5. 寄存器机器里的计算(I)

本章讨论更底层的计算，以及抽象层程序（Scheme 程序）到底层程序的翻译

本节课的内容：

- 寄存器机器
- 描述寄存器机器的计算过程
- 寄存器机器语言
- 子程序和递归带来的问题
- 寄存器机器语言的模拟器（解释器）
- 模拟器的实现

求值器的控制和寄存器机器

- 前面研究了计算和用 Lisp 过程描述计算的相关问题。通过若干求值模型解释过程的意义：代换模型，环境模型，元循环模型
- 元循环模型表现出求值过程的许多细节，但仍有些遗漏，主要是没解释 Lisp 系统里的控制。例如
  - 求值子表达式后，如何把值送给使用值的表达式？
  - 为什么有些递归过程产生迭代型计算过程（只需常量空间），而另一些却产生递归型计算过程（需要线性以上的空间）？

  原因：求值器本身是 Lisp 程序，继承并利用了基础系统的结构
- 要进一步理解 Lisp 求值器的控制，必须转到更低的层面
- 本章基于常规计算机（寄存器机器）的基本操作描述计算，这种计算机的特征是能顺序执行一条条指令，操作一组称为寄存器的存储单元
- 典型指令：把一个操作用于几个寄存器的内容，将操作结果存入某寄存器。计算过程的这种描述很像传统的机器指令程序

寄存器机器

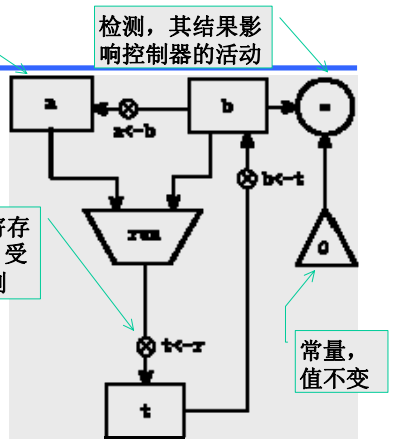
- 这里不涉及具体机器，仍是研究一些 Lisp 过程，并考虑为每个过程设计一部特殊的寄存器机器
- 初始工作就像设计硬件体系结构：开发出一些机制支持各种重要的程序结构，如递归、过程调用等；还要设计一种描述寄存器机器的语言，然后做一个 Lisp 程序来解释用这种语言描述的机器
- 机器中的大部分操作都很简单，可以由简单硬件执行
- 进一步需要是处理表结构。为此就需要实现基本的表操作和巧妙的存储管理机制。后面讨论有关的基础技术
- 有了语言和存储管理，就可以做一个机器，实现前面的元循环解释器。它能为解释器的细节提供一个清晰模型
- 本章最后讨论实现一个编译器，把 Scheme 语言程序翻译到这里的寄存器机器语言。该系统可以支持解释代码和编译代码之间的连接，支持动态编译等（这部分看时间安排）

设计寄存器机器

- 寄存器机器包含数据通路（寄存器和操作）和确定操作执行顺序的控制器
- 过程（以 GCD 算法为例）的机器：

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

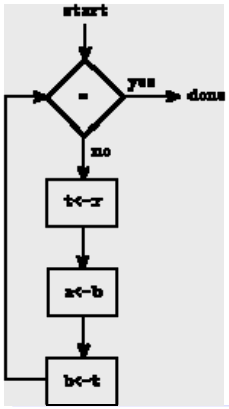
- 执行本算法的机器必须维护 a 和 b 的轨迹，假定两个值存于寄存器 a 和 b

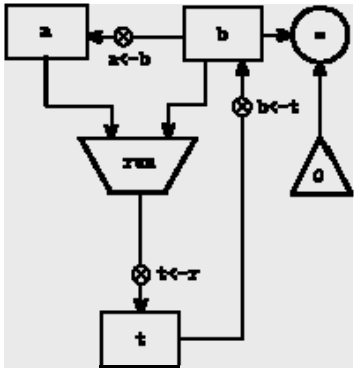


- 所需操作：判断 b 是否 0，计算 a 除以 b 的余数（假定有计算设备）
- 每一次循环迭代需要同时更新 a 和 b。由于一条简单指令只能更新一个寄存器，因此需要引进辅助寄存器 t
- 根据上述分析做出的数据通路见图

