中国科学院大学
University of Chinese Academy of Sciences

初识程序执行

递归与切片
程序执行
斐波那契计算机

zxu@ict.ac.cn
zhangjialin@ict.ac.cn

提纲

- 同学们的反馈
  - 学习小组，模数环，数字符号能表示一切吗？
- 程序设计（感性认识）
  - **循环**与数组
  - **递归**与切片，进阶实验：班级快排计算机
- 程序执行（感性认识）
  - 使用切片求斐波那契数
  - 回顾冯诺依曼模型，定义一个更为简单的斐波那契计算机（FC）
    - FC如何实现for循环
      - 指令集、汇编语言程序
  - 基址-索引寻址模式
  - 斐波那契计算机求斐波那契数的逐步指令执行过程

程序有限性
系统有限性

课件中包含教科书未包括的素材引用，特此致谢

复习循环与数组

如何用%c实现%d?

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
   var sum_bytes [4]byte
   var j int
   for j = 3; sum!= 0; j-- {
      sum_bytes[j] = byte(sum%10) + '0'
      sum = sum / 10
   }
   fmt.Printf("%c", sum_bytes[0])
   fmt.Printf("%c", sum_bytes[1])
   fmt.Printf("%c", sum_bytes[2])
   fmt.Printf("%c", sum_bytes[3])
   fmt.Printf("\n")
}
```

使用整数除法 / 和求余 % 操作，
从数值**1045**依次摘取出数字**5, 4, 0, 1**

```
sum % 10
sum = sum / 10
```

sum_bytes[3]	1045 % 10 = 5
sum	1045 / 10 = 104
sum_bytes[2]	104 % 10 = 4
sum	104 / 10 = 10
sum_bytes[1]	10 % 10 = 0
sum	10 / 10 = 1
sum_byte[0]	1 % 10 = 1
sum	1 / 10 = 0

sum_bytes = [‘1’, ‘0’, ‘4’, ‘5’]
数组索引 0 1 2 3

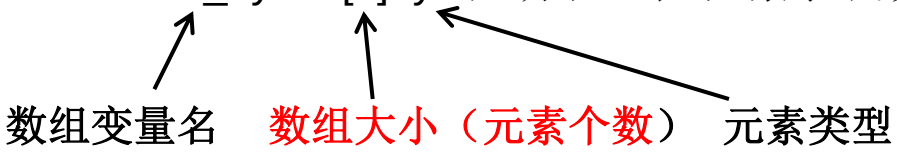
用字符占位符%c实现十进制占位符 %d
fmt.Printf("%d\n", sum)

四个打印语句fmt.Printf(“%c”, …)
依次打印出1, 0, 4, 5四个字符

复习：循环与数组天然地相互配合

- 数组是一组连续存放的相同类型元素

  var sum_bytes [4]byte声明了一个4元素字节数组



sum_bytes = [‘1’, ‘0’, ‘4’, ‘5’]

数组索引　0　1　2　3

- 通过索引访问数组元素
  - sum_bytes[1]的值为什么是48_{10}

- 数组的结构是静态的（编译时确定）
  - 但数组元素的值是动态的（执行时确定）

- 循环与数组天然地相互配合

```
var sum_bytes [4]byte
  var j int
  for j = 3; sum!= 0; j-- {
    sum_bytes[j] = byte(sum%10) + '0'
    sum = sum / 10
  }
```

使用整数除法 / 和求余 % 操作，
从数值1045依次摘取出数字5, 4, 0, 1

sum % 10
sum = sum / 10

sum_bytes[3]	1045 % 10 = 5
sum	1045 / 10 = 104
sum_bytes[2]	104 % 10 = 4
sum	104 / 10 = 10
sum_bytes[1]	10 % 10 = 0
sum	10 / 10 = 1
sum_byte[0]	1 % 10 = 1
sum	1 / 10 = 0

1. 递归与切片

- 霍尔悖论
  - 1959年在莫斯科国立大学当访问学生时产生快排思想
  - 1960年在Elliott Brothers公司为Elliott 803计算机开发排序程序时，发明并实现快排算法。但是，

“**Very difficult to explain**” 很难向他人说明快排算法

  - 1961年发表，非常简洁易懂（只有半页，8行伪代码）

Hoare, C. A. R. (1961). "Algorithm 64: Quicksort". Comm. ACM. 4 (7): 321

ALGORITHM 64
QUICKSORT
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure  quicksort (A,M,N); value M,N;
             array A; integer M,N;
comment  Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M−N) ln
(N−M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin      integer I,J;
           if M < N then begin partition (A,M,N,I,J);
                               quicksort (A,M,J);
                               quicksort (A, I, N)
                         end
end        quicksort
```

与教科书版本基本一样

ALGORITHM 65
FIND
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure  find (A,M,N,K); value M,N,K;
             array A; integer M,N,K;
comment  Find will assign to A [K] the value which it would
have if the array A [M:N] had been sorted. The array A will be
partly sorted, and subsequent entries will be faster than the first;
```

Communications of the ACM 321

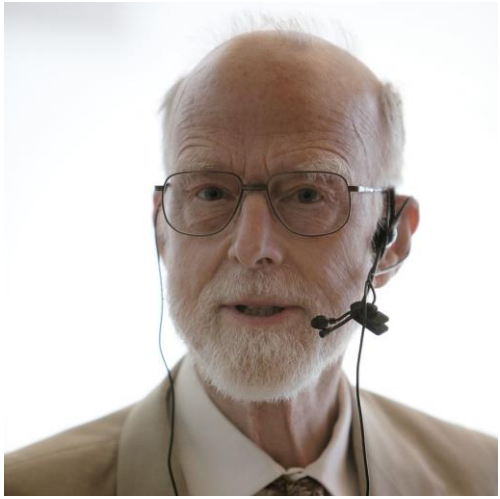
为什么？

因为1961年系统提供了递归抽象！

quicksort函数调用自身；quicksort也是自身的模块

这个递归程序能够无缝衔接自动执行

— Tony Hoare, 1980图灵奖演说：皇帝的旧衣



Acu-Exams

抽象化（**Abstraction**）：少数精心构造的计算抽象可描述万千应用

模块化（**Modularity**）：多个模块有规律地组合成为计算系统

无缝衔接（**Seamless Transition**）：计算过程在系统中流畅地执行

回顾Go程序的基本结构

为什么要声明函数、调用函数？

Declare a function, call a function

主包声明语句	package main	// 定义主包
导入语句	import …	// 导入其他人写的包，如不调用则不出现
常量声明语句	const …	// 声明全局常量，可不出现
变量声明语句	var …	// 声明全局变量，可不出现
函数声明语句	func X(…) …{…}	// 声明程序员自定义的函数，可不出现
主函数声明语句	func main() {	// 声明主包中必须有的主函数
	…	// 主函数的函数体
	}	

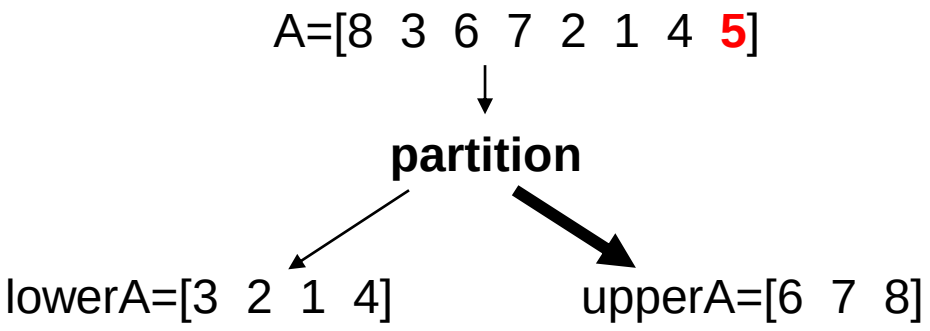
```
func X(…) …{                    // 如函数体等括起来的{…}称为一个代码块（code block）
   常量声明语句                  // 局部常量，作用域是本函数
   变量声明语句                  // 局部变量，作用域是本函数
   赋值语句
   循环语句
   条件判断语句
   函数调用语句，如打印语句
}
```

模块化语言抽象

- 程序语言需提供两类功能：完备性 + 扩展性
  - 变量声明+赋值语句+循环 是 图灵完备的
    - （变量声明+赋值语句+循环）程序可算出任意可计算数
  - 扩展性：如何编写100行、1万行、10万行的程序
    - 应对程序复杂性需要模块（module, modularity）
- 我们关注两类模块
  - 程序包（package）
    - 主包、导入包
    - 库函数：导入包中的函数，如fmt.Printf
  - 函数（function），包括递归函数
  - 其他还有对象（object）、服务（service）等

1.1 简版快速排序算法fastsort

- **输入**：8元素整数数组d = [8 3 6 7 2 1 4 5]
- **输出**：8元素整数数组d = [1 2 3 4 5 6 7 8]
  - 元素从小到大排好序了
- **步骤**： 调用fastsort(d)。
  函数fastsort(A)的计算过程如下：
  1. 基线情况（base case）：如果A只包含0个元素（如[]）或1个元素（如[3]），fastsort(A)结束
  2. 选择A的最后一个元素作为标杆元素（pivot）
  3. 调用划分函数partition
     - 将小于标杆元素的元素放入lowerA子数组（称为小数组）中，将大于标杆元素的元素放入upperA子数组（称为大数组）中
  4. 递归调用fastsort(lowerA)
  5. 递归调用fastsort(upperA)



d = [8 3 6 7 2 1 4 5]

d = [3 2 1 4] 5 [6 7 8]

d = [3 2 1] 4 [] 5 [6 7 8]

d = [] 1 [2 3] 4 5 [6 7 8]

d = 1 [2] 3 [] 4 5 [6 7 8]

d = 1 [] 2 [] 3 4 5 [6 7 8]

d = 1 2 3 4 5 [6 7] 8 []

d= 1 2 3 4 5 [6] 7 [] 8

d = 1 2 3 4 5 [] 6 [] 7 8

d = 1 2 3 4 5 6 7 8

如何实现partition？

- 选择1：使用数组

  var d [8]int = [8]int {8,3,6,7,2,1,4,5}

  递归调用初始情况：A = d

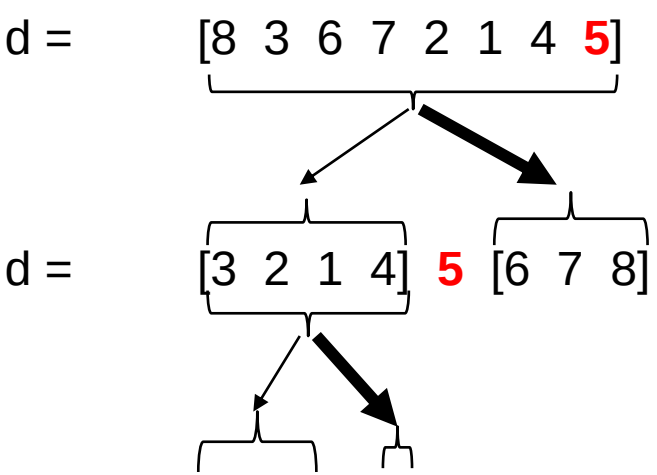
  var lowerA [**4**]int

  var upperA [3]int

  lowerA, upperA = partition(A)

  [3 2 1 4], [6, 7, 8] = partition([8 3 6 7 2 1 4 5])

- 但是，存在两个问题
  - len(lowerA)=**4**只有在运行时才知道
    - 假如d=[8 5 6 7 2 1 4 **3**], len(lowerA)=2
    - 不能用声明语句静态定义
  - 递归调用fastsort(A)，第二轮有
    - A=[3 2 1]，lowerA=[]，upperA=[2 3]
      子数组的长度动态变化
- 需要一种**动态数组**



d = [] **1** [2 3] 4 5 [6 7 8]
d = 1 [2] **3** [] 4 5 [6 7 8]
d = 1 [] **2** [] 3 4 5 [6 7 8]
d = 1 2 3 4 5 [6 7] **8** []
d= 1 2 3 4 5 [6] **7** [] 8
d = 1 2 3 4 5 [] **6** [] 7 8
d = 1 2 3 4 5 6 7 8

选择2：使用切片（slice）

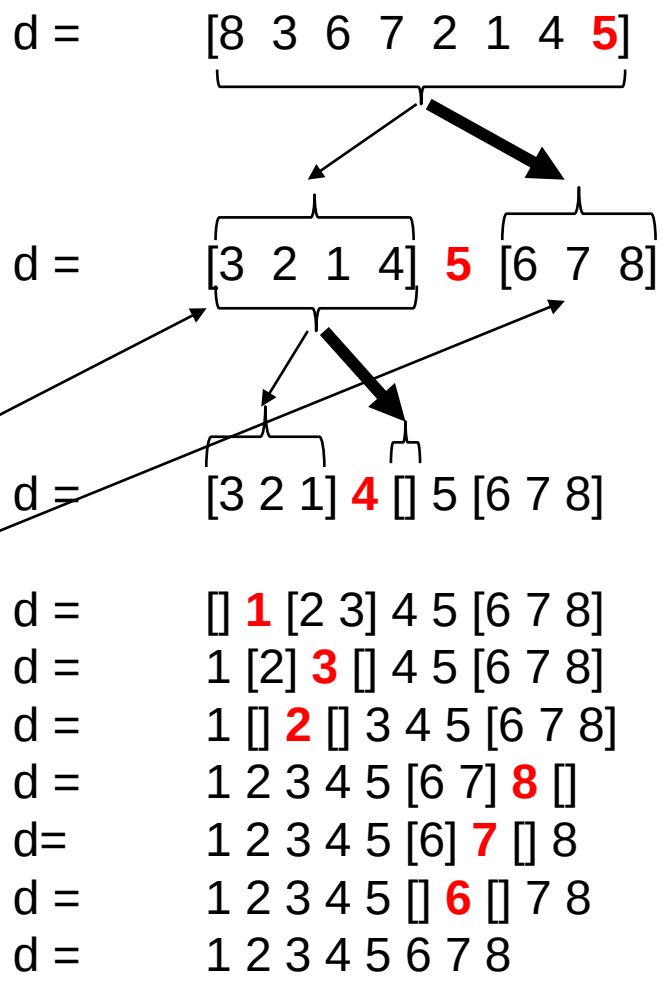
var s []int = d[0:8]切片声明之后，
len(s)=？
s[0]=？　s[7]=？　s[8]=？

