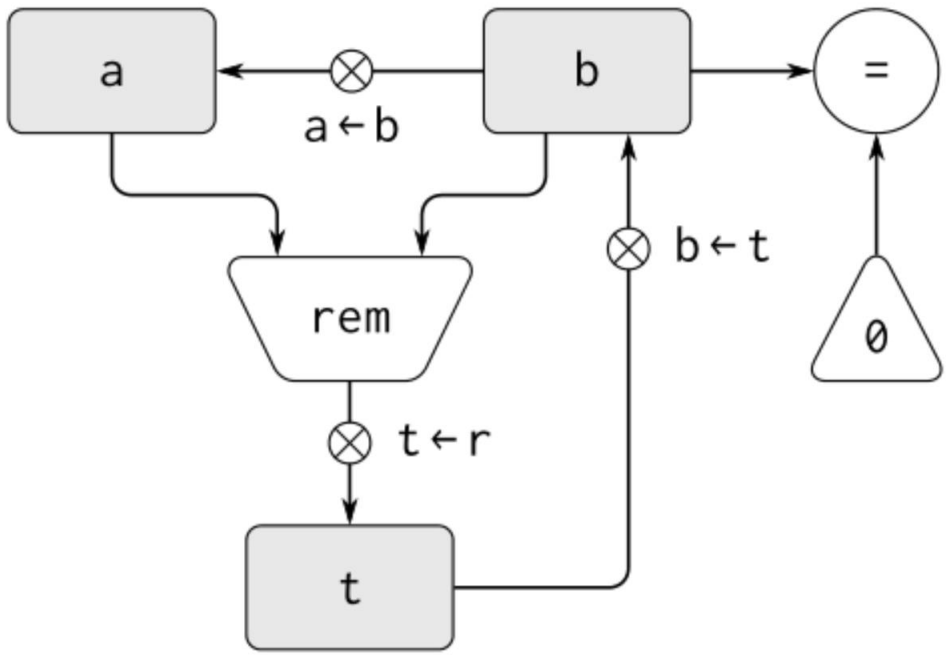


Figure 5.1: Data paths for a GCD machine.

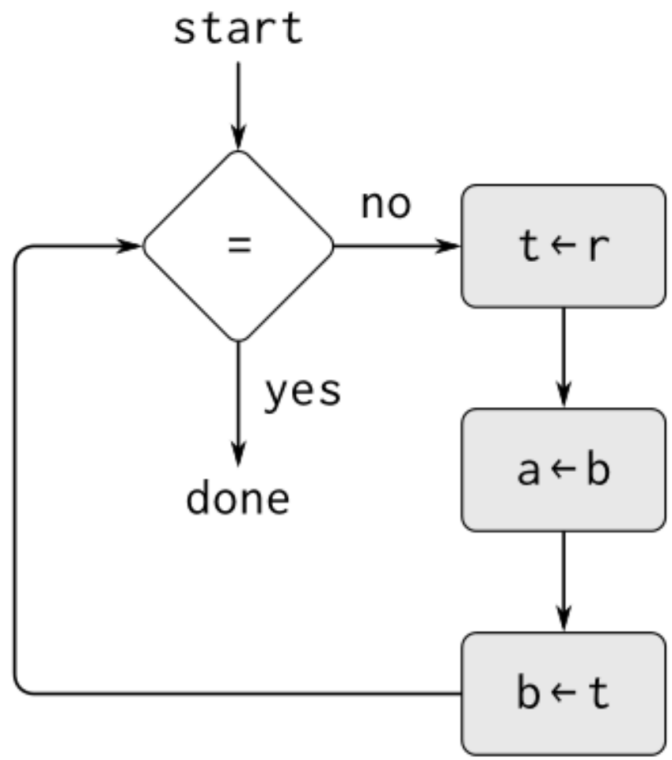


Figure 5.2: Controller for a GCD machine.

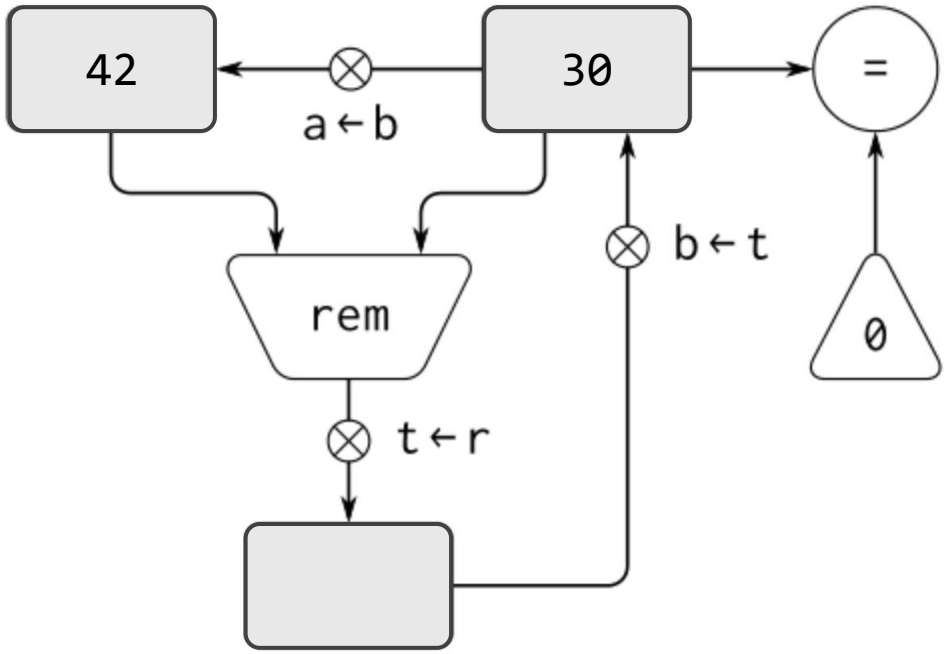


Figure 5.1: Data paths for a GCD machine.

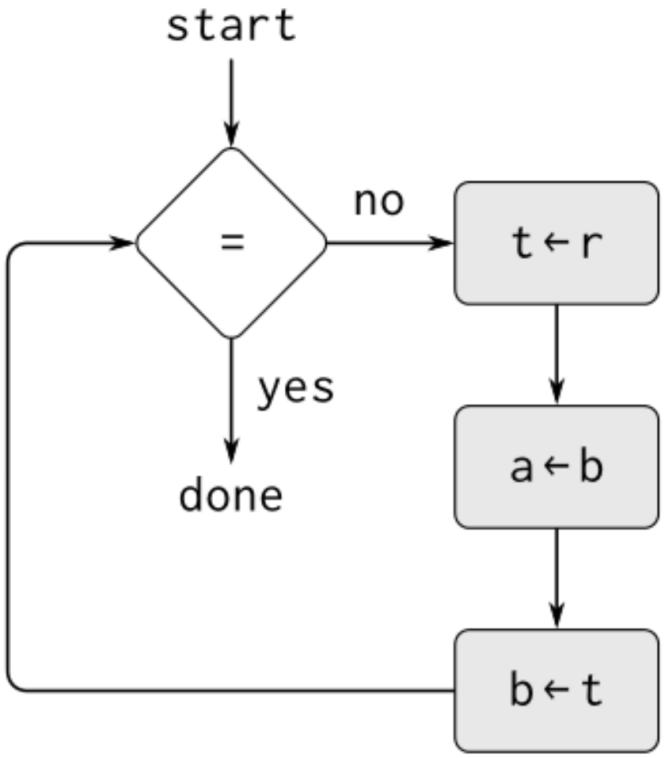


Figure 5.2: Controller for a GCD machine.