设计寄存器机器

- 为了寄存器机器能正确工作，必须正确控制其中各按钮的开闭顺序
- 下左图是 GCD 机器的控制器，用流程图表示：方框是动作，菱形框是判断。控制按箭头方向运行，进入判断后的流向由数据通路图中的检测决定。控制到达 done 时工作结束





对这个 GCD 机器，控制到达 done 时寄存器 a 里存着算出的 GCD 值

描述寄存器机器的语言

- 用这种图形描述很小的机器还比较方便，但难用于描述大型机器
- 为方便使用，需要考虑一种描述寄存器机器的文本语言。一种明显的设计是提供两套描述方式，分别用于描述数据通路和控制器
- 数据通路描述：寄存器和操作，包括为寄存器命名；给寄存器赋值的按钮（要命名）及其控制的数据传输的数据源（寄存器/常量/操作）。操作描述也需要命名，并描述其输入
- 控制器是指令序列，加上一些表示控制入口点的标号。指令可为：
  - 数据通路的一个按钮：指定寄存器赋值动作
  - test 指令：完成检测
  - 条件转跳指令（branch）：基于前面检测结果，检测为真时转到指定标号的指令；检测为假时继续下一条指令
  - 无条件转跳指令（goto），转到指定标号的指令执行

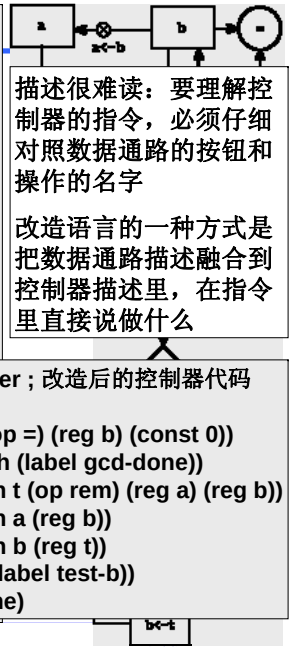
  标号作为 branch 和 goto 的目标

用所定义语言描述的 GCD 机器

```
(data-paths
 (registers
  ((name a)
   (buttons ((name a<-b) (source (register b)))))
  ((name b)
   (buttons ((name b<-t) (source (register t)))))
  ((name t)
   (buttons ((name t<-r) (source (operation rem))))))

 (operations
  ((name rem)
   (inputs (register a) (register b)))
  ((name =)
   (inputs (register b) (constant 0)))))

(controller
 test-b                    ; label
   (test =)                ; test
   (branch (label gcd-done)) ; conditional b
   (t<-r)                  ; button push
   (a<-b)                  ; button push
   (b<-t)                  ; button push
   (goto (label test-b))     ; unconditional b
 gcd-done)                 ; label
```



描述很难读：要理解控制器的指令，必须仔细对照数据通路的按钮和操作的名字

改造语言的一种方式是把数据通路描述融合到控制器描述里，在指令里直接说做什么

```
(controller ; 改造后的控制器代码
   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)
```


寄存器机器语言

- 改造后的语言清晰多了，但还有缺点，如：
  - 较罗嗦，如果控制器指令里多次提到某个数据通路元素，就要多次写出其完整描述（上例简单，无此情况）。重复出现使实际数据通路结构不够清晰，看不清有多少寄存器/操作/按钮，及其互连关系
  - 虽然指令用 Lisp 表达式表示，但实际上这里只能写合法指令

  虽然有这些缺点，下面还是准备用这套寄存器机器语言

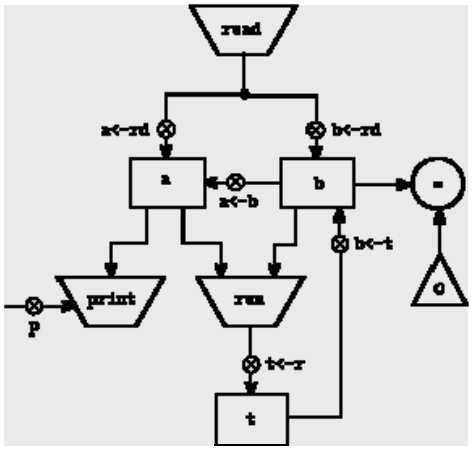
- 作为例子，现在想修改前面的 GCD 机器，使得能给它输入想求 GCD 的数，并能打印出计算结果
- 这里不想研究读入或输出的具体实现，而是假定有两个基本操作完成相应工作