```
var d [8]int = [8]int {8,3,6,7,2,1,4,5}
// 定义切片s，指向子数组d[0:8]
var s []int = d[0:8]
```

切片名是s，指向底层数组d
元素类型int，起始索引0，结束索引8-1=7

切片是一个结构，指向数组d的某个特定子数组。开始时，切片s指向整个数组d

第1次调用partition(A)返回两个切片
lowerA是切片d[0:4]
upperA是切片d[5:8]



选择2：使用切片（slice）

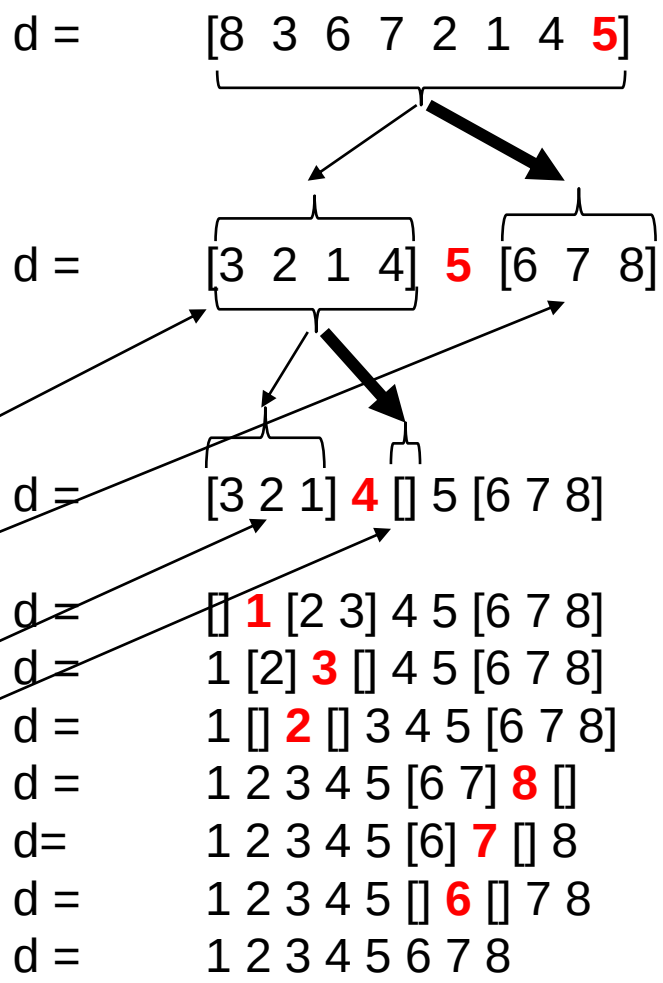
var s []int = d[0:8]切片声明之后，
len(s)=？
s[0]=？　s[7]=？　s[8]=？

```
var d [8]int = [8]int {8,3,6,7,2,1,4,5}
// 定义切片s，指向子数组d[0:8]
var s []int = d[0:8]
```

切片名是s，指向底层数组d
元素类型int，起始索引0，结束索引8-1=7

切片是一个结构，指向数组d的某个特定子数组。开始时，切片s指向整个数组d

第1次调用partition(A)返回两个切片
　　lowerA是切片d[0:4]
　　upperA是切片d[5:8]
第2次调用partition(A)返回两个切片
　　lowerA是切片d[0:3]
　　upperA是切片d[4:4]



使用切片（slice）

var s []int = d[0:8]切片声明之后，
len(s)=8，s[0]=d[0]=8，s[7]=d[7]=5
s[8]无定义，越界了

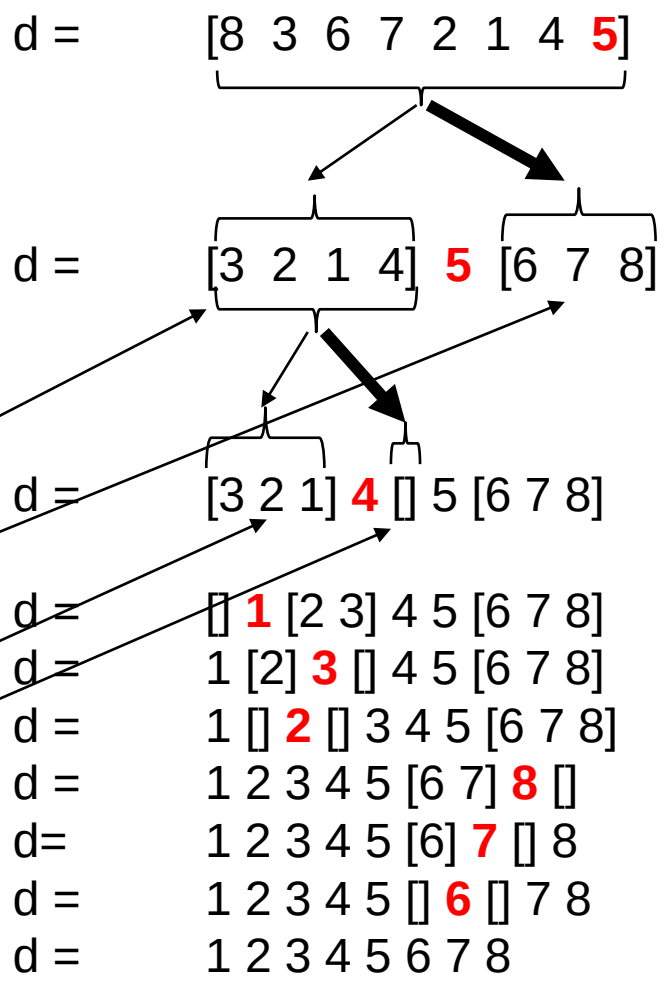
```
var d [8]int = [8]int {8,3,6,7,2,1,4,5}
// 定义切片s，指向子数组d[0:8]
var s []int = d[0:8]
```

切片名是s，指向底层数组d
元素类型int，起始索引0，结束索引8-1=7

切片是一个结构，指向数组d的某个特定子数组。开始时，切片s指向整个数组d

第1次调用partition(A)返回两个切片
- lowerA是切片d[0:4]
- upperA是切片d[5:8]

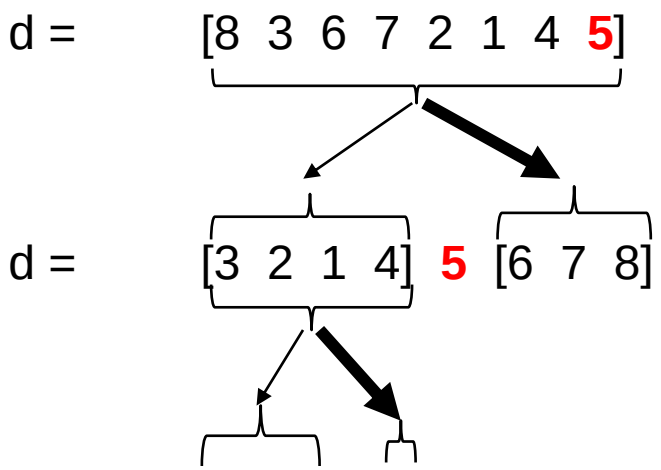
第2次调用partition(A)返回两个切片
- lowerA是切片d[0:3]
- upperA是切片d[4:4]



切片自然地支持动态数组与递归

1.2 简版快速排序程序fastsort.go