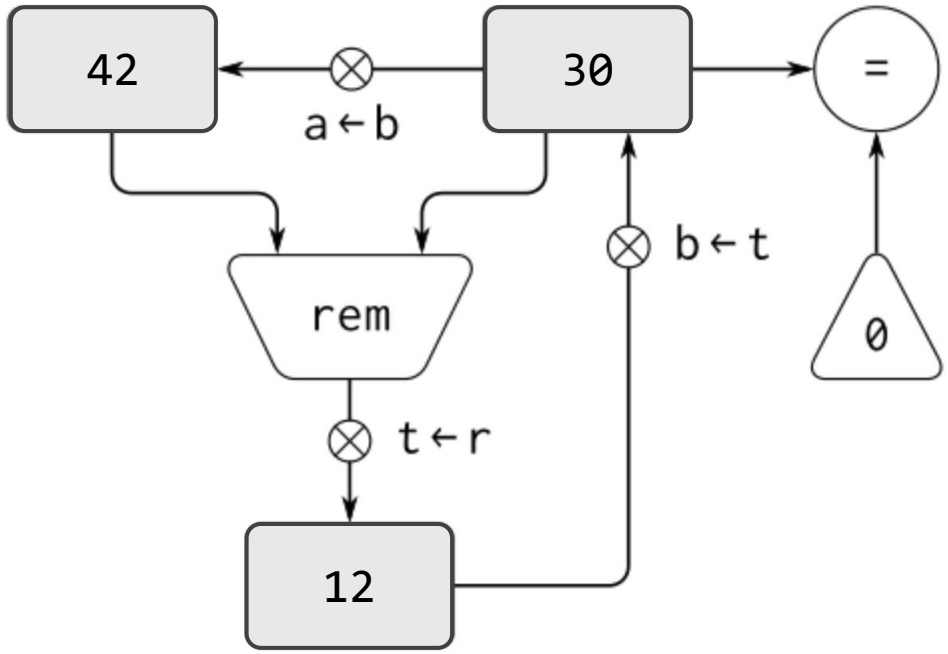


Figure 5.1: Data paths for a GCD machine.

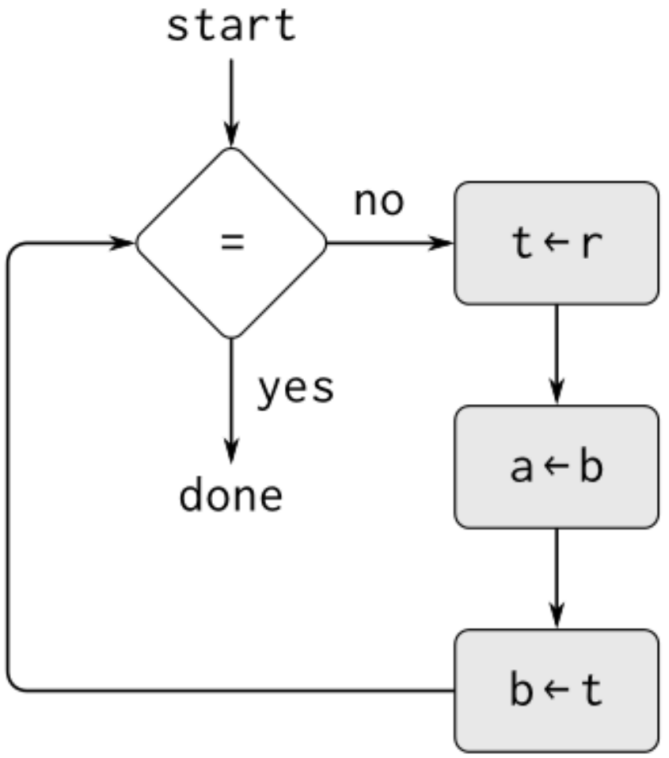


Figure 5.2: Controller for a GCD machine.

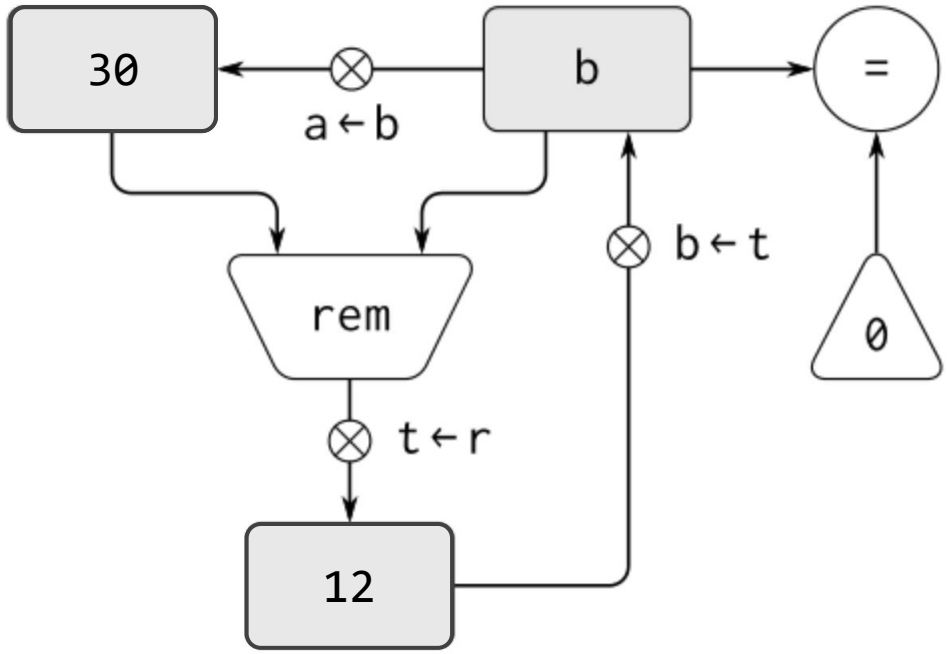


Figure 5.1: Data paths for a GCD machine.

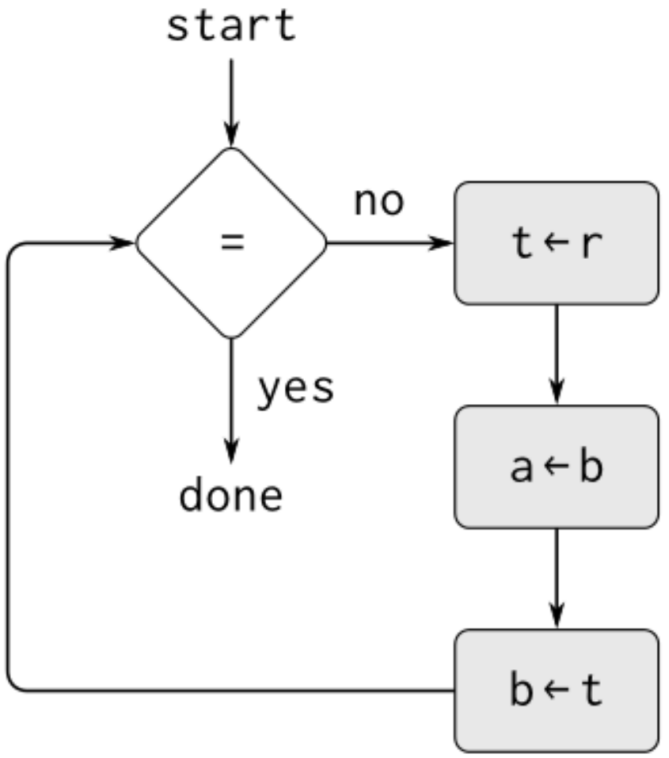


Figure 5.2: Controller for a GCD machine.

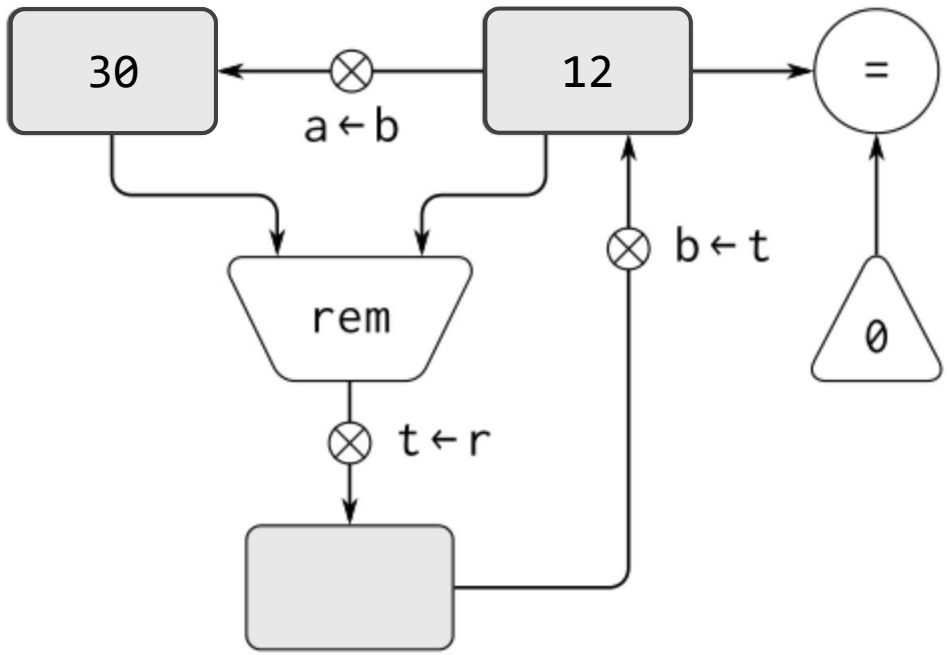


Figure 5.1: Data paths for a GCD machine.