假定有下面操作

- read 产生可存入寄存器的值，这种值来自机器之外
- print 给环境产生某种效果。图形上给 print 关联一个按钮，按压它导致 print 执行。指令形式为

  (perform (op print) (reg a))

寄存器机器语言



扩充后的机器
的数据通路

```
(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)))
```

扩充后的 GCD 机器控制器

工作过程：反复读入一对对数值，求出两个数的 GCD 并输出

机器设计的抽象

- 一部机器的定义总是基于某些基本操作，有些操作本身很复杂。有时我们可能把 Scheme 环境提供的操作作为基本操作
- 基于复杂操作定义机器，可以把注意力集中到某些关键方面，隐藏当前不关注的细节。需要时再基于更基本的操作构造这些操作，说明它们可以实现
- 例如，GCD 机器的一个操作是计算 a 除以 b 的余数赋给 t。如果希望机器不以它作为基本操作，需考虑基于更简单的操作计算余数
- 例如只用减法写出下面过程

```
(define (remainder n d)
  (if (< n d)
      n
      (remainder (- n d) d)))
```

可见，可以用一个减法操作和一个比较代替前面机器里的求余数。用这一方式重新构造的 GCD 机器见下页

机器设计的抽象

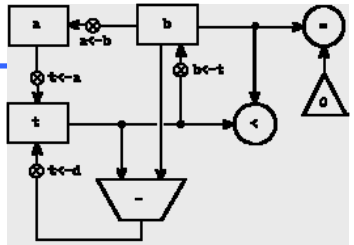
新 GCD 的控制器代码

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

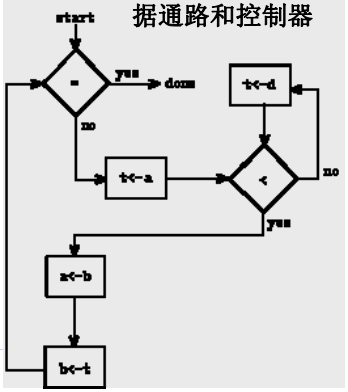
原指令：

(assign t (op rem) (reg a) (reg b))

代换为上面绿色指令序列，它又形成循环

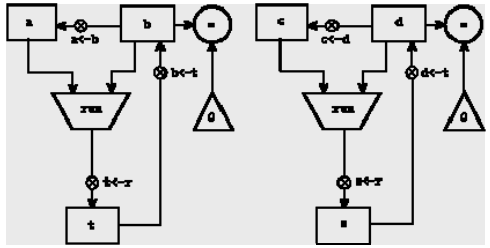


新 GCD 机器的数据通路和控制器



子程序

- 用更基本操作构成的结构代替原复杂操作，得到的控制器更复杂
- 我们希望能做某种安排，使相同的计算不必重复构造（以简化机器结构）
- 例：如果机器两次用 GCD，分别算 a 与 b 的和 c 与 d 的 GCD，数据通路将包含两个 GCD 块，控制器也会包含两段类似代码（不够令人满意）



用 GCD 两次的控制器代码片段：

```
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
```

子程序

- 多次出现同样部分不经济。应设法只用一个部件。现在考虑如何用一个 GCD 部件实现前面机器
- 如果第二次算 GCD 时寄存器 a 和 b 里的值都没用了（如有用，可以把它们移到其他寄存器），那么
  - 可以修改机器，第二次算 GCD 时先把用的值移到 a 和 b，用第一个 GCD 部分计算
  - 这样就能删去一个算 GCD 的通路，控制器代码片段如右

现在两个代码片段基本相同，只是入口和出口标号不同

还应设法消去重复的控制器代码

```
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
... ...
;; 这里把求 GCD 的数据移入 a 和 b
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
```


子程序

- 想法：让调用在进入 GCD 代码前把一个寄存器（用 continue）设为不同值，在 GCD 代码出口根据该寄存器跳到正确位置
- 得到的代码如右边所示，其中只有一段计算 GCD 的代码
- 这种技术可满足本程序需要（一段代码，正确返回）。但如果程序里有许多 GCD 计算，代码会很复杂，难写也难维护
- 要考虑更一般的实现模式（想想怎么办？）
- 下面想法基于代码指针，就是在寄存器里保存控制信息

```
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
```


