



数字逻辑与处理器基础

第五部分

程序与指令系统





程序与指令系统

1

基本概念

2

MIPS体系结构

3

计算机性能评价

4

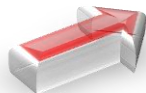
MIPS模拟器

5

MIPS汇编语言程序设计

6

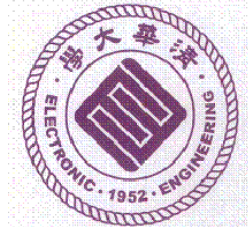
一个排序算法的实例





MIPS模拟器

- 为什么要用模拟器？
 - 个人计算机中一般都是**X86**,系统结构复杂,且不规范,难于理解和编写程序
 - 模拟器能够比实际计算机检测到更多的错误,同时还能提供更多的工具,提供友好的用户编程环境
 - 模拟器是在软件的基础上实现的,可以方便地添加指令集,便于研究计算机及其程序



MIPS模拟器

- **SPIM**

- **SPIM**是主要的**MIPS**模拟器，能够运行和调试**MIPS**汇编语言程序
- **SPIM**支持Uinx、Windows等多个操作系统平台
- <http://pages.cs.wisc.edu/~larus/spim.html>



MIPS模拟器

寄存器窗口

正文段

数据与堆栈段

SPIM消息

```
PCSpin
File Simulator Window Help

PC = 0040000c EPC = 00400000 Cause = 00000024 BadVAddr= 00000000
Status = 3000ff10 HI = 00000000 LO = 00000000
General Registers
R0 (r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 10010000 R24 (t8) = 00000000
R1 (at) = 10010000 R9 (t1) = 00000000 R17 (s1) = 00000000 R25 (t9) = 00000000
R2 (v0) = 00000000 R10 (t2) = 00000000 R18 (s2) = 00000000 R26 (k0) = 00000000

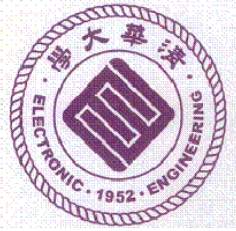
[0x00400000] 0x3c101001 lui $16, 4097 [fibs] ; 7: la $s0, fibs #
[0x00400004] 0x3c011001 lui $1, 4097 [size] ; 8: la $s5, size #
[0x00400008] 0x3435004c ori $21, $1, 76 [size]
[0x0040000c] 0x8eb50000 lw $21, 0($21) ; 9: lw $s5, 0($s5) #
[0x00400010] 0x34120001 ori $18, $0, 1 ; 21: li $s2, 1
[0x00400014] 0xae120000 sw $18, 0($16) ; 22: sw $s2, 0($s0)

DATA
[0x10000000]...[0x1001004c] 0x00000000
[0x1001004c] 0x00000013
[0x10010050] 0x20776f48 0x796e616d 0x62694620 0x63616e6f
[0x10010060] 0x6e206963 0x65626d75 0x74207372 0x6567206f
[0x10010070] 0x6172656e 0x203f6574 0x3c203228 0x2078203d

Memory and registers cleared and the simulator reinitialized.

SPIM Version Version 7.3 of August 26, 2006
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).

For Help, press F1 PC=0x0040000c EPC=0x00400000 Cause=0x00000024
```



MIPS模拟器

- **MARS**

- **MARS** 是 **MIPS Assembler and Runtime Simulator** (MIPS汇编器和运行时模拟器)的缩写
- 能够运行和调试**MIPS**汇编语言程序
- **MARS**采用**Java**开发，跨平台
- <http://courses.missouristate.edu/KenVollmar/MARS/>
- 好用，推荐使用