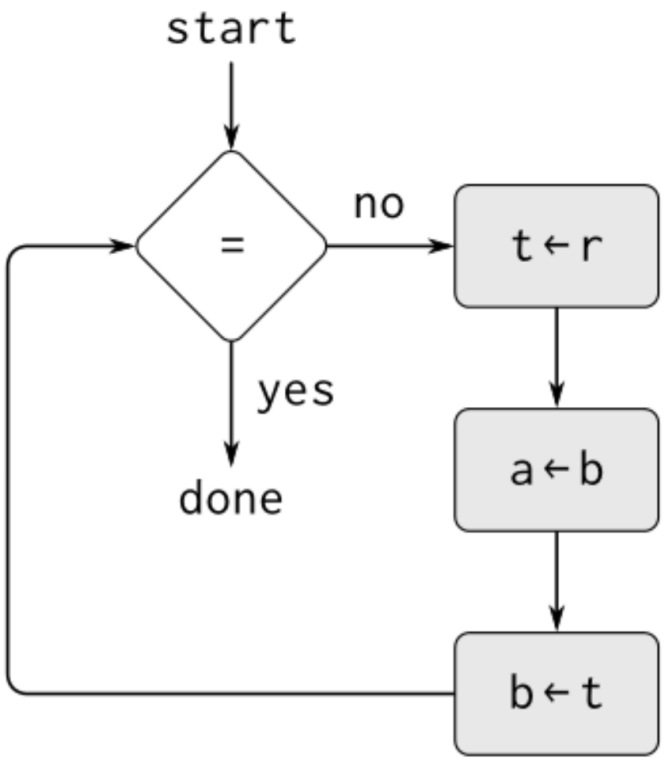


Figure 5.2: Controller for a GCD machine.

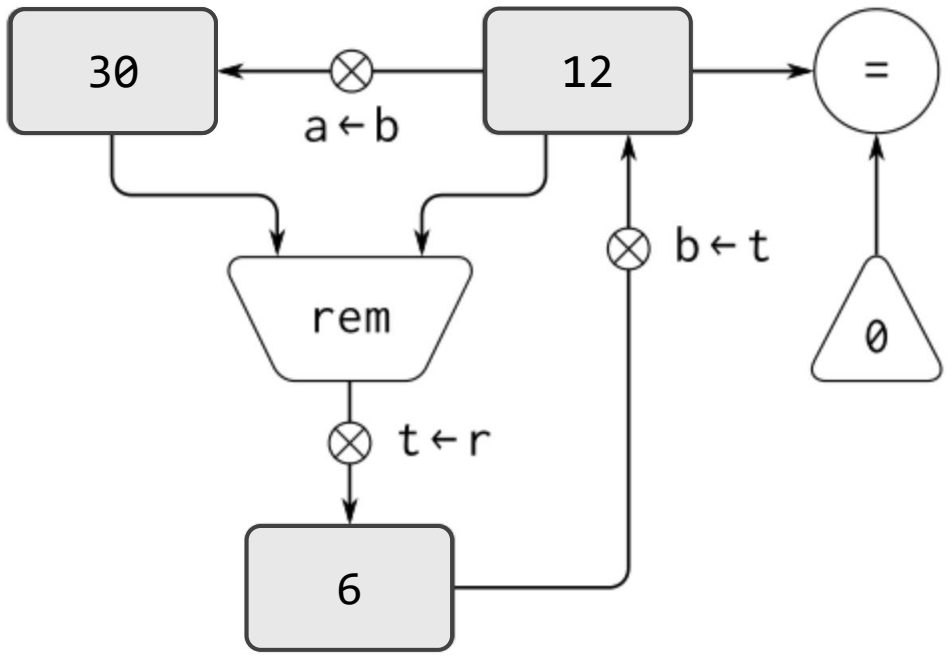


Figure 5.1: Data paths for a GCD machine.

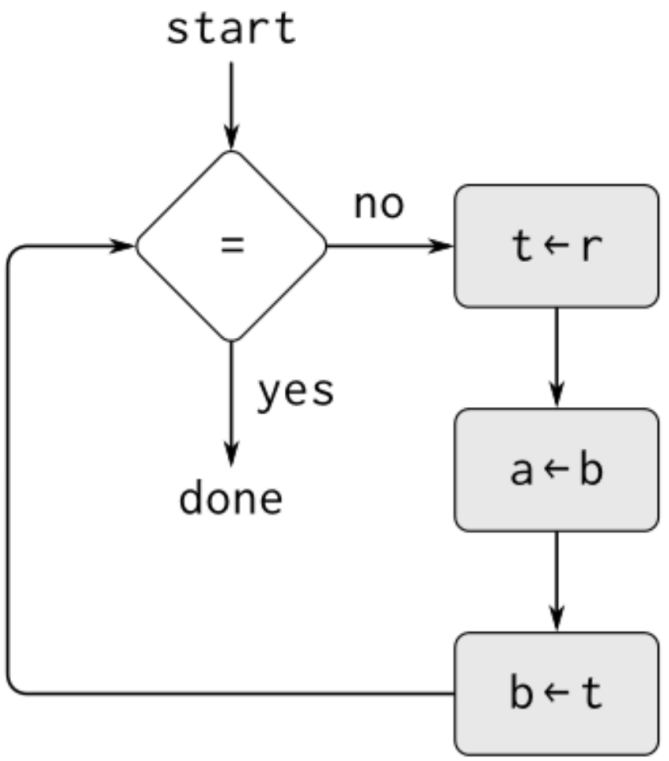


Figure 5.2: Controller for a GCD machine.

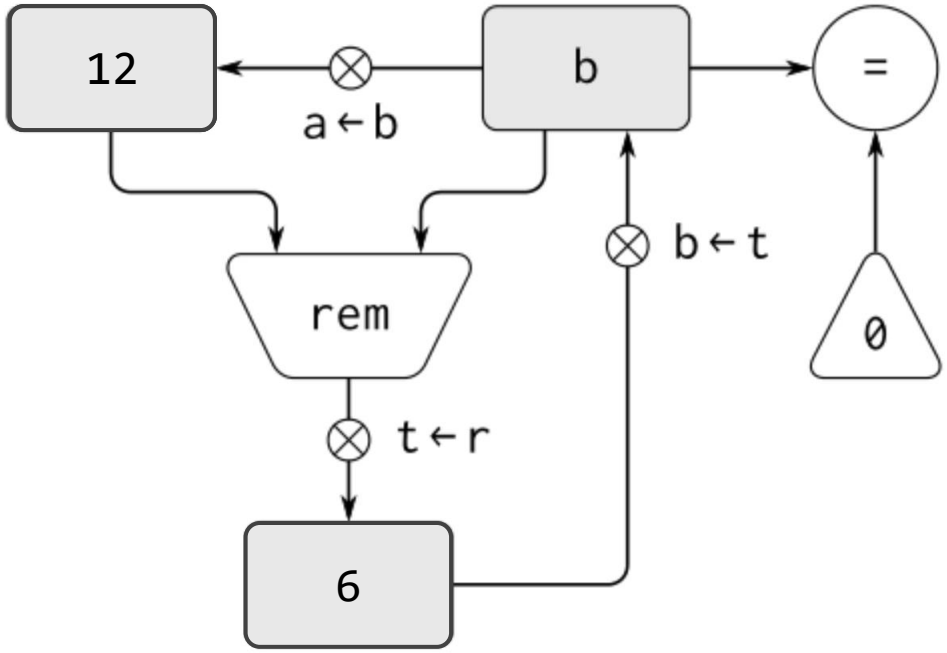


Figure 5.1: Data paths for a GCD machine.