子程序

- 新想法是在 continue 寄存器保存返回地址，GCD 代码最后按这个寄存器内容转跳。代码如右
- 需要扩充 goto 指令的能力：

  参数是 label 时，跳到相应标号的位置（直接转跳）

  参数是寄存器时跳到寄存器保存的标号处（寄存器间接跳）。不是跳到标号 continue，而是跳到 continue 寄存器里存的标号
- 这样就实现了子程序和调用
- 多个子程序调用相互无关时可共用一个 continue 寄存器。如子程序里调子程序，就要用多个 continue 寄存器（否则会丢失外层调用的返回标号）

```
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
 (goto (reg continue))

;; Before calling gcd, we assign to
;; continue the label to which gcd
;; should return.
 (assign continue (label after-gcd-1))
 (goto (label gcd))
after-gcd-1

;; Here is the second call to gcd,
;; with a different continuation.
 (assign continue (label after-gcd-2))
 (goto (label gcd))
after-gcd-2
```


用栈实现递归

- 用上面机制可以实现各种迭代计算过程。模式：一个寄存器表示一个状态变量，机器执行一个控制循环，其中修改寄存器值，直至完成计算
- 要想实现递归计算过程，还需要新的机制。考虑阶乘程序：

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

  计算中需要把另一个数的阶乘作为子问题先行解决
- 好像可以用同一机器去解决子问题。但这里有些情况
  - 子问题的回答并不是原问题的回答，其结果还需要乘以 n
  - 如直接模仿前面设计，减少 n 值后求 n-1 阶乘，原来的 n 值就丢了，没法再把它找回来求乘积
  - 相对子问题 n-1 还可能有子问题，由于初始 n 可取任意整数，子问题可有任意层嵌套，无法用有穷个部件构造出所需的机器

用栈实现递归

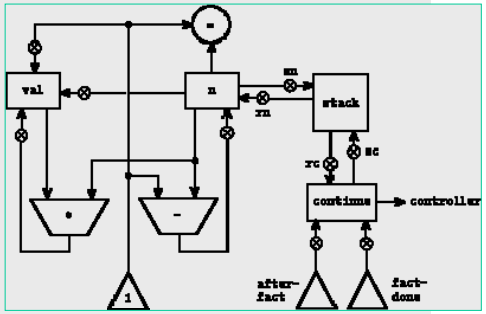
- 解决阶乘的问题，需要一种安排，使所有计算都由同一机器完成，子问题都用它解决。表面看阶乘计算需要无穷多嵌套的机器，但任何时刻只有一部在用。可以让它在遇到子问题时挂起正进行的计算，解决了子问题之后再回来继续该计算
- 注意：进入子问题时的状态与原问题不同（如 n 变成 n-1），要能在未来继续原来计算，就必须保存状态（保存当时 n 的值）
- 由于不能确知递归的深度，必须能保存任意多个寄存器值。这些值的使用顺序与保存顺序相反，后存先用。为此需要具有后进先出性质的结构 —— 栈。现在假定有一对栈操作 save 和 restore
- 有了栈就能任意多次重用同一阶乘机器，完成阶乘计算的所有子问题
- 控制问题：子程序结束后返回哪里？用 continue 寄存器存返回位置，但递归使用同一机器时又需要该寄存器，赋了新值就会丢掉将来要返回的位置。为保证正确返回，调用前也要把 continue 的值存入栈
- 这样就可以实现计算阶乘的机器了（数据通路和控制器）

用栈实现递归

设置最终返回地址

保存 continue 和 n 的值，以便子程序返回后恢复，使计算可以继续下去

```
(controller
    (assign continue (label fact-done))
  fact-loop
    (test (op =) (reg n) (const 1))
    (branch (label base-case))
    (save continue)
    (save n)
    (assign n (op -) (reg n) (const 1))
    (assign continue (label after-fact))
    (goto (label fact-loop))
  after-fact
    (restore n)
    (restore continue)
    (assign val (op *) (reg n) (reg val))   ; val now contains n(n - 1)!
    (goto (reg continue))                   ; return to caller
  base-case
    (assign val (const 1))                  ; base case: 1! = 1
    (goto (reg continue))                   ; return to caller
  fact-done)
```



进一步递归调用前保存返回的位置

用栈实现递归

- 原理上说，实现递归计算需要无穷的机器，这里用一部有穷机器实现。但这里仍有无穷的东西：栈的存储空间没有上界
- 在实际机器里栈的规模有限制，这一情况限制了机器能递归的深度，从而也限制了能求解的阶乘的大小
- 本例说明了处理递归的一般方法：一部常规寄存器机器加一个栈。一旦遇到递归子程序，就把将来还需要用的寄存器的值转存入栈。特别是当时 continue 寄存器的值
- 完全可以把所有子程序调用的处理都统一到这一模式。前面讲到在子程序里调用子程序的麻烦也就一起解决了