MIPS模拟器

D:\微机原理\资源\Fibonacci.asm - MARS 4.0 Build 2

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Registers Coproc 1 Coproc 0

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$s8	24	0x00000000
\$s9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffffc
\$fp	30	0x00000000

```
1 # Compute first twelve Fibonacci numbers and put in array, then print
2 .data
3 fibs: .word 0 : 12      # "array" of 12 words to contain fib values
4 size: .word 12         # size of "array"
5 .text
6     la $t0, fibs        # load address of array
7     la $t5, size        # load address of size variable
8     lw $t5, 0($t5)      # load array size
9     li $t2, 1           # 1 is first and second Fib. number
10    add.d $f0, $f2, $f4
11    sw $t2, 0($t0)       # F[0] = 1
12    sw $t2, 4($t0)       # F[1] = F[0] = 1
13    addi $t1, $t5, -2    # Counter for loop, will execute (size-2) times
14loop: lw $t3, 0($t0)     # Get value from array F[n]
15    lw $t4, 4($t0)       # Get value from array F[n+1]
16    add $t2, $t3, $t4    # $t2 = F[n] + F[n+1]
17    sw $t2, 8($t0)       # Store F[n+2] = F[n] + F[n+1] in array
18    addi $t0, $t0, 4     # increment address of Fib. number source
19    addi $t1, $t1, -1    # decrement loop counter
20    bgtz $t1, loop       # repeat if not finished yet.
21    la $a0, fibs         # first argument for print (array)
22    add $a1, $zero, $t5  # second argument for print (size)
23    jal print            # call print routine.
24    li $v0, 10           # system call for exit
25    syscall              # we are out of here.
```

Line: 1 Column: 1 ☒ Show Line Numbers

Mars Messages Run I/O

Clear



程序与指令系统

1

基本概念

2

MIPS体系结构

3

计算机性能评价

4

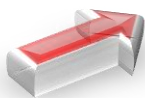
MIPS模拟器

5

MIPS汇编语言程序设计

6

一个排序算法的实例





MIPS汇编语言程序设计

- Ref.2/2.9,2.10,附录A
- MIPS汇编语言语句格式
- 汇编命令(directive)
- MIPS汇编器语法
- 存储器中位置
- 编程指南



MIPS汇编语言程序设计

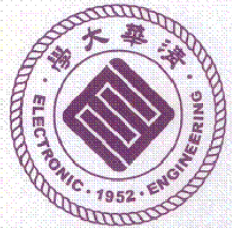
MIPS汇编语言语句格式

- 指令（直接对应机器码）与伪指令语句
（汇编器将其翻成指令）

[Label:] <op> Arg1, [Arg2], [Arg3] [#comment]

- 汇编命令(**directive**)语句（不变成机器码）

[Label:] .Directive [arg1], [arg2], ... [#comment]

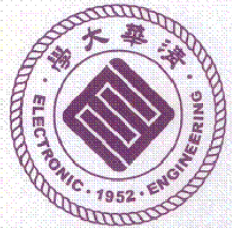


MIPS汇编语言程序设计

汇编命令(directive)

- 汇编器用来定义数据段、代码段以及为数据分配存储空间

.data [address] #定义数据段, [address]为可选的地址
.text [address] #定义正文段(即代码段)
.align n #以 2^n 字节边界对齐数据,只能用于数据段
.ascii <string> # 在内存中存放字符串
.asciiz <string> # 在内存中存放以NULL结束的字符串
.word w1, w2, ..., wn # 在内存中存放n个字
.half h1, h2, ..., hn # 在内存中存放n个半字
.byte b1, b2, ..., bn # 在内存中存放n个字节



MIPS汇编语言程序设计

MIPS汇编器语法

- **ref.2(P470/A.10.2)**
- 注释行以“#”开始
- 标识符由字母、下划线（_）、点（.）构成，但不能以数字开头
 - 指令操作码是一些保留字，不能用作标识符
- 标记放在行首，后跟冒号（:）

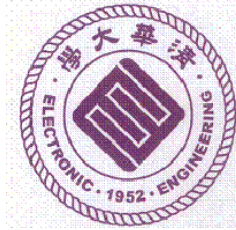
```
.data          #将子数据项，存放到数据段中
Item: .word 1,2  #将2个32位数值送入地址连续的内存字中
.text          #将子串即指令或字送入用户文件段
.global main   #必须为全局变量
Main: lw $t0, item
```



MIPS汇编器：存储器中位置

- 汇编语言源文件：.s
 - “.” MIPS汇编命令标识符
 - “label:”
 - label被赋值为当前位置的地址
 - fact = 0x00400100
 - 编译时就确定了
 - 汇编程序在地址0x00400000开始

```
0x00400020  move $s5, $31
0x00400024  beq $0,$0,fact
0x00400028  sw  $s0,f($0)
...
.text 0x00400100
0x00400100  fact: addiu $s0,$0,1
0x00400104          lw  $s1,n($0)
0x00400108  loop: mul  $s0,$s1,$s0
0x0040010C          subu $s1,$s1,1
0x00400110          bnez $s1,loop
0x00400114          addi $31,$s5,4
0x00400118          jr   $31
...
.data 0x10000200
0x10000200  n:   .word 4
0x10000204  f:   .word 0
```