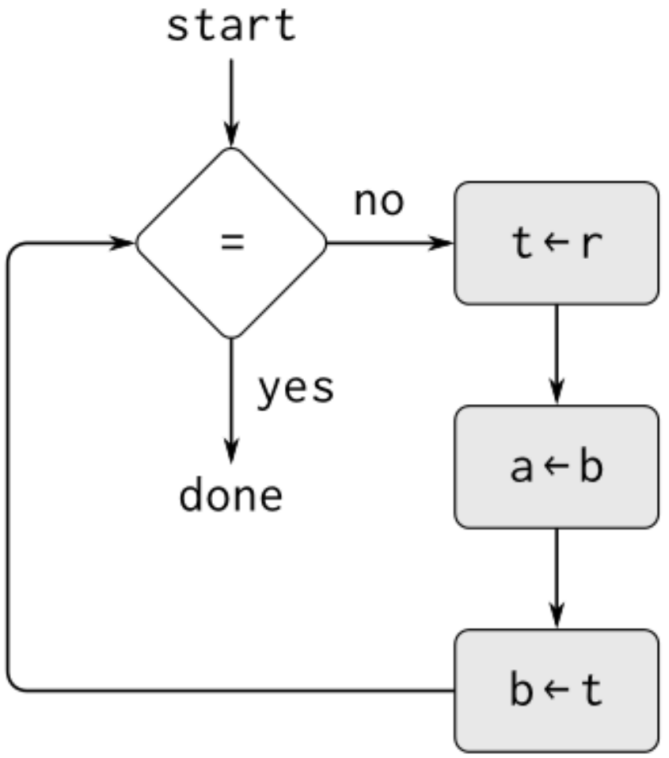


Figure 5.2: Controller for a GCD machine.

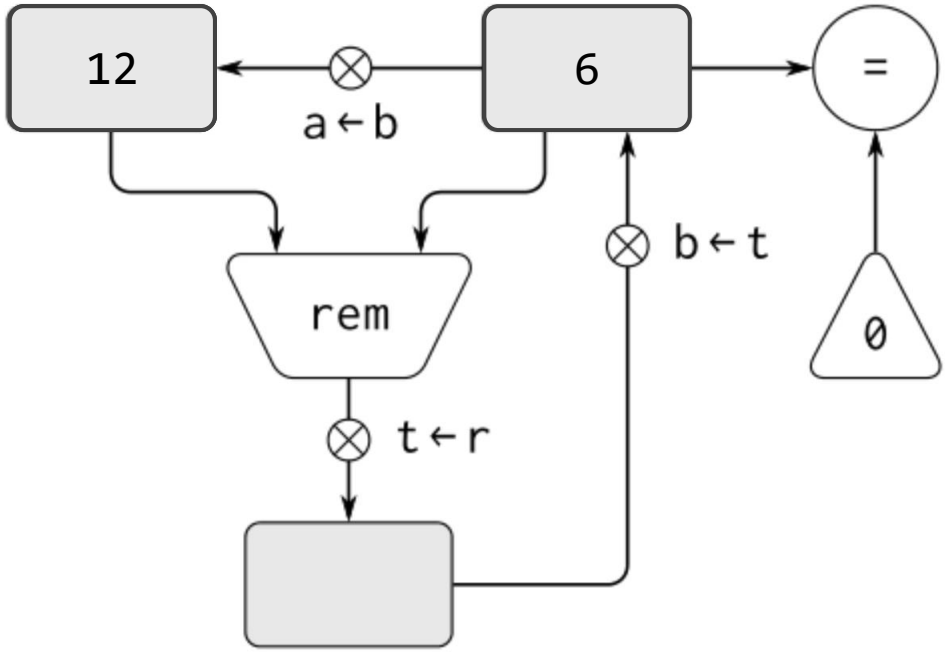


Figure 5.1: Data paths for a GCD machine.

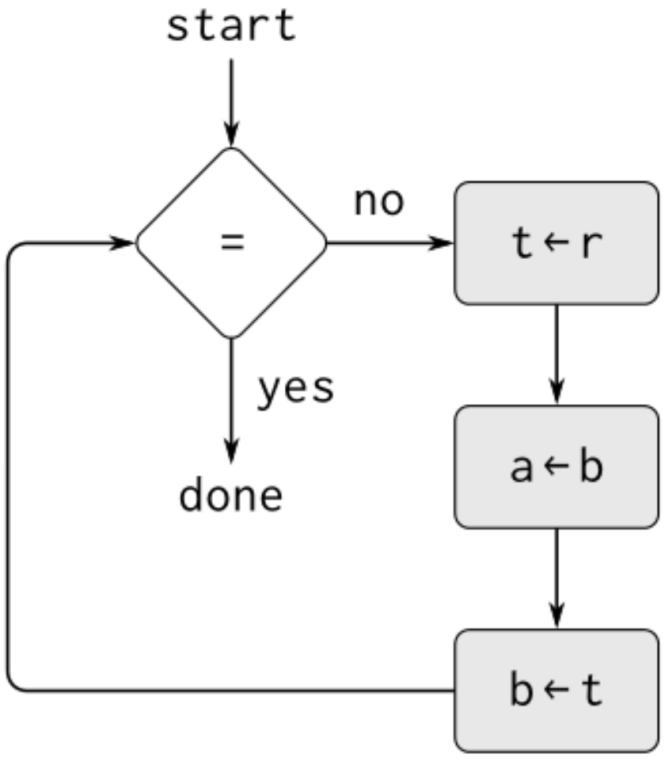


Figure 5.2: Controller for a GCD machine.

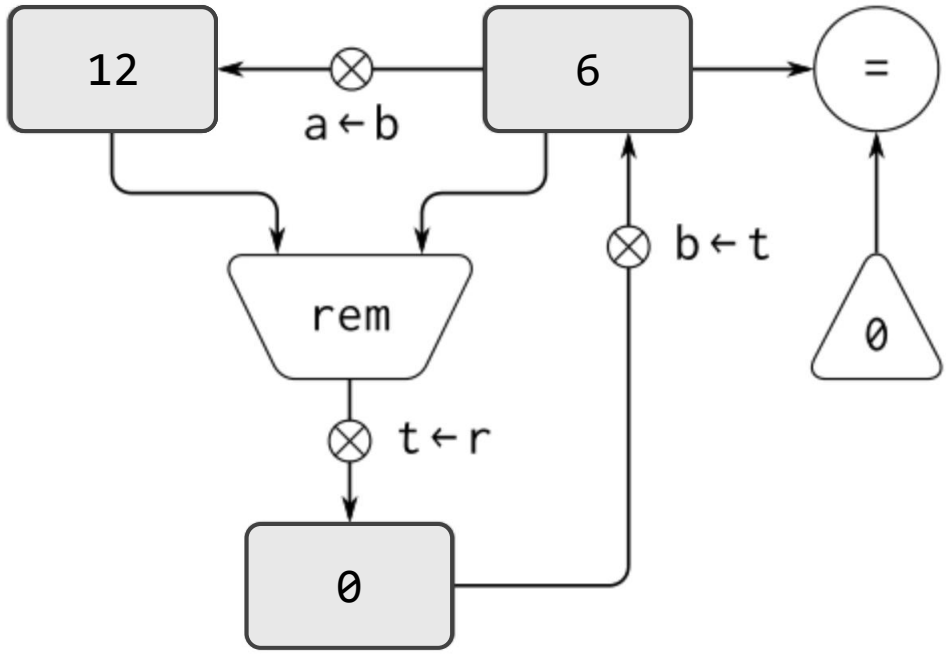


Figure 5.1: Data paths for a GCD machine.