考虑双递归，以过程 fib 为例：

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

同样可以把斐波纳契数的计算实现为一部寄存器机器

过程里的两个调用都用这部机器完成，为此只需在调用前设置好 continue 寄存器，指明完成计算后转跳的位置


```
(controller
      (assign continue (label fib-done))
    fib-loop
      (test (op <) (reg n) (const 2))
      (branch (label immediate-answer))
      (save continue)  ;; set up to compute Fib(n - 1)
      (assign continue (label afterfib-n-1))
      (save n)                      ; save old value of n
      (assign n (op -) (reg n) (const 1)); clobber n to n - 1
      (goto (label fib-loop))       ; perform recursive call
    afterfib-n-1                    ; upon return, val contains Fib(n - 1)
      (restore n)
      (restore continue)
      (assign n (op -) (reg n) (const 2)) ;; set up to compute Fib(n - 2)
      (save continue)
      (assign continue (label afterfib-n-2))
      (save val)                    ; save Fib(n - 1)
      (goto (label fib-loop))
    afterfib-n-2                    ; upon return, val contains Fib(n - 2)
      (assign n (reg val))          ; n now contains Fib(n - 2)
      (restore val)                 ; val now contains Fib(n - 1)
      (restore continue)
      (assign val (op +) (reg val) (reg n))  ;  Fib(n - 1) +  Fib(n - 2)
      (goto (reg continue))         ; return to caller, answer is in val
    immediate-answer
      (assign val (reg n))          ; base case:  Fib(n) = n
      (goto (reg continue))
    fib-done)
```

寄存器指令的总结

下面的 <input> 或为 (reg <register-name>) 或为 (constant <constant-value>)

(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*>))

将标号存入寄存器和通过寄存器间接转跳

(assign <*register-name*> (label <*label-name*>))

(goto (reg <*register-name*>))

栈指令：

(save <*register-name*>)

(restore <*register-name*>)

前面的 <constant-value> 都是数值，后面还会用到字符串、符号和表常量：
(constant "abc")， (const abc)， (const (a b c))

寄存器机器模拟器

- 现在考虑寄存器机器语言的意义，原因：需要测试设计的机器能否按设想方式工作。手工模拟枯燥冗长。应实现一个寄存器机器模拟器
- 下面开发的模拟器是一个 Scheme 程序，它有 4 个接口过程
- 根据被模拟机器的描述构造一个用数据结构表示的模型。其中包含被模拟机器的寄存器、操作和控制器

(make-machine <*register-names*> <*operations*> <*controller*>)

- 另外三个过程用于操作被模拟的机器：

(set-register-contents! <*machine-model*> <*register-name*> <*value*>)
把一个值存入指定的寄存器

(get-register-contents <*machine-model*> <*register-name*>)
取出一个寄存器的内容

(start <*machine-model*>)
让机器开始运行

使用机器模拟器

定义 GCD 机器：

```
(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)))
```

make-machine 的参数依次为：

寄存器表

操作表（两维表，子表里是操作名和实现操作的 Scheme 过程）

控制器代码

启动一次计算。需要先设寄存器后启动

```
(set-register-contents! gcd-machine 'a 206)
done
(set-register-contents! gcd-machine 'b 40)
done
(start gcd-machine)
done
(get-register-contents gcd-machine 'a)
2
```


模拟器：机器模型

- make-machine 生成的机器模型是一个包含局部变量的过程，采用消息传递技术。它用 make-new-machine 构造出所有寄存器机器都有的公共部分，包括若干内部寄存器、一个栈和一个执行器
- make-machine 扩充公共模型，加入具体机器的寄存器/操作。而后用一个汇编器把控制器表翻译为机器能用的指令，并安装这个指令序列，最后返回修改后的机器模型：

```
(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))
```

加入所定义机器的寄存器

安装机器的操作

安装指令序列

机器模型：寄存器

- 一个寄存器是一个有局部状态的过程，其中可以保存一个值、访问或修改。make-register 构造这种寄存器

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

- 两个用于访问寄存器的过程：

```
(define (get-contents register)
  (register 'get))

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


机器模型：栈

- 栈是有局部状态的过程，用 make-stack 创建，接收 push、pop 和 initialize 消息（压入元素，弹出元素，初始化）