```
package main            // fastsort.go
import "fmt"
var d [8]int = [8]int {8,3,6,7,2,1,4,5} // 定义数组d = [8 3 6 7 2 1 4 5]
var s []int = d[0:8]    // 定义切片s，指向子数组d[0:8]
func main() {
   fastsort(s)
   fmt.Println(s)
}
func fastsort(A []int) {    // 递归函数定义，输入参数是一个整数切片
   if len(A) < 2 {  // 判断是不是基础情况（base case）
      return  // 当切片A的长度<2时，数组包含0或1个元素，退出
   }
   lowerA, upperA := partition(A)  //划分A并返回小、大两个子数组
   fastsort(lowerA)                // 递归排序小数组lowerA
   fastsort(upperA)                // 递归排序大数组upperA
}
```



d = [] 1 [2 3] 4 5 [6 7 8]
d = 1 [2] 3 [] 4 5 [6 7 8]
d = 1 [] 2 [] 3 4 5 [6 7 8]
d = 1 2 3 4 5 [6 7] 8 []
d= 1 2 3 4 5 [6] 7 [] 8
d = 1 2 3 4 5 [] 6 [] 7 8
d = 1 2 3 4 5 6 7 8

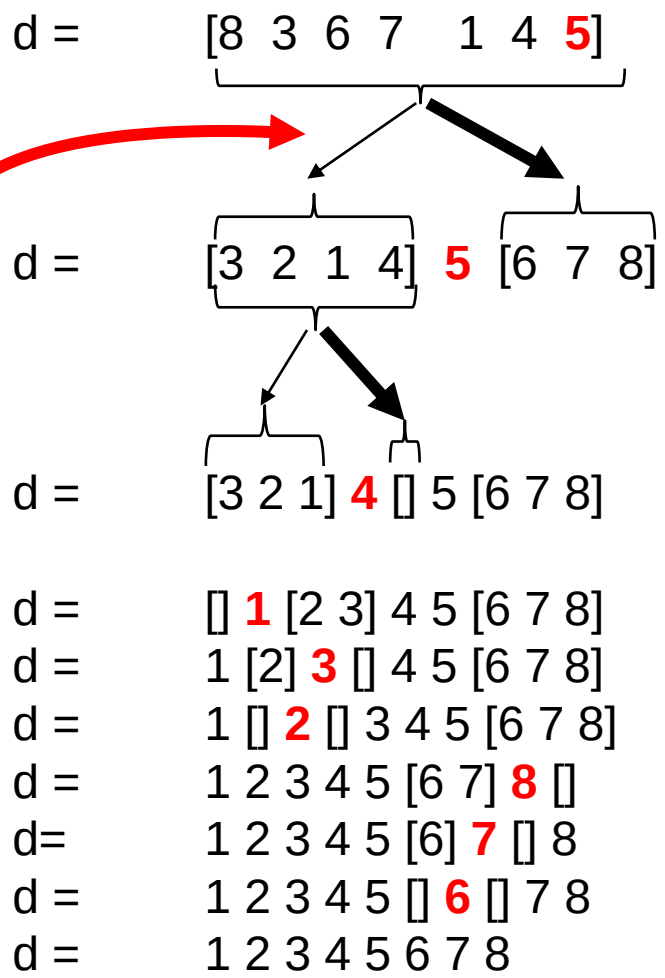
函数partition的功能

输入参数	函数调用	返回值
第一轮：d[0:8] →	partition(s) →	**d[0:4]**, d[5:8]
第二轮：**d[0:4]** →	partition(lowerA) →	**d[0:3]**, d[4:4]
第三轮：**d[0:3]** →	partition(lowerA) →	d[0:0], d[1:3]

func partition(A []int) ([]int, []int) { ... }

1.2 简版快速排序程序fastsort.go（演示）

```
package main            // fastsort.go
import "fmt"
var d [8]int = [8]int {8,3,6,7,2,1,4,5} // 定义数组d = [8 3 6 7 2 1 4 5]
var s []int = d[0:8]    // 定义切片s，指向子数组d[0:8]
func main() {
   fastsort(s)
   fmt.Println(s)
}
func fastsort(A []int) {    // 递归函数定义，参数是一个整数切片
   if len(A) < 2 {  // 判断是不是基础情况（base case）
      return  // 当切片A的长度<2时，数组包含0或1个元素，退出
   }
   lowerA, upperA := partition(A)  //划分A并返回小、大两个子数组
   fastsort(lowerA)                // 递归排序小数组lowerA
   fastsort(upperA)                // 递归排序大数组upperA
}
func partition(A []int) ([]int, []int) { // A的数组至少包含两个元素
   lower := 0  // lower 是小数组lowerA的长度，初始值为0
   for i:= 0; i<len(A); i++ {  // 逐个扫描A的元素A[i]
      // 此处插入你的代码
      // 建议：如果A[i]<标杆值，A[i]与A[lower]交换并更新lower
   }
   // 标杆应紧跟在lowerA之后
   A[lower], A[len(A)-1] = A[len(A)-1], A[lower]
   return A[0:lower], A[lower+1:len(A)] // 返回切片lowerA和upperA
}
```



2. 使用切片的斐波那契程序（演示）

- 自底向上（bottom up）的斐波那契程序fib.up.go
  - 草稿：输入n的值写死在代码中了，程序只能计算F(50)
  - 改进版：程序启动后提示用户敲入n的值，再计算出F(n)
    - fib是动态创建的，不是静态声明的

```
> go run fib.up.go
Enter n: （用户敲入F(n)的n的值，如10）
F( 10 )= 55
> go run fib.up.go
Enter n: （用户敲入50）
F( 50 )= 12586269025
>
```

```
package main                    // fib.bu.go 草稿
import "fmt"
const n = 50                    // 声明F(n)的n
var fib [n + 1]int              // 声明n+1元素数组
func main() {
        fmt.Println("F(", n, ")=", fibonacci(n))
}
func fibonacci(n int) int {
        fib[0] = 0
        fib[1] = 1
        for i := 2; i <= n; i++ {
           fib[i] = fib[i-1] + fib[i-2]
        }
        return fib[n]
}
```

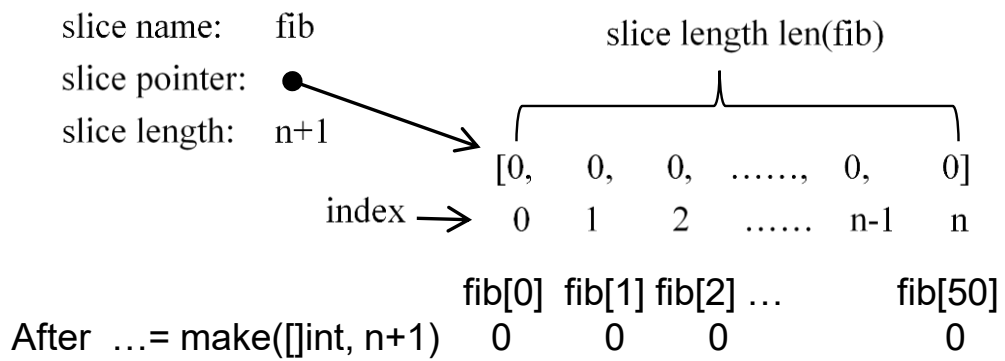
```
package main                                    // fib.bu.go
import "fmt"
func main() {
        var n int                               // 声明输入变量 n
        fmt.Printf("Enter n: ")                 // 提示敲入n的值，如50
        fmt.Scanf("%d",&n)                      // 将该值50放在变量n中
        fmt.Println("F(", n, ")=", fibonacci(n))
}
func fibonacci(n int) int {
        fib := make([]int, n+1)                 // 创建长度为n+1的切片fib
        fib[0] = 0
        fib[1] = 1
        for i := 2; i <= n; i++ {
           fib[i] = fib[i-1] + fib[i-2]
        }
        return fib[n]
}
```

使用切片的斐波那契程序

- 使用系统提供的函数（built-in function）make构建切片fib
- 切片是一个结构体，指向一个make函数创建的底层数组，长度为n+1
  - 切片名是fib
  - 底层数组相当于 var fib [n+1]int
  - 不过，是执行make函数动态创建的，不是静态声明的