能运行的版本 (1)

.text

main:

```
ori $s6,$0,0x1000
sll $s6,$s6,16
addiu $s4,$s6,0x0200 #$s4=n
addiu $s5,$s6,0x0204 #$s5=f
beq $0,$0, fact
```

```
result: sw $s0,0($s5)
        jr $ra #跳出main
```

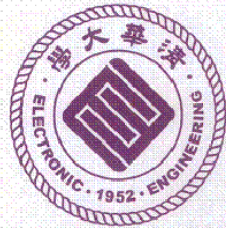
.text 0x00400100

fact:

```
addiu $s0,$0,1
lw $s1,0($s4) #$s0=n!
loop: mul $s0,$s1,$s0
      subu $s1,$s1,1
      bnez $s1,loop #f=n!
      j result
```

.data 0x10000200

```
n: .word 4
f: .word 0
```



能运行的版本（2）

```
.data 0x10000000  
.word 4,0  
.text
```

main:

```
ori    $s6,$0,0x1000
```

```
sll     $s6,$s6,16
```

```
addiu   $s5,$s6,0x004
```

```
fact:   addiu   $s0,$0,1
```

```
lw      $s1,0($s6)
```

```
loop:   mul     $s0,$s1,$s0
```

```
addi    $s1,$s1,-1
```

```
bnez    $s1,loop
```

```
sw      $s0,0($s5)
```

```
jr $ra
```

#下列语句行是数据代码行

#定义了两个字型立即数4和0

#下列语句行是指令代码行

获得数据起始地址

\$s6=0x10000000

\$s5=0x10000004

循环计数器赋初值

把 word型数 4 载入 \$s1

\$s0=n!, n=4

对应伪指令 subu \$s1,\$s1,1

#

f=n!=24

#根据ra寄存器中的返回地址返回

编程指南

- (1) 变量
- (2) 分支
- (3) 数组
- (4) 过程调用
- (5) 阅读、改进程序
- (6) 设计实例

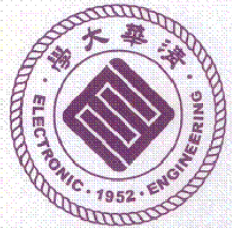


单指令计算机



编程指南： (1) 变量

- 变量存储在主存储器内（而不是寄存器内）
 - 因为通常有不止**32**个变量要存
- 为了实现功能, 用**LW** 语句将变量加载到寄存器中, 对寄存器进行操作, 然后再把结果**SW**回去
- 对于比较长的操作(e.g., loops):
 - 让变量在寄存器中保留时间越长越好
 - **LW**和**SW** 只在一块代码开始和结束时使用
 - 节省指令
 - 事实上**LW**和**SW** 比寄存器操作要慢得多得多！
- 由于一条指令最多采用两个输入, 所以必须采用临时寄存器计算复杂的问题（e.g., $(x+y)+(x-y)$ ）



编程指南： (1) 变量

```
.data 0x10000000  
.word 4,0  
.text
```

lui \$s6, 0x1000

```
main: ori $s6,$0,0x1000
```

```
sll $s6,$s6,16
```

```
addiu $s5,$s6,4
```

```
fact: addiu $s0,$0,1
```

```
lw $s1,0($s6)
```

#s1 get 4

```
loop: mul $s0,$s1,$s0
```

```
subu $s1,$s1,1
```

```
bnez $s1,loop
```

```
sw $s0,0($s5)
```

#s0 hold result, 将变量结果放在内存中

```
jr $ra
```

#return result in s0





编程指南： (2) 分支

- 在标识符汇编语句中,分支语句的目标位置是用绝对地址方式写的

– `beq $0,$0,fact`

– **`PC ← 0x00400100`**

- 在机器运行代码时,要用相对于PC的地址（偏移量）来定义

– `beq $0,$0,0x3f`

– **`PC ← 0x00400100`**

```
.text
main:  ori $s6,$0,0x1000
      sll $s6,$s6,16
      addiu $s4,$s6,0x0200
      addiu $s5,$s6,0x0204
      addiu $s7,$s6,0x0208
      beq $0,$0, fact
result: sw $s0,0($s5)
      jr $ra
.text 0x00400100
fact: sw $ra,0($s7)
      addiu $s0,$0,1
      lw $s1,0($s4)    # $s0=n!
loop:  mul $s0,$s1,$s0
      subu $s1,$s1,1
      bnez $s1,loop    # f=n!
      j  result
.data 0x10000200
n:     .word 4
f:     .word 0
```