```
(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 (stack message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) (pop))
            ((eq? message 'initialize) (initialize))
            (else (error "Unknown request -- STACK" message))))
    stack))
```

定义一对访问栈的过程：

```
(define (pop stack)
  (stack 'pop))

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

基本机器

make-new-machine 过程如下。其内部状态包括指令计数器 pc，寄存器 flag，栈 stack 和一个空指令序列。操作表里开始时只包含初始化栈的操作，寄存器表里只包含 pc 和 flag。

```
(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))))  ;; 接下页
```

flag 用于实现分支，由检测操作设置，后续操作可以检查和使用；pc 确定当前指令位置

分配寄存器，在寄存器表里加入指定名字的寄存器

取寄存器的当前值

```
(define (execute)
  (let ((insts (get-contents pc)))
    (if (null? insts)
        'done
        (begin
          ((instruction-execution-proc (car insts)))
          (execute)))))
(define (basic-machine 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))))
basic-machine)))  ;; 最后返回构造好的基本机器
```

执行指令。总取 pc 所指向的指令来执行

指令执行将改变 pc

基本机器

- 为方便，给 start 定义一个使用接口过程，另外定义两个过程：

```
(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)
```

- 下面基本操作取指定寄存器的信息，许多过程都用它（包括上面过程）

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


汇编程序

- 汇编程序将寄存器机器的控制器表达式翻译为指令序列，每条指令带着自己的执行过程。汇编程序与前面求值器的分析器类似，但处理寄存器机器语言，实现分析与执行的分离
- 不知道 Scheme 表达式的值也可以做许多有用分析。不知道寄存器内容也可以做许多优化，例如
  - 用指向寄存器对象的指针代替对寄存器的引用
  - 用指向指令序列里具体位置的指针代替对标号的引用
- 生成执行过程前要确定标号的位置，工作过程是：
  - 扫描控制器正文，识别指令序列里的标号，构造一个指令表和一个标号位置关联表（将每个标号关联到指令表的一个位置）
  - 重新扫描控制器正文，生成并设置指令表里各指令的执行过程
- 汇编程序的入口是 assemble，它以一个控制器正文和一个基本机器模型为参数，返回存入模型的指令序列

汇编程序

- 汇编程序的代码

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

构造初始指令表和标号表，而后用这两个表调用其第二个参数

以指令表、标号表和机器为参数，生成指令的执行过程并将其插入指令表，最后返回指令表

逐项检查指令表内容，提取其中的标号

```
(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)
               (receive insts
                        (cons (make-label-entry next-inst insts) labels))
               (receive (cons (make-instruction next-inst) insts) labels)))))))
```

递归处理控制器正文序列的 cdr，将对其 car 处理得到的标号项加在对其 cdr 处理得到的指令表和标号表前面

检查控制器正文的第一项是否标号，根据情况加入指令表项或标号表项

汇编程序

- 标号表项就是序对，查标号表项的过程很简单：

```
(define (make-label-entry label-name insts)
  (cons label-name insts))

(define (lookup-label labels label-name)
  (let ((val (assoc label-name labels)))
    (if val
        (cdr val)
        (error "Undefined label -- ASSEMBLE" label-name))))
```

- 构造和使用指令表的几个简单过程：

```
(define (make-instruction text)
  (cons text '()))
(define (instruction-text inst)
  (car inst))
(define (instruction-execution-proc inst)
  (cdr inst))
(define (set-instruction-execution-proc! inst proc)
  (set-cdr! inst proc))
```

构造指令表时，执行过程暂时用一个空表，后面将填入实际执行过程

汇编程序

- undate-insts! 修改机器里的指令表。这里原来只有指令正文，执行过程用空表作为占位符。本过程加入实际的执行过程

```
(define (update-insts! insts labels machine)
  (let ((pc (get-register machine 'pc))
        (flag (get-register machine 'flag))
        (stack (machine 'stack))
        (ops (machine 'operations)))
    (for-each
     (lambda (inst)
       (set-instruction-execution-proc!
        inst
        (make-execution-procedure
         (instruction-text inst) labels machine
         pc flag stack ops)))
     insts)))
```