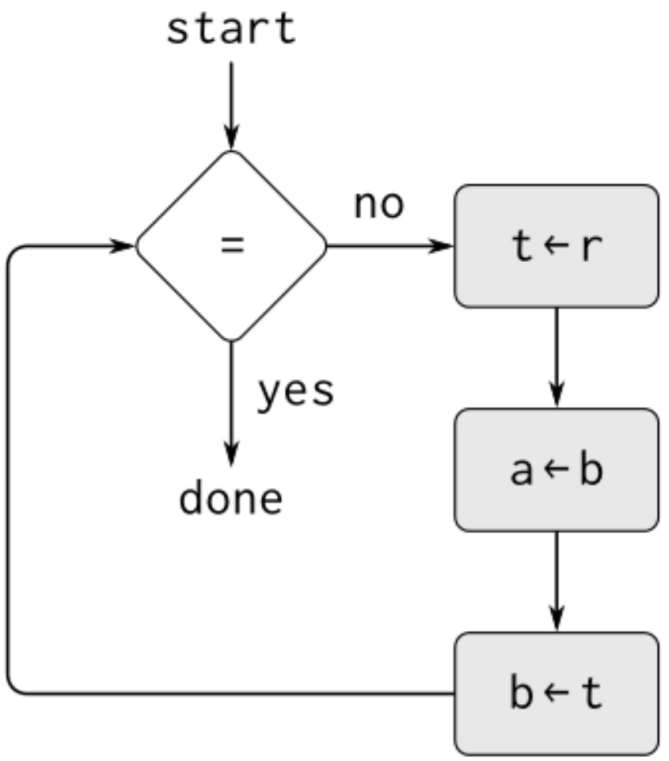


Figure 5.2: Controller for a GCD machine.

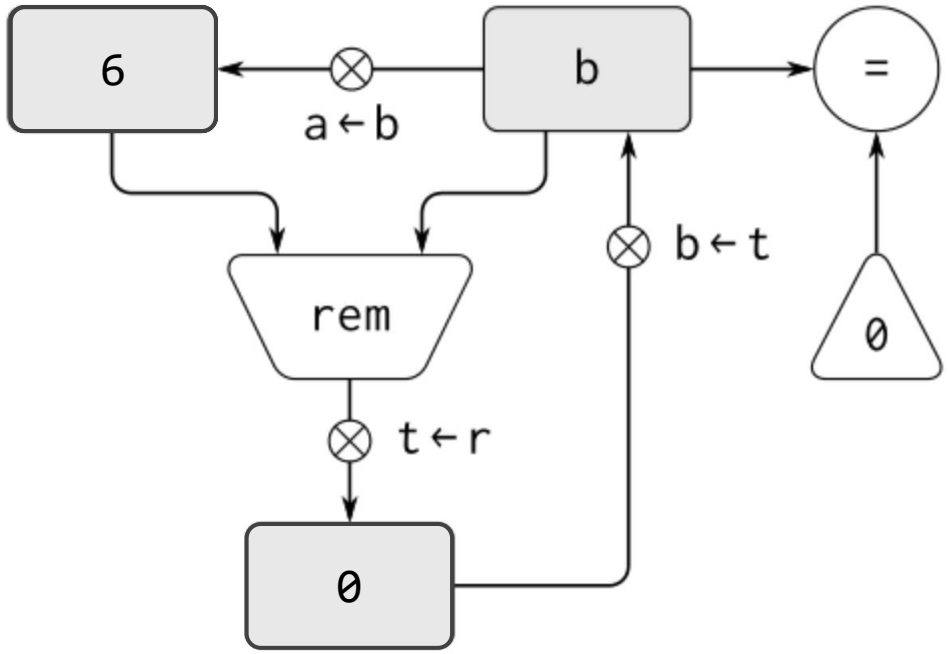


Figure 5.1: Data paths for a GCD machine.

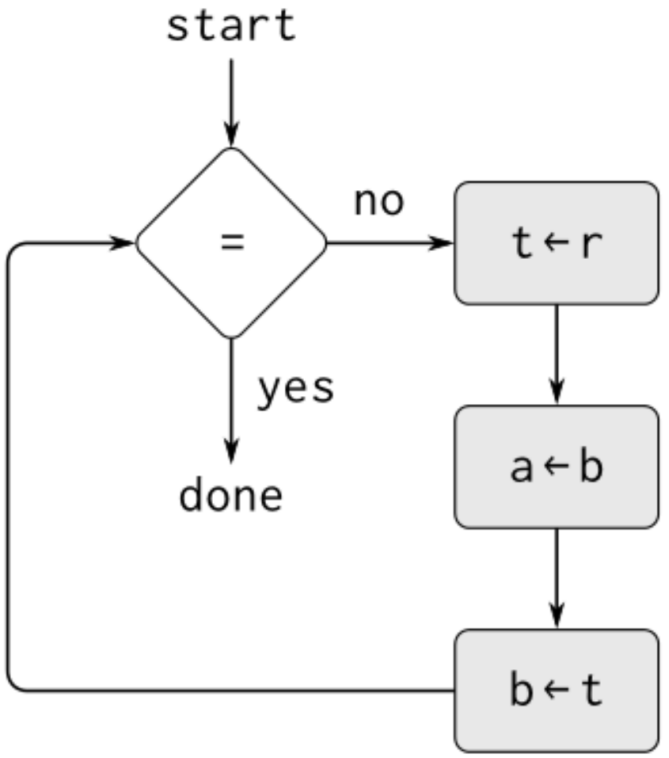


Figure 5.2: Controller for a GCD machine.

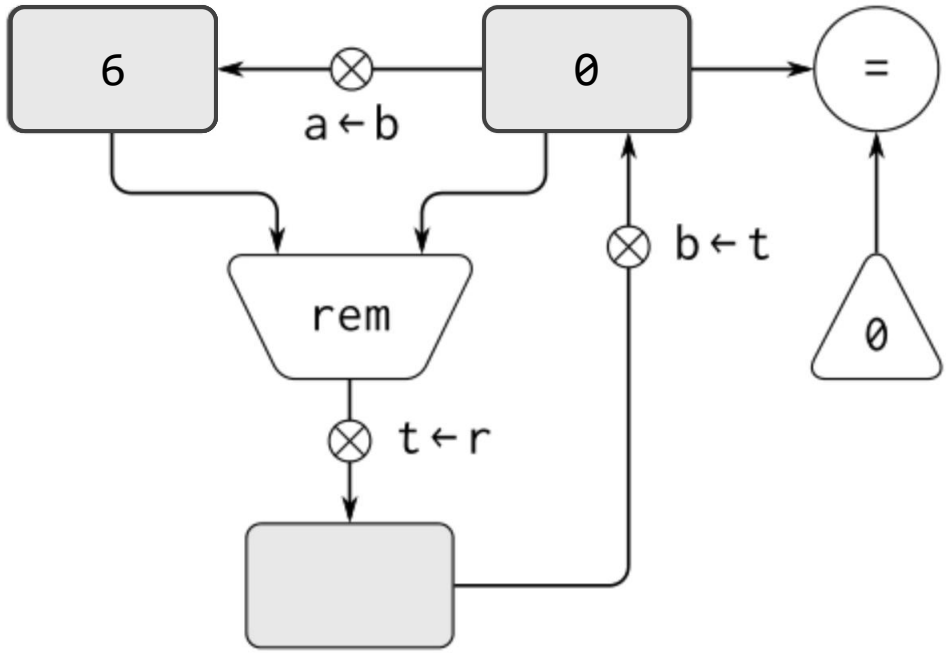


Figure 5.1: Data paths for a GCD machine.

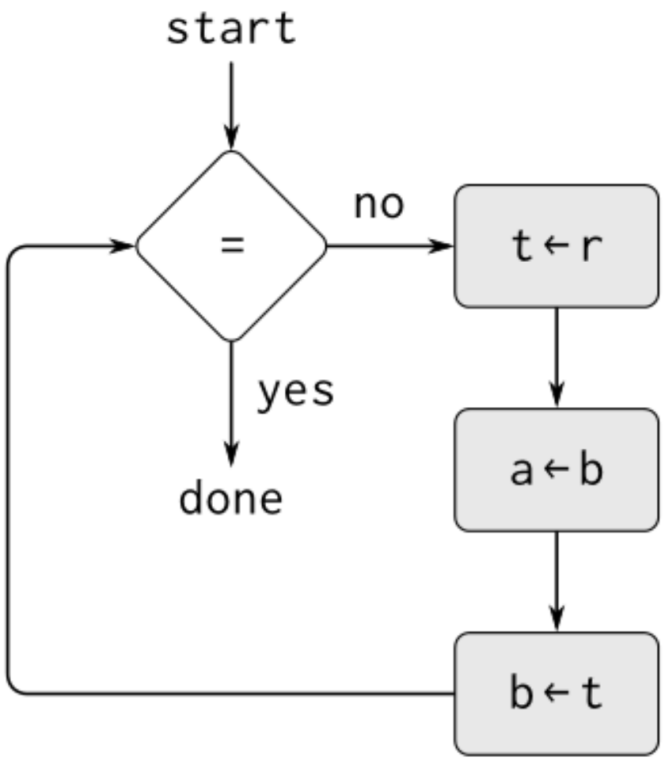


Figure 5.2: Controller for a GCD machine.