编程指南：（2）分支

- **偏移量**= 从下一条指令对应的PC开始到标号位置还有多少条指令
 - beq \$0,\$0, fact 如果位于地址 **0x00400000** 的话,
偏移量
$$= (\text{fact 所处位置} - (<\text{PC}> + 4)) / 4$$
$$= (0x00400100 - 0x00400004) / 4$$
$$= 0xfc / 4 = \mathbf{0x3f}$$
 - 偏移量为0则表示执行下一条指令不产生任何跳转
- **为什么在代码中用相对的偏移量?**
 - 可重新定位(*relocatable*)的代码
 - 分支语句可以在每次被加载到内存不同位置的情况下正常工作



编程指南： (2) 分支

- 如果和0比较，则直接使用**blez**, **bgez**, **bltz**, **bgtz**, **bnez**
 - 如前例的**loop**
- 更复杂的比较，采用比较指令（如**slt**），然后再用与0比较
- 实例

```
if ( x >= 0 )  
    y = x;  
else  
    y = -x;
```



编程指南: (2) 分支

test2

```
.data 0x10000000
.word -6,0          #x: -6, y: 0
.text
main: ori    $s6,$0,0x1000 #计算内存中数据存放地址
      sll    $s6,$s6,16
      addiu  $s5,$s6,4      # $s6=x, $s5=y
      lw     $s0,0($s6)
      slt    $s2,$0,$s0     # 0<x, $s2=1
      beqz   $s2,else      # $s2=0, 跳到else
      move   $s1,$s0
      j done              # $s2=1, 跳到done
else:  sub    $s1,$0,$s0
done:  sw     $s1,0($s5)
      jr     $ra
```

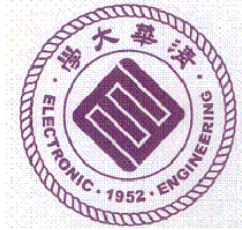
功能：
求绝对值





编程指南: (3) 数组array

- 用 **.word**来给数组开辟空间
 - 在编译时 *静态地*开辟 $n*4$ bytes (n 个32-bit字)
- 使用**LW**和**SW**时, 要设数组基址
 - LW \$1, const(\$2)**
 - SW \$3, const(\$4)**
 - 将 **const** 作为数组偏移量
 - 将寄存器**\$2**和**\$4**作为数组中的开始地址(**A[0]**)



test3

编程指南: (3) 数组array

.data 0x10000000

.word 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17

.text

main: addu \$s7,\$ra,\$0

ori \$s5,\$0,0x1000

sll \$s5,\$s5,16

#\$s5=A[]=0x10000000,

addiu \$s6,\$s5,0x400

#\$s6=B[]=0x10000400

addiu \$s0,\$0,0x11

#Size(A)=Size(B)=0x11

loop: subu \$s0,\$s0,1

addiu \$s1,\$0,4

mul \$s2,\$s1,\$s0

addu \$s3,\$s2,\$s5

#计算A[]偏移量,送到\$s3

lw \$s4,0(\$s3)

#读出A[]中的值

addu \$s3,\$s2,\$s6

#计算B[]偏移量,送到\$s3

sw \$s4,0(\$s3)

#写到B[]中去

bnez \$s0,loop

addu \$ra,\$0,\$s7

#返回调用程序

jr \$ra



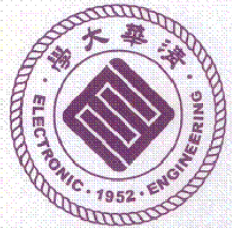
编程指南：(3) 数组array——数组访问

- 对无符号数
 - SLLV by k 等价于 MUL by 2^k
 - SRLV by k 等价于 DIV by 2^k
 - 很有用，因 MUL 和 DIV 一般都比 SLLV 和 SRLV 慢
 - 想想怎么实现 MUL 和 DIV?
- 对于有符号数
 - SRA: 算术右移
 - 高位用符号位填充(在2补表示情况下)
 - e.g.,
R1 = -6 = 0b11...11010
SRL \$R1,\$R1,1 → 0b01...11101 ×
SRA \$R1,\$R1,1 → 0b11...11101 = -3 ✓
 - 想想为什么这样是对的 ...