给一条指令设置执行过程

构造一条指令的执行过程

指令的执行过程

- 生成一条指令的执行过程，工作方式很像求值器的 analyze 过程

```
(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))))
```

每种指令有一个执行过程的生成过程，细节由具体指令的语法和意义确定

用数据抽象技术隔离指令的具体表示和对指令的操作

下面逐条考虑各种指令的处理

执行过程：assign

从指令中取出被赋值的寄存器和值表达式

根据表达式的运算符构造执行过程，区分基本表达式和一般表达式

- 生成赋值指令的执行过程

```
(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 ()                ; assign 的执行过程
        (set-contents! target (value-proc))
        (advance-pc pc)))))
```

构造一般 op 表达式的执行过程

其中用到的两个辅助过程

```
(define (assign-reg-name assign-instruction)
  (cadr assign-instruction))
(define (assign-value-exp assign-instruction)
  (cddr assign-instruction))
```

构造基本表达式的执行过程

基本表达式包括 reg,label,const

通用的指令计数器更新过程

```
(define (advance-pc pc) (set-contents! pc (cdr (get-contents pc))))
```


执行过程：test

- make-test 处理 test 指令，设置 flag 寄存器，更新 pc：

```
(define (make-test inst machine labels operations flag pc)
  (let ((condition (test-condition inst)))
    (if (operation-exp? condition)
        (let ((condition-proc
               (make-operation-exp condition machine
                                   labels operations)))
          (lambda ()
            (set-contents! flag (condition-proc))
            (advance-pc pc)))
        (error "Bad TEST instruction -- ASSEMBLE" inst))))

(define (test-condition test-instruction)
  (cdr test-instruction))
```

做出 op 表达式的执行过程

执行过程：branch

- branch 指令根据 flag 更新 pc：

```
(define (make-branch inst machine labels flag pc)
  (let ((dest (branch-dest inst)))
    (if (label-exp? dest)
        (let ((insts (lookup-label labels (label-exp-label dest))))
          (lambda ()
            (if (get-contents flag)
                (set-contents! pc insts)
                (advance-pc pc))))
        (error "Bad BRANCH instruction -- ASSEMBLE" inst))))

(define (branch-dest branch-instruction)
  (cadr branch-instruction))
```

取得转跳指令里的标号

从标号表里找出标号在指令序列里的位置

根据 flag 的值决定如何更新 pc

注意：只有 goto 指令可用寄存器间接。当然也可扩充 branch 指令功能

执行过程：goto

- goto 的特殊情况是转跳位置可能用标号或寄存器描述，分别处理

```
(define (make-goto inst machine labels pc)
  (let ((dest (goto-dest inst)))
    (cond ((label-exp? dest)
           (let ((insts
                  (lookup-label labels (label-exp-label dest))))
             (lambda () (set-contents! pc insts))))
          ((register-exp? dest)
           (let ((reg
                  (get-register machine (register-exp-reg dest))))
             (lambda () (set-contents! pc (get-contents reg)))))
          (else (error "Bad GOTO instruction -- ASSEMBLE" inst)))))

(define (goto-dest goto-instruction)
  (cadr goto-instruction))
```

是标号，找出标号位置

这里可扩充找不到位置的处理

处理转跳位置由寄存器描述的情况

执行过程：save 和 restore

- 这两条指令针对特定寄存器使用栈，并更新 pc

```
(define (make-save inst machine stack pc)
  (let ((reg (get-register machine (stack-inst-reg-name inst))))
    (lambda ()
      (push stack (get-contents reg))
      (advance-pc pc))))

(define (make-restore inst machine stack pc)
  (let ((reg (get-register machine (stack-inst-reg-name inst))))
    (lambda ()
      (set-contents! reg (pop stack))
      (advance-pc pc))))

(define (stack-inst-reg-name stack-instruction)
  (cadr stack-instruction))
```


执行其他指令

- 其他指令的执行由 make-perform 处理，它生成相应的执行过程，在实际模拟中执行对应动作并更新 pc：

```
(define (make-perform inst machine labels operations pc)
  (let ((action (perform-action inst)))
    (if (operation-exp? action)
        (let ((action-proc
               (make-operation-exp
                action machine labels operations)))
          (lambda ()
            (action-proc)
            (advance-pc pc)))
        (error "Bad PERFORM instruction -- ASSEMBLE" inst))))

(define (perform-action inst) (cdr inst))
```