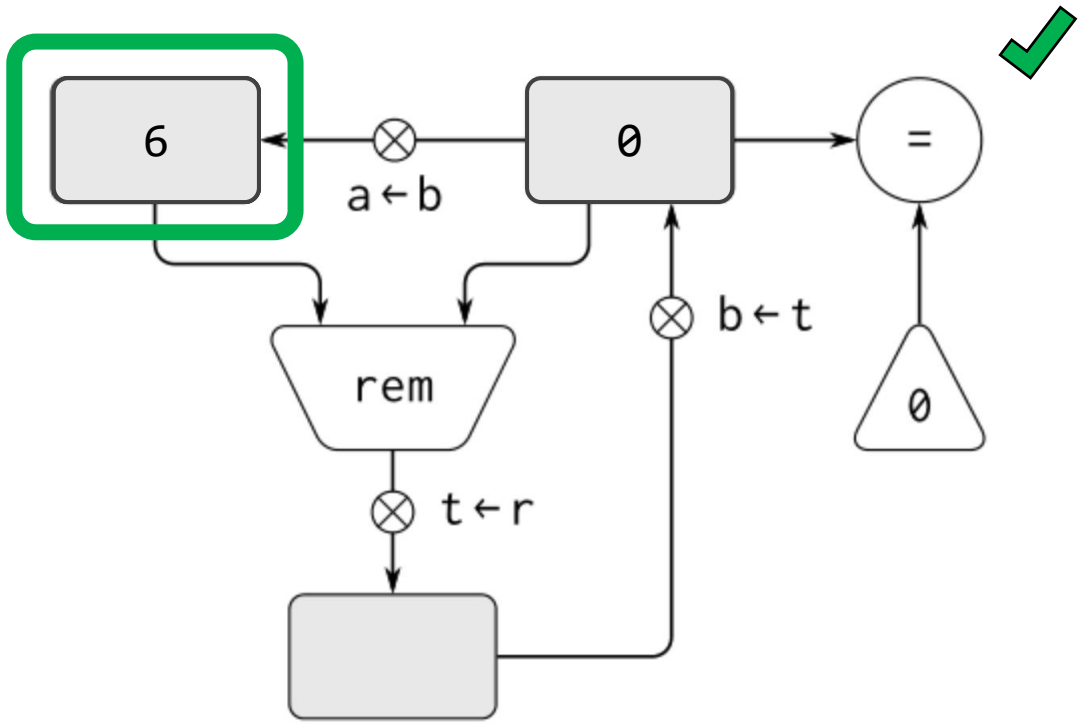


Figure 5.1: Data paths for a GCD machine.

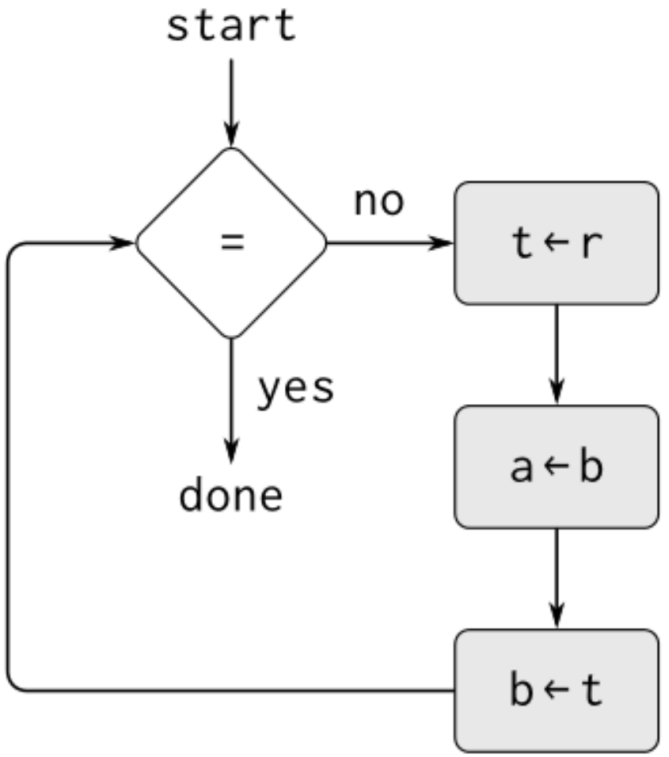
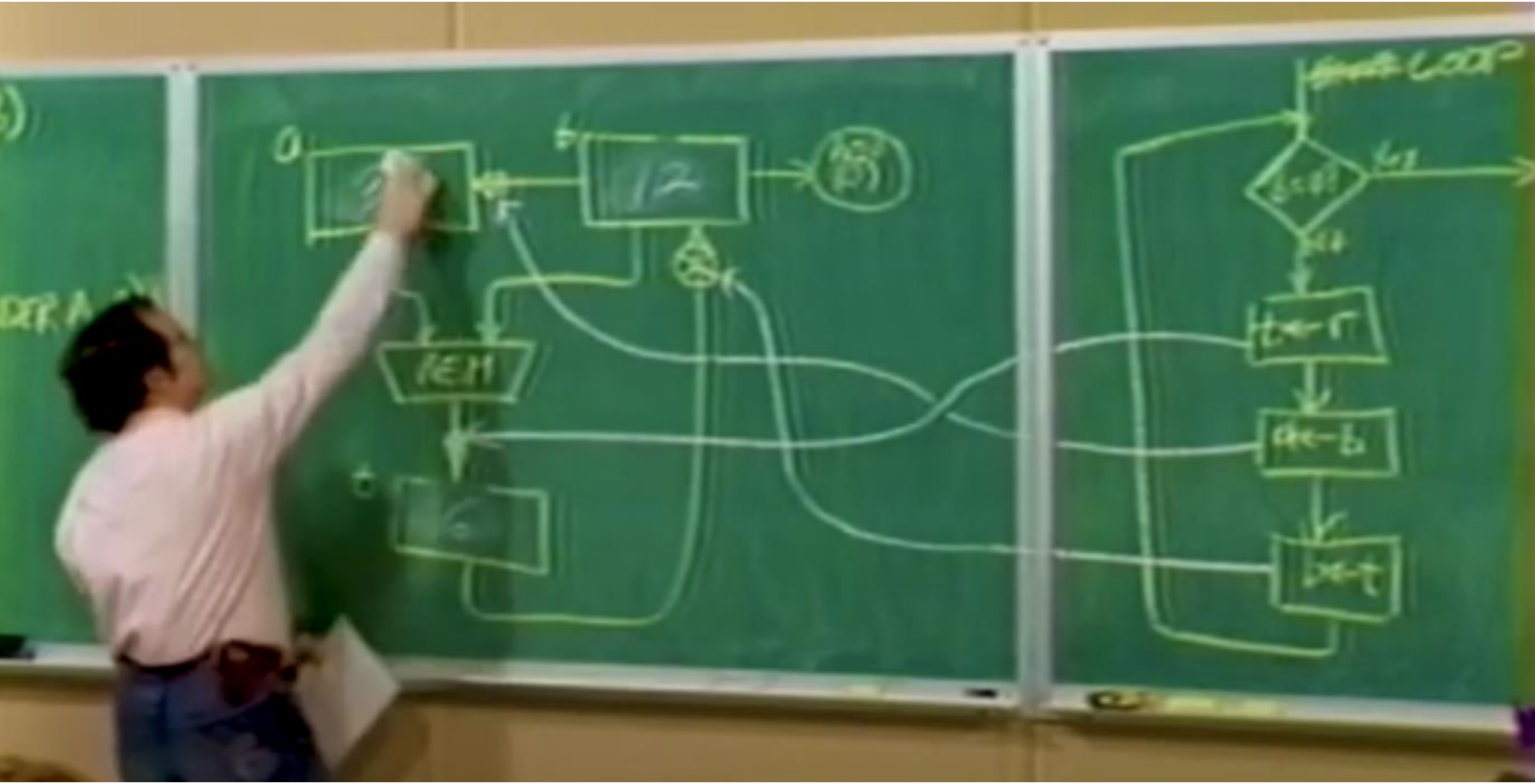
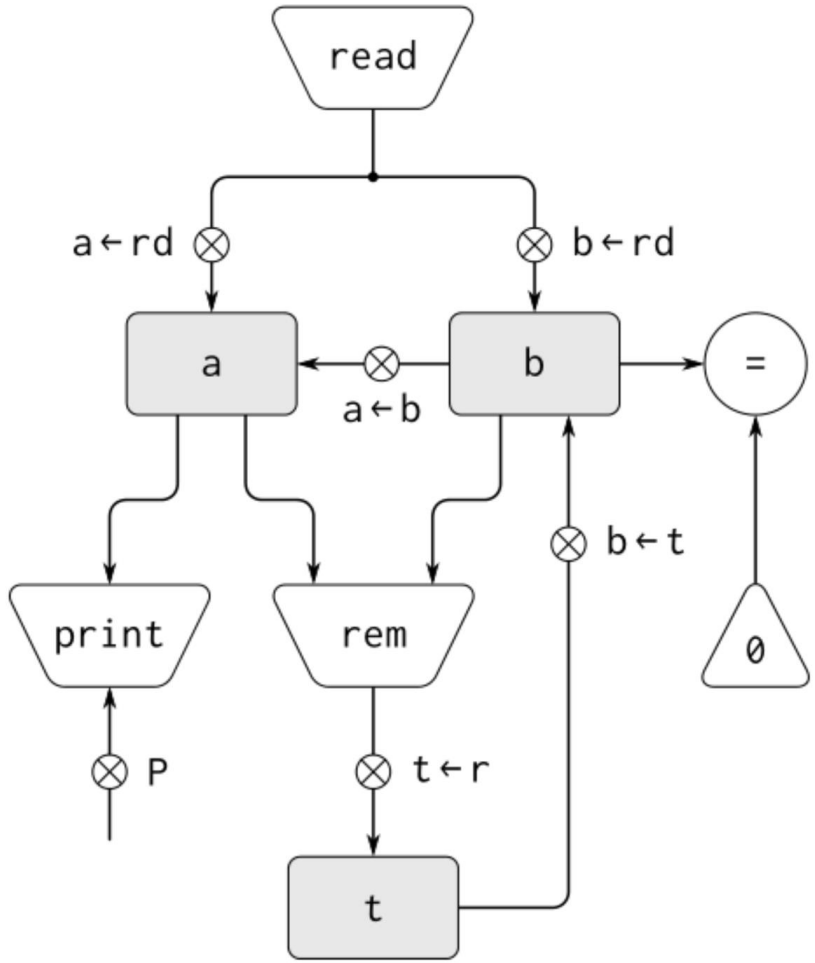