```
package main                    // fib.bu.go
import "fmt"
func main() {
        var n int
        fmt.Printf("Enter n: ")
        fmt.Scanf("%d",&n)
        fmt.Println("F(", n, ")=", fibonacci(n))
}
func fibonacci(n int) int {
        fib := make([]int, n+1)
        fib[0] = 0
        fib[1] = 1
        for i := 2; i <= n; i++ {
           fib[i] = fib[i-1] + fib[i-2]
        }
        return fib[n]
 }
```

var fib []int = make([]int, n+1)

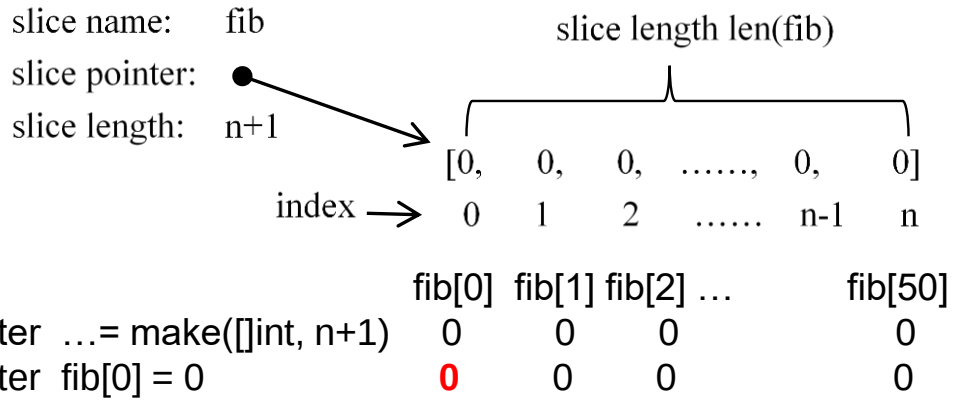


使用切片的斐波那契程序

- 使用系统提供的函数（built-in function）make构建切片fib
- 切片是一个结构体，指向一个make函数创建的底层数组，长度为n+1
  - 切片名是fib
  - 底层数组相当于 var fib [n+1]int
  - 不过，是执行make函数动态创建的，不是静态声明的

```
package main              // fib.bu.go
import "fmt"
func main() {
        var n int
        fmt.Printf("Enter n: ")
        fmt.Scanf("%d",&n)
        fmt.Println("F(", n, ")=", fibonacci(n))
}
func fibonacci(n int) int {
        fib := make([]int, n+1)
        fib[0] = 0
        fib[1] = 1
        for i := 2; i <= n; i++ {
           fib[i] = fib[i-1] + fib[i-2]
        }
        return fib[n]
 }
```

var fib []int = make([]int, n+1)



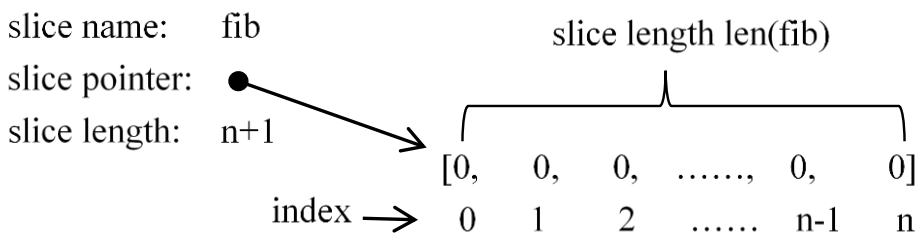
	fib[0]	fib[1]	fib[2] …	fib[50]
After …= make([]int, n+1)	0	0	0	0
After fib[0] = 0	**0**	0	0	0

使用切片的斐波那契程序

- 使用系统提供的函数（built-in function）make构建切片fib
- 切片是一个结构体，指向一个make函数创建的底层数组，长度为n+1
  - 切片名是fib
  - 底层数组相当于 var fib [n+1]int
  - 不过，是执行make函数动态创建的，不是静态声明的

```
package main              // fib.bu.go
import "fmt"
func main() {
        var n int
        fmt.Printf("Enter n: ")
        fmt.Scanf("%d",&n)
        fmt.Println("F(", n, ")=", fibonacci(n))
}
func fibonacci(n int) int {
        fib := make([]int, n+1)
        fib[0] = 0
        fib[1] = 1
        for i := 2; i <= n; i++ {
           fib[i] = fib[i-1] + fib[i-2]
        }
        return fib[n]
 }
```

var fib []int = make([]int, n+1)



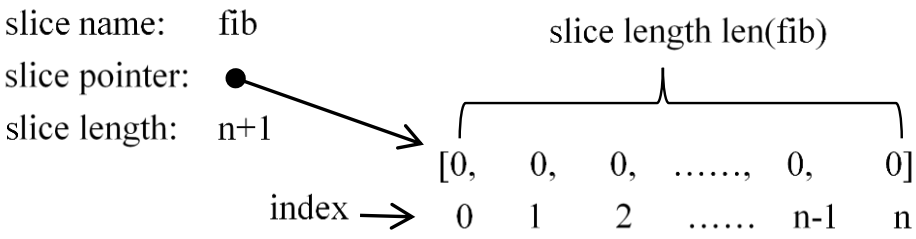
	fib[0]	fib[1]	fib[2] …	fib[50]
After …= make([]int, n+1)	0	0	0	0
After fib[0] = 0	**0**	0	0	0
After fib[1] = 1	**0**	**1**	0	0

使用切片的斐波那契程序

- 使用系统提供的函数（built-in function）make构建切片fib
- 切片是一个结构体，指向一个make函数创建的底层数组，长度为n+1
  - 切片名是fib
  - 底层数组相当于 var fib [n+1]int
  - 不过，是执行make函数动态创建的，不是静态声明的

```
package main              // fib.bu.go
import "fmt"
func main() {
        var n int
        fmt.Printf("Enter n: ")
        fmt.Scanf("%d",&n)
        fmt.Println("F(", n, ")=", fibonacci(n))
}
func fibonacci(n int) int {
        fib := make([]int, n+1)
        fib[0] = 0
        fib[1] = 1
        for i := 2; i <= n; i++ {
           fib[i] = fib[i-1] + fib[i-2]
        }
        return fib[n]
 }
```

var fib []int = make([]int, n+1)



	fib[0]	fib[1]	fib[2] …	fib[50]
After …= make([]int, n+1)	0	0	0	0
After fib[0] = 0	**0**	0	0	0
After fib[1] = 1	**0**	**1**	0	0
After fib[2] = fib[1]+fib[0]	**0**	**1**	**1**	0

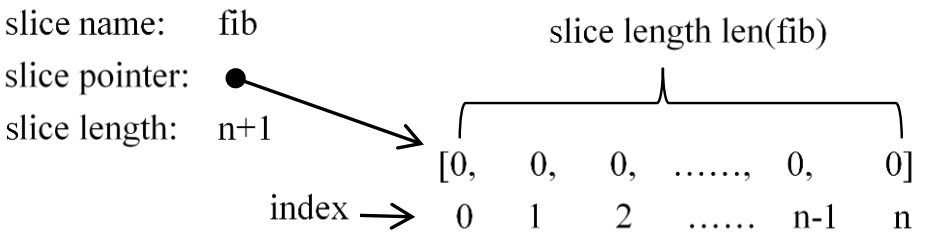
…

使用切片的斐波那契程序

- 使用系统提供的函数（built-in function）make构建切片fib
- 切片是一个结构体，指向一个make函数创建的底层数组，长度为n+1
  - 切片名是fib
  - 底层数组相当于 var fib [n+1]int
  - 不过，是执行make函数动态创建的，不是静态声明的