编程指南: (3) 数组array——数组访问



```
add    $s0,$0,$a2    # i = N    数组长度N = 0x100, 存放在$a2中
loop: subu $s0 $s0,1    # i--    数组长度计数器$s0
addiu  $s1,$0,4    # $s1=4
mul    $s2,$s1,$s0    # $s2 = i*4
addiu  $t0,$0,2    # $t0=2
div    $s0,$t0    # 商寄存$lo=floor(i/2)
mflo   $t1    # $t1=floor(i/2)
mul    $t1,$s1,$t1    # $t1=$t1*4    计算4字节地址偏移量
add    $t2,$t1,$a0    # 计算数组A地址    数组A的基地址存放在$a0中
lw     $s4,0($t2)    # $s4=A[i/2]    取对应的数组元素
add    $t3,$s2,$a1    # 计算数组B地址    数组B的基地址存放在$a1中
sw     $s4,0($t3)    # B[i] = $s4
bnez   $s0,loop    # while(i!=0) loop
done: ...
```



编程指南: (3) 数组array——数组访问 用移位代替乘法

```
add $s0,$0,$a2      # i = N
loop:
subu  $s0,$s0,1      # i--
sll   $s2,$s0,2       # dest = i*4
sra   $t1,$s0,1       # $t1=floor(i/2)
sll   $t1,$t1,2       # $t1=$t1*4
add   $t2,$t1,$a0
lw    $s4,0($t2)      # $s4=A[i/2]
add   $t3,$s2,$a1
sw    $s4,0($t3)      # B[i] = $s4
bnez  $s0,loop        # while(i!=0) loop
done: ...
```

- 对无符号数
 - SLL by k 等价于 MUL by 2^k
 - SRL by k 等价于 DIV by 2^k
- 对于有符号数用 SRA
 - 高位用符号位填充 (在2补表示情况下)





编程指南：（4）过程调用

- 用汇编编写程序时需要明确说明每一次的调用和返回
 - **Why?** 为什么用**C**编写程序时不需要了解子程序被调用的具体细节?

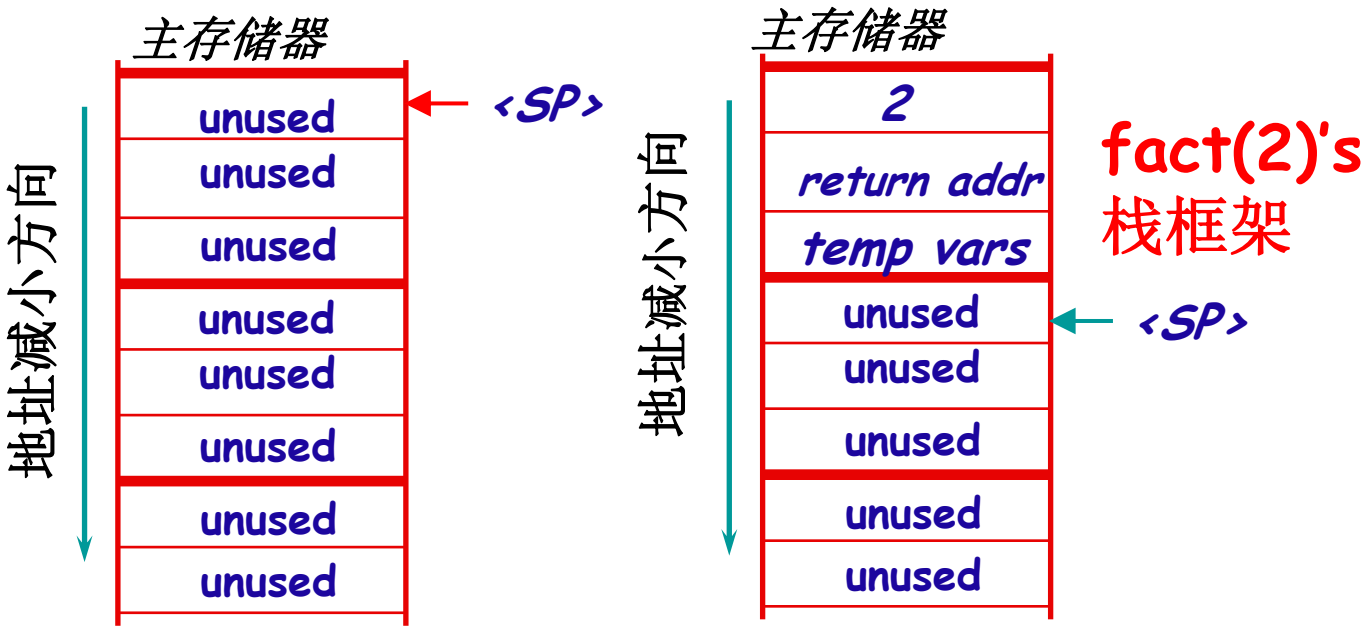
编译程序替你做了！
- 程序的调用和返回严格遵从后进先出（**LIFO**）原则
- 需要存储：
 - 返回地址(**old ra**) —参数 **x, y, z** in **f(x, y, z)**
 - 临时/局部的变量 (**f执行过程中**) —被破坏的寄存器
- 想法：存在栈里！
 - 增长(**PUSH** 进去) 栈，每次调用函数时
 - 缩减 (**POP** 出来)栈，每次返回时
 - 每次调用都有自己的“栈框架”