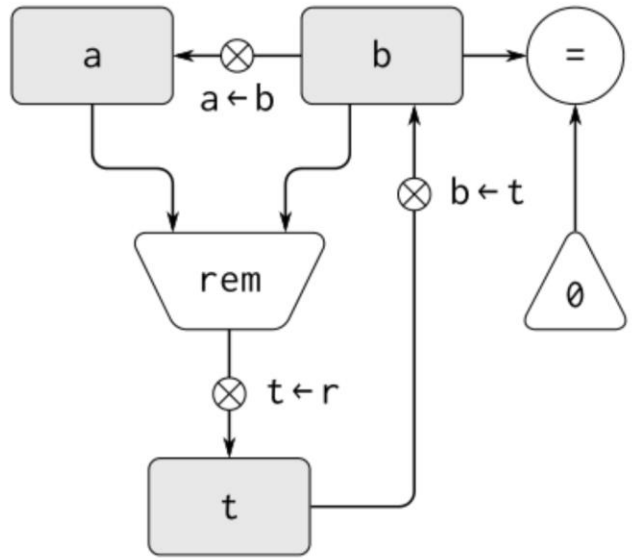
Figure 5.2: Controller for a GCD machine.

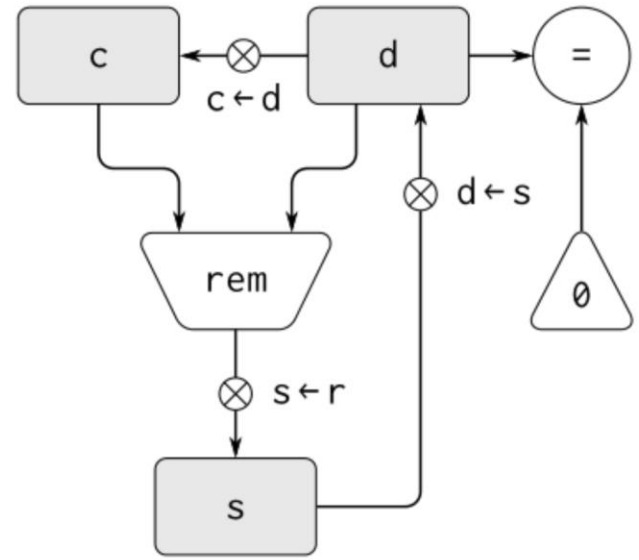

12
MEM




```
(controller
 gcd-loop
   (assign a (op read))
   (assign b (op read))
 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
   (perform (op print)
            (reg a))
   (goto (label gcd-loop)))
```

Figure 5.4: A GCD machine that reads inputs and prints results.






```
gcd-1
 (test (op =) (reg b) (const 0))
 (branch (label after-gcd-1))
 (assign t (op rem) (reg a) (reg b))
 (assign a (reg b))
 (assign b (reg t))
 (goto (label gcd-1))
after-gcd-1
```

```
gcd-2
 (test (op =) (reg d) (const 0))
 (branch (label after-gcd-2))
 (assign s (op rem) (reg c) (reg d))
 (assign c (reg d))
 (assign d (reg s))
 (goto (label gcd-2))
after-gcd-2
```

Figure 5.7: Portions of the data paths and controller sequence for a machine with two GCD computations.

Figure 5.8: ↓ Portions of the controller sequence for a machine that uses the same data-path components for two different GCD computations.

```
gcd-1
 (test (op =) (reg b) (const 0))
 (branch (label after-gcd-1))
 (assign t (op rem) (reg a) (reg b))
 (assign a (reg b))
 (assign b (reg t))
 (goto (label gcd-1))
after-gcd-1
  ...
gcd-2
 (test (op =) (reg b) (const 0))
 (branch (label after-gcd-2))
 (assign t (op rem) (reg a) (reg b))
 (assign a (reg b))
 (assign b (reg t))
 (goto (label gcd-2))
after-gcd-2
```

Figure 5.9: ↓ Using a `continue` register to avoid the duplicate controller sequence in Figure 5.8.

```
gcd
 (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 gcd))
gcd-done
 (test (op =) (reg continue) (const 0))
 (branch (label after-gcd-1))
 (goto (label after-gcd-2))
  ...
 ;; Before branching to gcd from the first place where
 ;; it is needed, we place 0 in the continue register
 (assign continue (const 0))
 (goto (label gcd))
after-gcd-1
  ...

 ;; Before the second use of gcd, we place 1
 ;; in the continue register
 (assign continue (const 1))
 (goto (label gcd))
after-gcd-2
```

```
(assign ⟨register-name⟩ (reg ⟨register-name⟩))
(assign ⟨register-name⟩ (const ⟨constant-value⟩))
(assign ⟨register-name⟩
        (op ⟨operation-name⟩)
        ⟨input_1⟩ ... ⟨input_n⟩)
(perform (op ⟨operation-name⟩) ⟨input_1⟩ ... ⟨input_n⟩)
(test (op ⟨operation-name⟩) ⟨input_1⟩ ... ⟨input_n⟩)
(branch (label ⟨label-name⟩))
(goto (label ⟨label-name⟩))
```

The use of registers to hold labels was introduced in Section 5.1.3:

```
(assign ⟨register-name⟩ (label ⟨label-name⟩))
(goto (reg ⟨register-name⟩))
```