```
package main              // fib.bu.go
import "fmt"
func main() {
        var n int
        fmt.Printf("Enter n: ")
        fmt.Scanf("%d",&n)
        fmt.Println("F(", n, ")=", fibonacci(n))
}
func fibonacci(n int) int {
        fib := make([]int, n+1)
        fib[0] = 0
        fib[1] = 1
        for i := 2; i <= n; i++ {
           fib[i] = fib[i-1] + fib[i-2]
        }
        return fib[n]
}
```

var fib []int = make([]int, n+1)



	fib[0]	fib[1]	fib[2] …	fib[50]
After …= make([]int, n+1)	0	0	0	0
After fib[0] = 0	**0**	0	0	0
After fib[1] = 1	**0**	**1**	0	0
After fib[2] = fib[1]+fib[0]	**0**	**1**	**1**	0

…

程序执行：计算机如何实现这段代码

课堂小测验

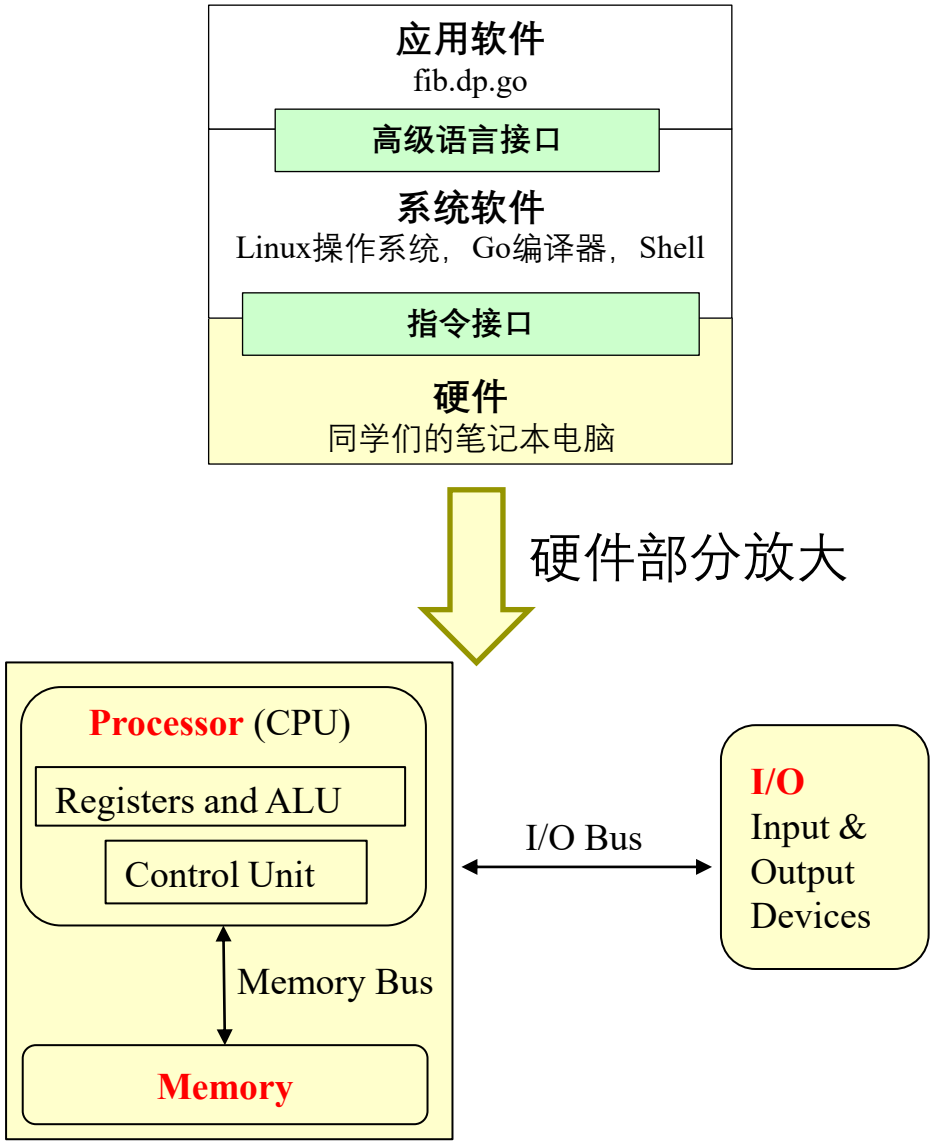
- 姓名：　　学号：

- 下述标粗的赋值语句为什么不是**sum = sum + name[i]**（）
  A. 因为name[i]是字节类型，需要通过类型转换操作int(name[i])变换成整数类型
  B. 因为name[i]是整数类型，需要通过类型转换操作int(name[i])变换成字节类型
  C. 因为i是字节类型，需要通过类型转换操作int(name[i])变换成整数类型
  D. 因为i是整数类型，需要通过类型转换操作int(name[i])变换成字节类型

```
package main //name_to_number-0.go
import "fmt"
func main() {
        var name string = "Alan Turing"
        sum := 0
        for i := 0; i < 11; i++ {
                sum = sum + int(name[i])
        }
        fmt.Printf("%d\n", sum)
}
```

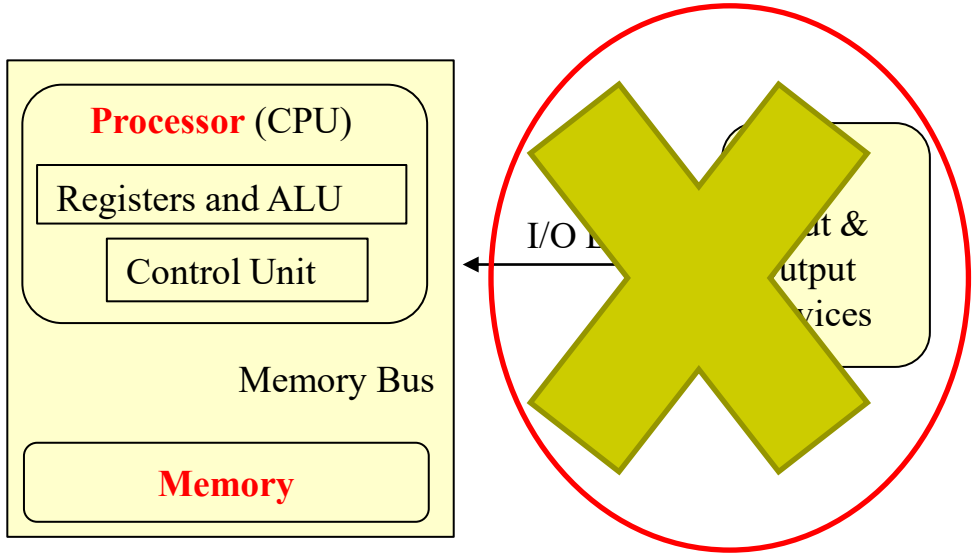
3. 简化冯诺依曼计算机模型

- 计算机大体上分为三层
  - 硬件、系统软件、应用软件，通过两个接口连接
- 冯诺依曼模型五要点
  - 二进制表示
  - 三类部件（P-M-I/O）
  - 存储程序计算机
    - 字节寻址：字节是访问存储器的基本单位
  - 指令驱动
  - 串行执行
- 计算机如何支持循环与数组？
  - 设计斐波那契计算机



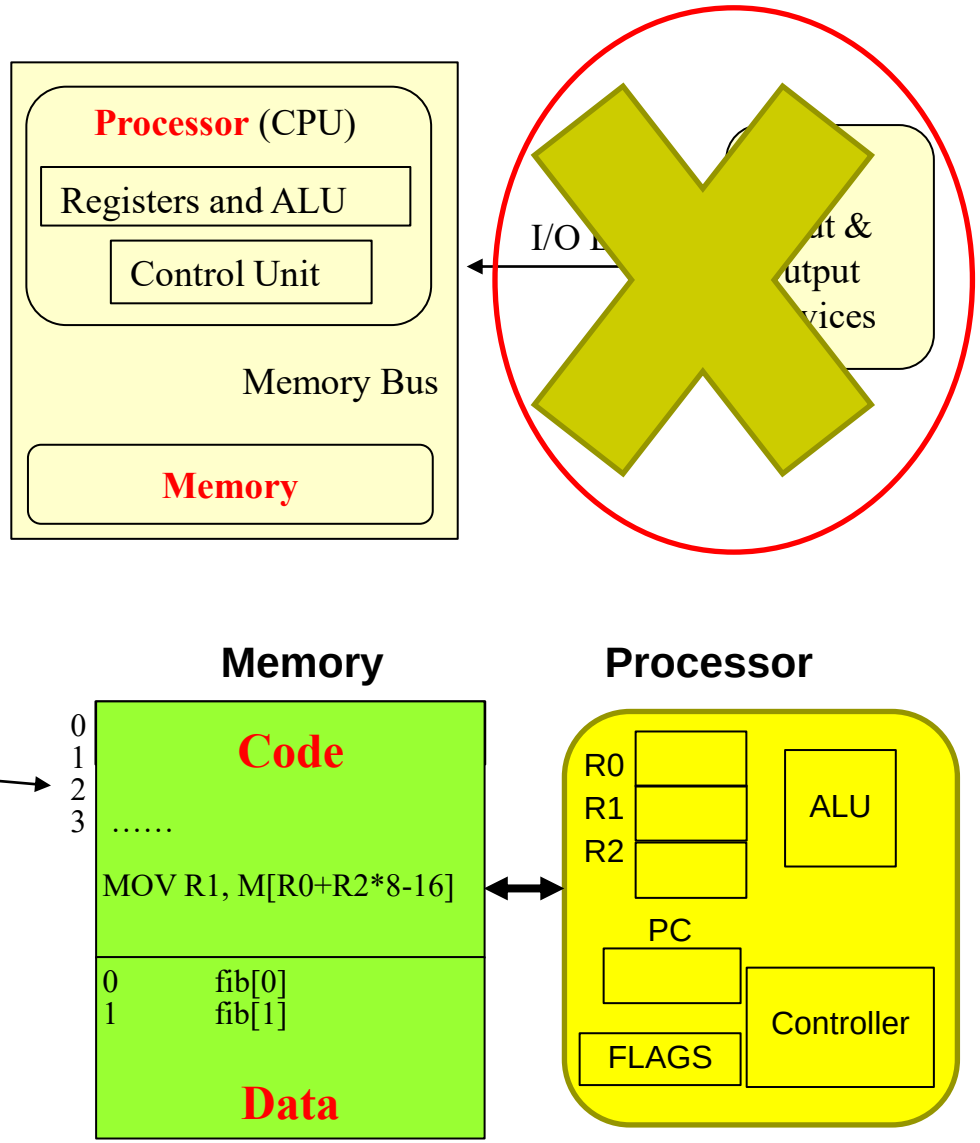
3.1 斐波那契计算机（Fibonacci Computer, FC）

- 通过简化冯诺依曼计算机模型获得
  - 忽略冯诺依曼计算机的I/O子系统



斐波那契计算机（FC）

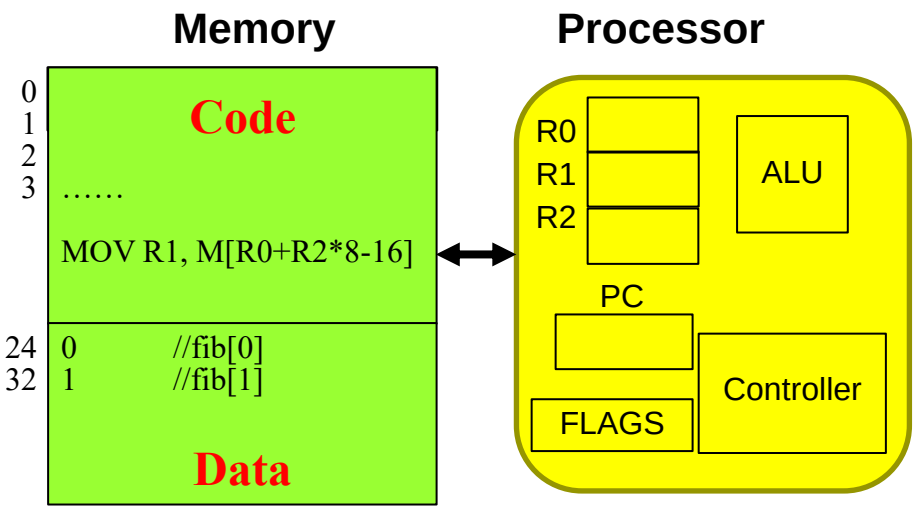
- 通过简化冯诺依曼计算机模型获得
  - 忽略冯诺依曼计算机的I/O子系统
  - 忽略控制器和运算器的实现
- 重点关注
  - 寄存器
    - R0，R1，R2，PC，FLAGS
  - 存储器
    - 内存地址2存放多少比特？
  - 指令集（共有6条指令）
  - 寻址模式
  - 状态：寄存器和存储器的值
    - 初始状态
    - 每一步执行后的状态



斐波那契计算机（FC）

- 只执行所示Go代码
- 人工编译成汇编程序
  - 12条指令
- 字节寻址存储器
  - 24字节存放代码
  - 408字节存放数据，即数组fib
- 寄存器
  - 三个64位通用寄存器R0、R1、R2，每个寄存器存放64位数
  - 程序计数器PC，存放下一条指令的地址
  - 状态寄存器FLAGS，存放指令执行的状态信息，如R2是否小于51
- 指令集（共有6条指令）
  - MOV to Register
  - MOV to Memory
  - ADD 加法指令
  - INC Increment 增1指令
  - CMP Compare 比较指令
  - JL Jump if Less than 条件跳转