构造 op 表达式的执行过程

执行：子表达式

- 赋值指令和其他指令中用到 make-operation-exp，其中可能用 reg、label 或 const 的值，这些是基本表达式，相应执行过程：

```
(define (make-primitive-exp exp machine labels)
  (cond ((constant-exp? exp)
         (let ((c (constant-exp-value exp)))
           (lambda () c)))
        ((label-exp? exp)
         (let ((insts (lookup-label labels (label-exp-label exp))))
           (lambda () insts)))
        ((register-exp? exp)
         (let ((r (get-register machine (register-exp-reg exp))))
           (lambda () (get-contents r))))
        (else (error "Unknown expression type -- ASSEMBLE" exp))))
```

基本表达式的语法过程：

```
(define (register-exp? exp) (tagged-list? exp 'reg))
(define (register-exp-reg exp) (cadr exp))
(define (constant-exp? exp) (tagged-list? exp 'const))
(define (constant-exp-value exp) (cadr exp))
(define (label-exp? exp) (tagged-list? exp 'label))
(define (label-exp-label exp) (cadr exp))
```


执行：子表达式

- assign、perform 和 test 指令的执行过程都将机器操作应用于操作对象（reg 表达式或 const 表达式），这种操作的执行过程

```
(define (make-operation-exp exp machine labels operations)
  (let ((op (lookup-prim (operation-exp-op exp) operations))
        (aprocs
          (map (lambda (e) (make-primitive-exp e machine labels))
               (operation-exp-operands exp))))
    (lambda ()
      (apply op (map (lambda (p) (p)) aprocs)))))
```

为每个操作对象生成一个执行过程

调用操作对象的执行过程，得到它们的值；而后应用操作本身的执行过程

相应语法过程：

```
(define (operation-exp? exp)
  (and (pair? exp) (tagged-list? (car exp) 'op)))
(define (operation-exp-op operation-exp)
  (cadr (car operation-exp)))
(define (operation-exp-operands operation-exp)
  (cdr operation-exp))
```


执行：子表达式

- 要找出模拟过程，用操作名到从机器的操作表里查找：

```
(define (lookup-prim symbol operations)
  (let ((val (assoc symbol operations)))
    (if val
        (cadr val)
        (error "Unknown operation -- ASSEMBLE" symbol))))
```

注意：这样找到的是对应的 Scheme 过程，而后用 apply 应用它

监视执行

- 模拟可用于验证所定义机器的正确性，还可以考查其性能
- 可以给模拟程序安装一些“测量仪器”，例如记录栈操作的次数等
- 下面做这一工作，为此需要在基本机器模型里加一个操作

```
(list (list 'initialize-stack (lambda () (stack 'initialize)))
      (list 'print-stack-statistics (lambda () (stack 'print-statistics))))
```

修改 make-stack 的定义，加入计数和输出统计结果的功能：

```
(define (make-stack)
  (let ((s '())
        (number-pushes 0)
        (max-depth 0)
        (current-depth 0))
    (define (push x)
      (set! s (cons x s))
      (set! number-pushes (+ 1 number-pushes))
      (set! current-depth (+ 1 current-depth))
      (set! max-depth (max current-depth max-depth)))   ;; 接下页
```

监视执行

```
(define (pop)
  (if (null? s)
      (error "Empty stack -- POP")
      (let ((top (car s)))
        (set! s (cdr s))
        (set! current-depth (- current-depth 1))
        top)))
(define (initialize)
  (set! s '()) (set! number-pushes 0)
  (set! max-depth 0) (set! current-depth 0)
  'done)
(define (print-statistics)
  (newline)
  (display (list 'total-pushes  = ' number-pushes
                 'maximum-depth = ' max-depth)))
(define (stack message)
  (cond ((eq? message 'push) push)
        ((eq? message 'pop) (pop))
        ((eq? message 'initialize) (initialize))
        ((eq? message 'print-statistics) (print-statistics))
        (else (error "Unknown request -- STACK" message))))
stack))
```


总结

- 介绍的寄存器机器的概念和结构（数据通道，控制器）
- 如何设计寄存器机器，用它实现计算
- 寄存器机器的文本表示形式（寄存器机器语言）
- 复杂操作的实现和子程序
- 递归子程序和栈，通用的子程序调用模式
- 寄存器机器模拟器
  - 机器结构
  - 基本机器
  - 汇编程序
  - 各种操作的执行程序
  - 监视运行