Instructions to use the stack were introduced in Section 5.1.4:

```
(save ⟨register-name⟩)
(restore ⟨register-name⟩)
```

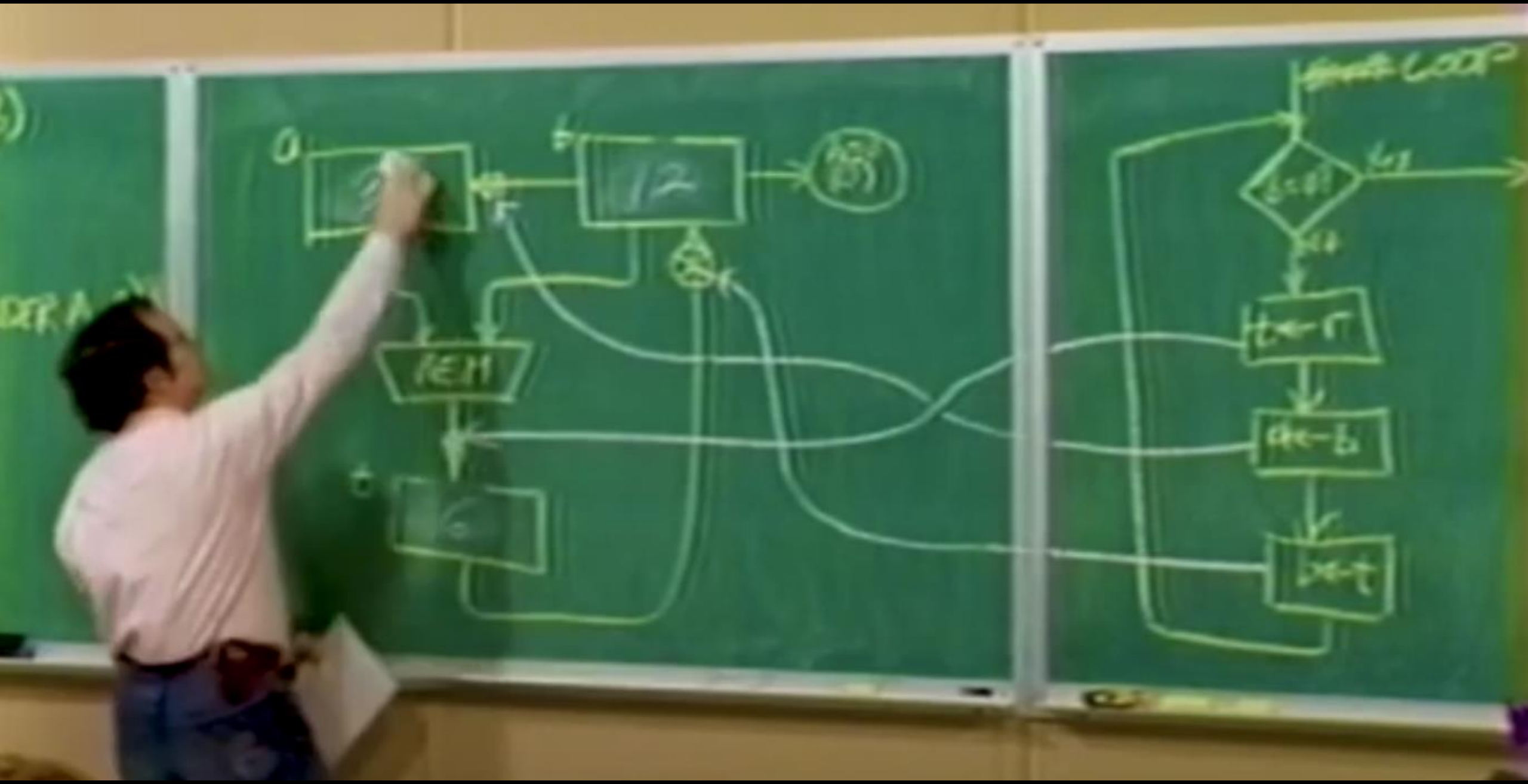
The only kind of ⟨*constant-value*⟩ we have seen so far is a number, but later we will use strings, symbols, and lists. For example,

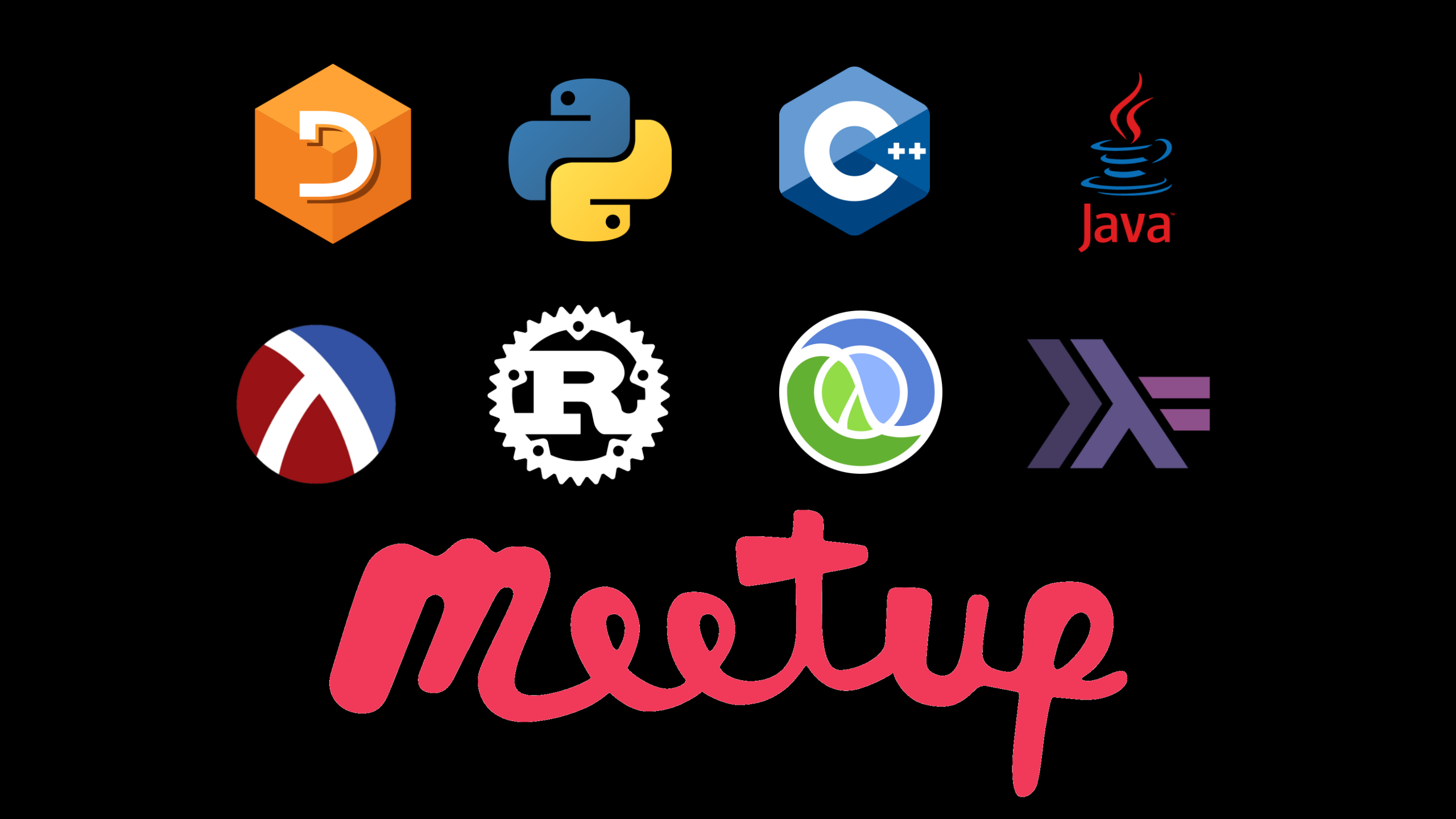
`(const "abc")` is the string `"abc"`,
`(const abc)` is the symbol `abc`,
`(const (a b c))` is the list `(a b c)`,
and `(const ())` is the empty list.

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman


12
MEM
LOOP


Java
meetup