```
fib[0] = 0                  MOV 0, R1
                            MOV R1, M[R0]          //R0=12 initially
fib[1] = 1                  MOV 1, R1
                            MOV R1, M[R0+8]
for i := 2; i < 51; i++ {   MOV 2, R2              // i:=2
   fib[i] = fib[i-1] + fib[i-2]
                            MOV 0, R1              // label Loop
                            ADD M[R0+R2*8-16], R1
                            ADD M[R0+R2*8-8], R1
                            MOV R1, M[R0+R2*8-0]
                            INC R2                 // i++
                            CMP 51, R2             // i < 51?
}                           JL Loop          // Jump to Loop if Less than
```



FC初始状态，注意基址寄存器R0=24

寄存器内容		存储器内容		
寄存器	值	地址	指令	注释
FLAGS		0	MOV 0, R1	0→R1；每条指令占2个地址
PC	0	2	MOV R1, M[R0]	R1→M[R0]
R0	24	4	MOV 1, R1	1→R1
R1		6	MOV R1, M[R0+8]	R1→M[R0+8]
R2		8	MOV 2, R2	2→R2
R0: 基址寄存器 初始值=24 R1: 累加器 R2: 索引寄存器 地址= 基址+索引*8+偏移量 Address=base+index*8+offset fib[i-2]所在地址 =R0+R2*8 -16 fib[i-1]所在地址 =R0+R2*8 -8 fib[i]所在地址 =R0+R2*8 -0		10 Loop	MOV 0, R1	0→R1；标签Loop=10
		12	ADD M[R0+R2*8-16], R1	R1+ M[R0+R2*8-16] → R1
		14	ADD M[R0+R2*8-8], R1	R1+ M[R0+R2*8-8] → R1
		16	MOV R1, M[R0+R2*8-0]	R1→ M[R0+R2*8-0]
		18	INC R2	R2+1→R2
		20	CMP 51, R2	如果R2<51，’<’→FLAGS
		22	JL Loop	如果FLAGS=’<’, Loop→PC
		24		fib[0]；每个数据占8个地址
		32		fib[1]
		40		fib[2]
		48		fib[3]
		……		……
		424		fib[50]

FC初始状态

寄存器内容		存储器内容		
寄存器	值	地址	指令	注释
FLAGS		0	MOV 0, R1	0→R1；
PC	**0**	2	MOV R1, M[R0]	R1→M[R0]
R0	**24**	4	MOV 1, R1	1→R1
R1		6	MOV R1, M[R0+8]	R1→M[R0+8]
R2		8	MOV 2, R2	2→R2
R0: 基址寄存器 初始值=24 R1: 累加器 R2: 索引寄存器 地址= 基址+索引*8+偏移量 Address=base+index*8+offset fib[i-2]所在地址 =R0+R2*8 -16 fib[i-1]所在地址 =R0+R2*8 -8 fib[i]所在地址 =R0+R2*8 -0		10 Loop	MOV 0, R1	0→R1；
		12	ADD M[R0+R2*8-16], R1	R1+ M[R0+R2*
		14	ADD M[R0+R2*8-8], R1	R1+ M[R0+R2*
		16	MOV R1, M[R0+R2*8-0]	R1→ M[R0+R2
		18	INC R2	R2+1→R2
		20	CMP 51, R2	如果R2<51，’<
		22	JL Loop	如果FLAGS=’<
		24		fib[0]；
		32		fib[1]
		40		fib[2]
		48		fib[3]
		……		……
		424		fib[50]

fig[0] =0

fig[1] =1

i:=2

M[12]是指地址为12的内存单元

理解重点：多条指令如何支持循环

- 以及数组
  - 在科学计算中，循环和数组往往配套出现
- 注意
  - 汇编代码如何忠实反映了循环之变与不变
  - 数组与循环如何配合

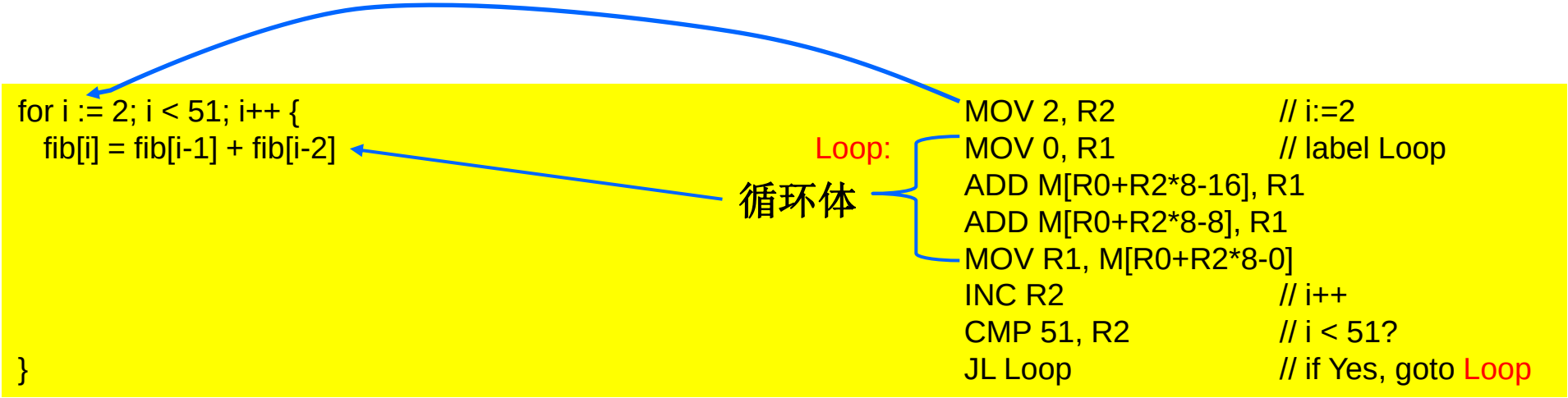
```
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]

}
```

```
        MOV 2, R2              // i:=2
Loop:   MOV 0, R1              // label Loop
        ADD M[R0+R2*8-16], R1
        ADD M[R0+R2*8-8], R1
        MOV R1, M[R0+R2*8-0]
        INC R2                 // i++
        CMP 51, R2             // i < 51?
        JL Loop                // if Yes, goto Loop
```

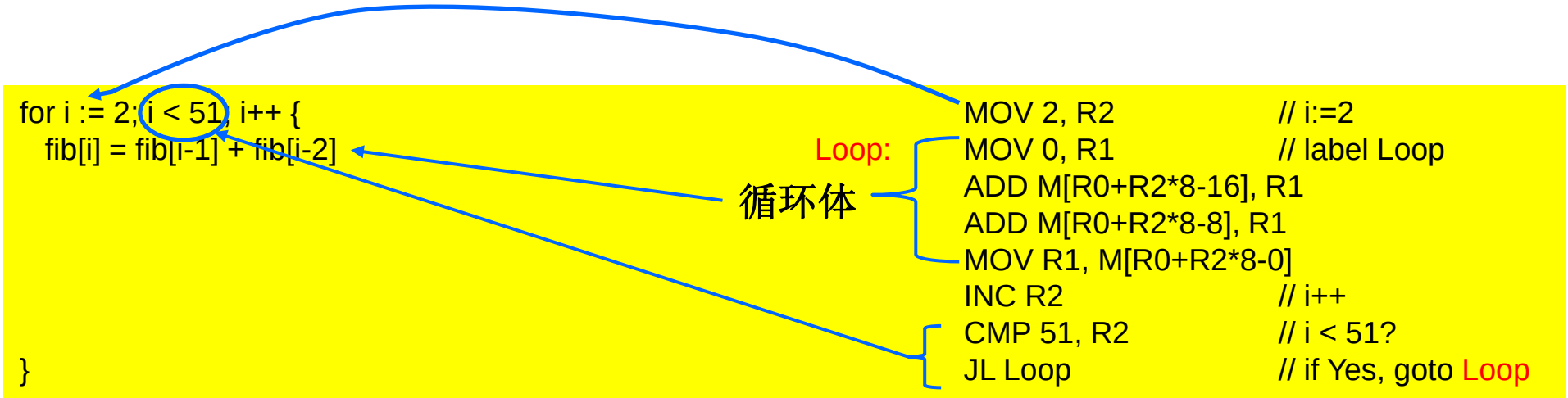
理解重点：多条指令如何支持循环

- 以及数组
  - 在科学计算中，循环和数组往往配套出现
- 注意
  - 汇编代码如何忠实反映了循环之变与不变
  - 数组与循环如何配合



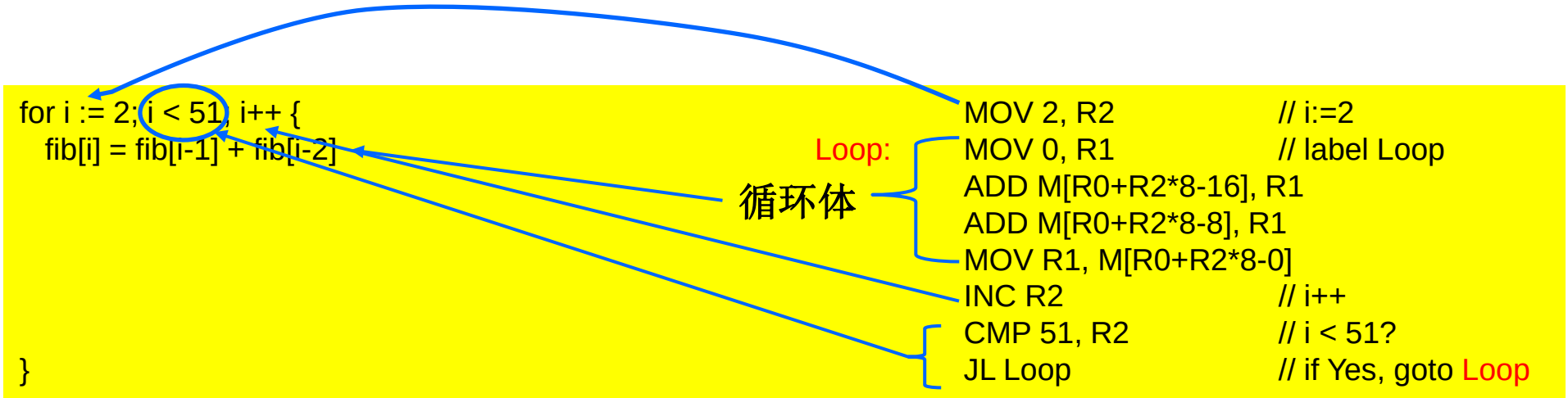
理解重点：多条指令如何支持循环

- 以及数组
  - 在科学计算中，循环和数组往往配套出现
- 注意
  - 汇编代码如何忠实反映了循环之变与不变
  - 数组与循环如何配合



理解重点：多条指令如何支持循环

- 以及数组
  - 在科学计算中，循环和数组往往配套出现
- 注意
  - 汇编代码如何忠实反映了循环之变与不变
  - 数组与循环如何配合



理解重点：多条指令如何支持循环

- 以及数组
  - 在科学计算中，循环和数组往往配套出现
- 注意
  - 汇编代码如何忠实反映了循环之变与不变
  - 数组与循环如何配合

```
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
                                  0+fib[i-2]
}
```

```
        MOV 2, R2              // i:=2
Loop:   MOV 0, R1              // label Loop
        ADD M[R0+R2*8-16], R1
        ADD M[R0+R2*8-8], R1
        MOV R1, M[R0+R2*8-0]
        INC R2                 // i++
        CMP 51, R2             // i < 51?
        JL Loop                // if Yes, goto Loop
```

理解重点：多条指令如何支持循环

- 以及数组
  - 在科学计算中，循环和数组往往配套出现
- 注意
  - 汇编代码如何忠实反映了循环之变与不变
  - 数组与循环如何配合

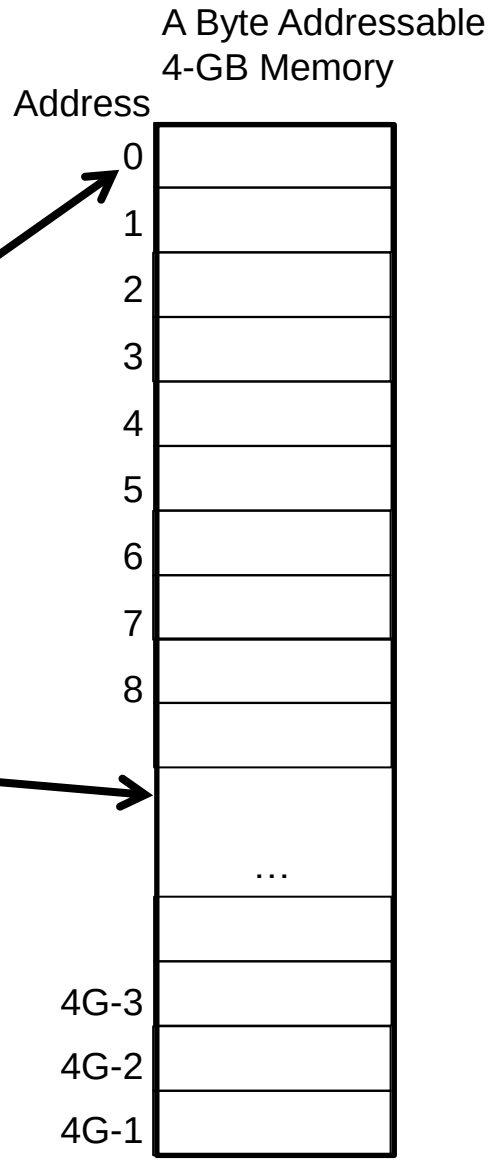
```
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
                                                  0+fib[i-2]
                                                  0+fib[i-2]+fib[i-1]
                                                  fib[i]=fib[i-2]+fib[i-1]
}
```

```
                MOV 2, R2              // i:=2
Loop:           MOV 0, R1              // label Loop
                ADD M[R0+R2*8-16], R1
                ADD M[R0+R2*8-8], R1
                MOV R1, M[R0+R2*8-0]
                INC R2                 // i++
                CMP 51, R2             // i < 51?
                JL Loop                // if Yes, goto Loop
```

3.2 基址索引偏移量寻址模式

The base-index-offset addressing mode

- **address = base + index*8 + offset**
  实际地址 = 基址 + 索引*比例因子 + 偏移量
- Assume
  - 基址 Base = 0
  - 索引 Index = 1
- When Offset = - 8
  - **address = 0 + 1*8 + (-8) = 0**
- When Offset = 2
  - **address = 0 + 1*8 + 2 = 10**



基址索引偏移量寻址模式　天然适配循环

- **address = base + index*8 + offset**
  实际地址 = 基址 + 索引*比例因子 + 偏移量
- 进入for循环，i:=2
  - 基址寄存器R0=24
  - 索引寄存器R2=2
  - 比例因子=8，因为fib[i]是64位整数
  - 赋值语句fib[i] = fib[i-1] + fib[i-2]编译成

    MOV 0, R1

    ADD M[R0+R2*8-16], R1

    ADD M[R0+R2*8-8], R1

    MOV R1, M[R0+R2*8-0]
  - 第一条加法指令实现0+fib[0]

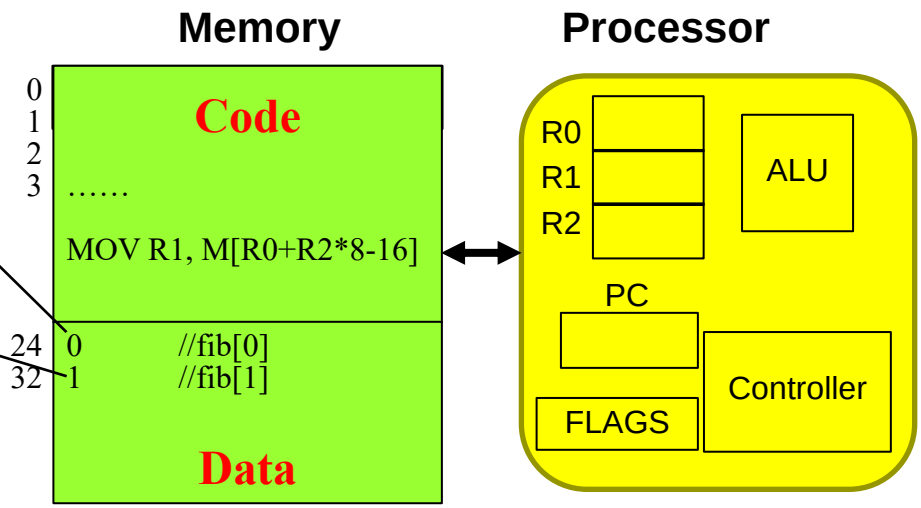
    R1+ M[R0+R2*8-16] → R1，即

    0 + M[24+2*8-16] → R1，即0+fib[0] → R1
  - 第二条加法指令实现0+fib[0]+fib[1]

    R1+ M[R0+R2*8-8] → R1，即

    0 + M[24+2*8-8] → R1，即0+fib[1] → R1

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0] //R0=12 initially
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8]
for i := 2; i < 51; i++ {           MOV 2, R2    // i:=2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1    // label Loop
                                    ADD M[R0+R2*8-16], R1
                                    ADD M[R0+R2*8-8], R1
                                    MOV R1, M[R0+R2*8-0]
                                    INC R2       // i++
                                    CMP 51, R2   // i < 51?
}                                   JL Loop      // if Yes, goto Loop
```



基址索引偏移量寻址模式　天然适配循环

- **address = base + index*8 + offset**
  实际地址 = 基址 + 索引*比例因子 + 偏移量
- 进入for循环，i:=2
  - 基址寄存器R0=24
  - 索引寄存器R2=2
  - 比例因子=8，因为fib[i]是64位整数
  - 赋值语句fib[i] = fib[i-1] + fib[i-2]编译成

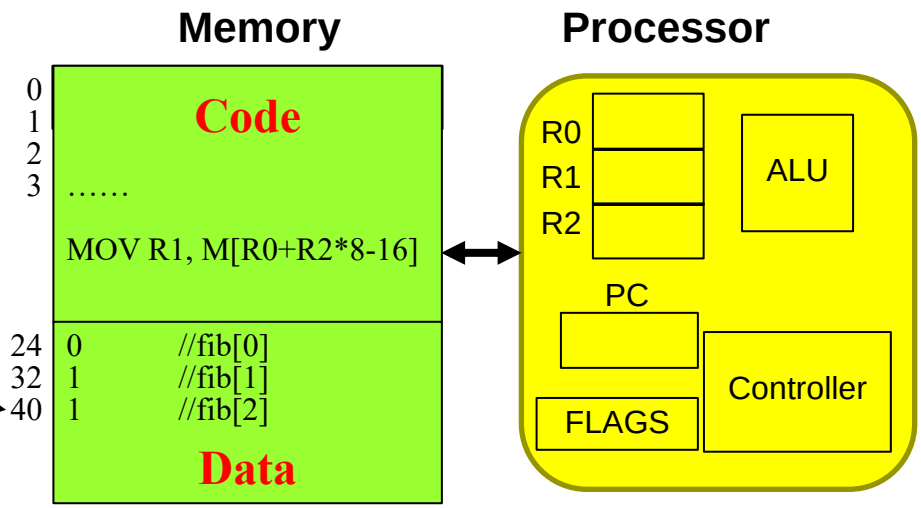
    MOV 0, R1
    ADD M[R0+R2*8-16], R1
    ADD M[R0+R2*8-8], R1
    MOV R1, M[R0+R2*8-0]
  - 第一条加法指令实现0+fib[0]

    R1+ M[R0+R2*8-16] → R1，即
    0 + M[24+2*8-16] → R1，即0+fib[0] → R1
  - 第二条加法指令实现0+fib[0]+fib[1]

    R1+ M[R0+R2*8-8] → R1，即
    0 + M[24+2*8-8] → R1，即0+fib[1] → R1
  - 指令MOV实现fib[2]=0+fib[0]+fib[1]

R1 → M[24+2*8-0]，即1→M[40]，即1→fib[2]

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0] //R0=12 initially
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8]
for i := 2; i < 51; i++ {           MOV 2, R2    // i:=2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1    // label Loop
                                    ADD M[R0+R2*8-16], R1
                                    ADD M[R0+R2*8-8], R1
                                    MOV R1, M[R0+R2*8-0]
                                    INC R2       // i++
                                    CMP 51, R2   // i < 51?
}                                   JL Loop      // if Yes, goto Loop
```



基址索引偏移量寻址模式　天然适配数组和循环

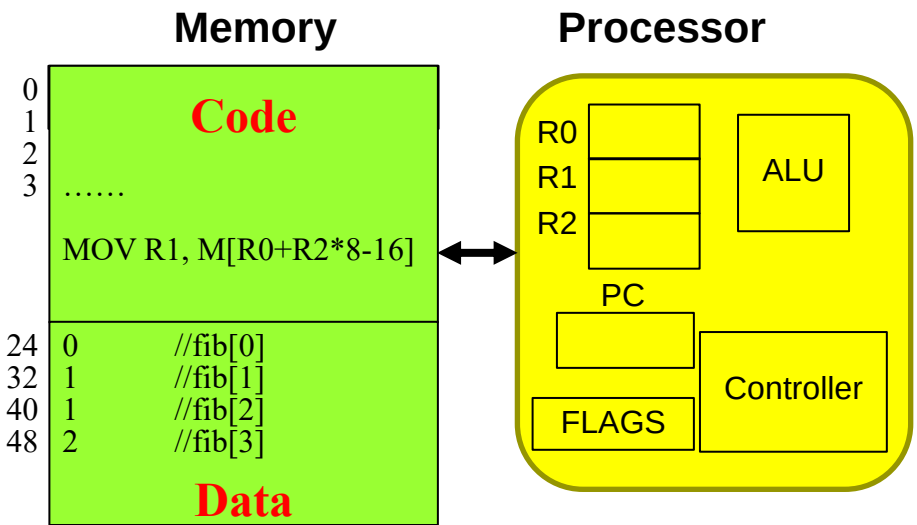
- **address = base + index*8 + offset**
  实际地址 = 基址 + 索引*比例因子 + 偏移量
- 后面三条指令后，
  进入下一次迭代，i:=3
  - 基址寄存器R0=24
  - 索引寄存器R2=**3**（变了！）
  - 比例因子=8，因为fib[i]是64位整数

  - 第一条加法指令实现0+fib[1]
    R1+ M[R0+R2*8-16] → R1，即
    0 + M[24+**3***8-16] → R1，即0+M[32] → R1
    即0+fib[1] → R1，即0+1 → R1
  - 第二条加法指令实现0+fib[1]+fib[2]
    R1+ M[R0+R2*8-8] → R1，即
    1 + M[24+**3***8-8] → R1，即1+M[40] → R1
    即1+fib[2] → R1，即1+1 → R1
  - 指令MOV实现fib[3]=0+fib[1]+fib[2]

R1 → M[24+**3***8-0]，即2→M[48]，即2→fib[3]

Loop中的7条指令不变
唯一变了的是R2的值

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0] //R0=12 initially
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8]
for i := 2; i < 51; i++ {           MOV 2, R2    // i:=2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1    // label Loop
                                    ADD M[R0+R2*8-16], R1
                                    ADD M[R0+R2*8-8], R1
                                    MOV R1, M[R0+R2*8-0]
                                    INC R2       // i++
                                    CMP 51, R2   // i < 51?
}                                   JL Loop      // if Yes, goto Loop
```



3.3 逐步验证 A step-by-step walkthrough

- 理解“计算机如何支持循环与数组”
- 验证斐波那契计算机满足冯诺依曼机五要点
  - 二级制表示
    - 满足，不过人在逐步验证过程中采用熟悉的十进制
  - P-M-I/O
    - 满足，不过忽略了I/O子系统
  - 存储程序计算机
    - 满足，程序存放在内存地址0~23，数据存放在地址24~431
  - 指令驱动
    - 满足，可在逐步验证中确认
  - 串行执行
    - 满足，可在逐步验证中确认
    - 每一步有两处改变：PC与内存单元（寄存器可看成是特殊的内存单元）

Step 1

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS		0	MOV 0, R1
PC	2	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	0	6	MOV R1, M[R0+8]
R2		8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	//fib[0]
		32	//fib[1]
		40	//fib[2]

Step 2

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS		0	MOV 0, R1
PC	4	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	0	6	MOV R1, M[R0+8]
R2		8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0
		32	
		40	

Step 3

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS		0	MOV 0, R1
PC	6	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	1	6	MOV R1, M[R0+8]
R2		8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0 //fib[0]
		32	//fib[1]
		40	//fib[2]

Step 4

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS		0	MOV 0, R1
PC	8	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	1	6	MOV R1, M[R0+8]
R2		8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0 //fib[0]
		32	1 //fib[1]
		40	//fib[2]

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS		0	MOV 0, R1
PC	10	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	1	6	MOV R1, M[R0+8]
R2	2	8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0 //fib[0]
		32	1 //fib[1]
		40	//fib[2]

Step 5

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS		0	MOV 0, R1
PC	12	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	0	6	MOV R1, M[R0+8]
R2	2	8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0 //fib[0]
		32	1 //fib[1]
		40	//fib[2]

Step 6

Step 7

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS		0	MOV 0, R1
PC	14	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	0	6	MOV R1, M[R0+8]
R2	2	8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0 //fib[0]
		32	1 //fib[1]
		40	//fib[2]

Step 8

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS		0	MOV 0, R1
PC	16	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	1	6	MOV R1, M[R0+8]
R2	2	8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0 //fib[0]
		32	1 //fib[1]
		40	//fib[2]

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS		0	MOV 0, R1
PC	18	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	1	6	MOV R1, M[R0+8]
R2	2	8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0 //fib[0]
		32	1 //fib[1]
		40	1 //fib[2]

Step 9

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS		0	MOV 0, R1
PC	20	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	1	6	MOV R1, M[R0+8]
R2	3	8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0 //fib[0]
		32	1 //fib[1]
		40	1 //fib[2]

Step 10

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS	<	0	MOV 0, R1
PC	22	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	1	6	MOV R1, M[R0+8]
R2	3	8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0 //fib[0]
		32	1 //fib[1]
		40	1 //fib[2]

Step 11

处理器内容		存储器内容	
寄存器	值	地址	指令
FLAGS	<	0	MOV 0, R1
PC	10	2	MOV R1, M[R0]
R0	24	4	MOV 1, R1
R1	1	6	MOV R1, M[R0+8]
R2	3	8	MOV 2, R2
		10 Loop	MOV 0, R1
		12	ADD M[R0+R2*8-16], R1
		14	ADD M[R0+R2*8-8], R1
		16	MOV R1, M[R0+R2*8-0]
		18	INC R2
		20	CMP 51, R2
		22	JL Loop
		24	0 //fib[0]
		32	1 //fib[1]
		40	1 //fib[2]

Step 12

Step 13 Step 14
Step 15 Step 16

请同学们自行补全