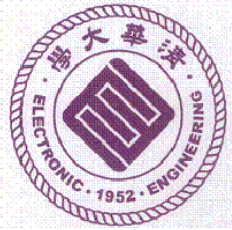


编程指南: (4) 过程调用

```
int fact( int n )
{
    if ( n > 0 )
        return fact( n-1 ) * n;
    else
        return 1 ;
}
```

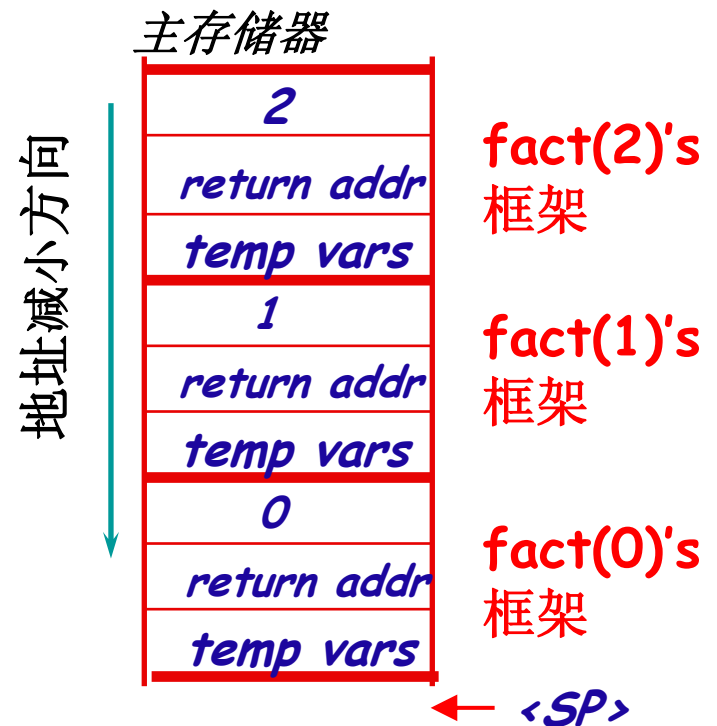
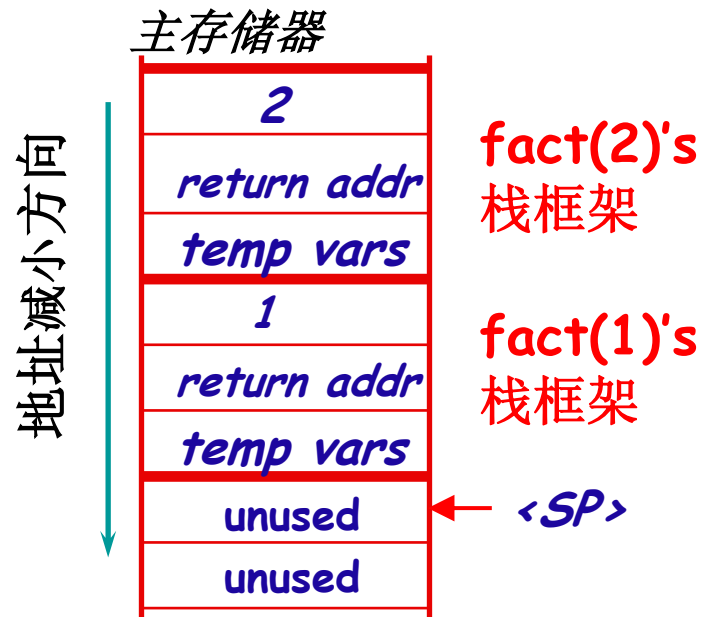
例如
调用fact(2)





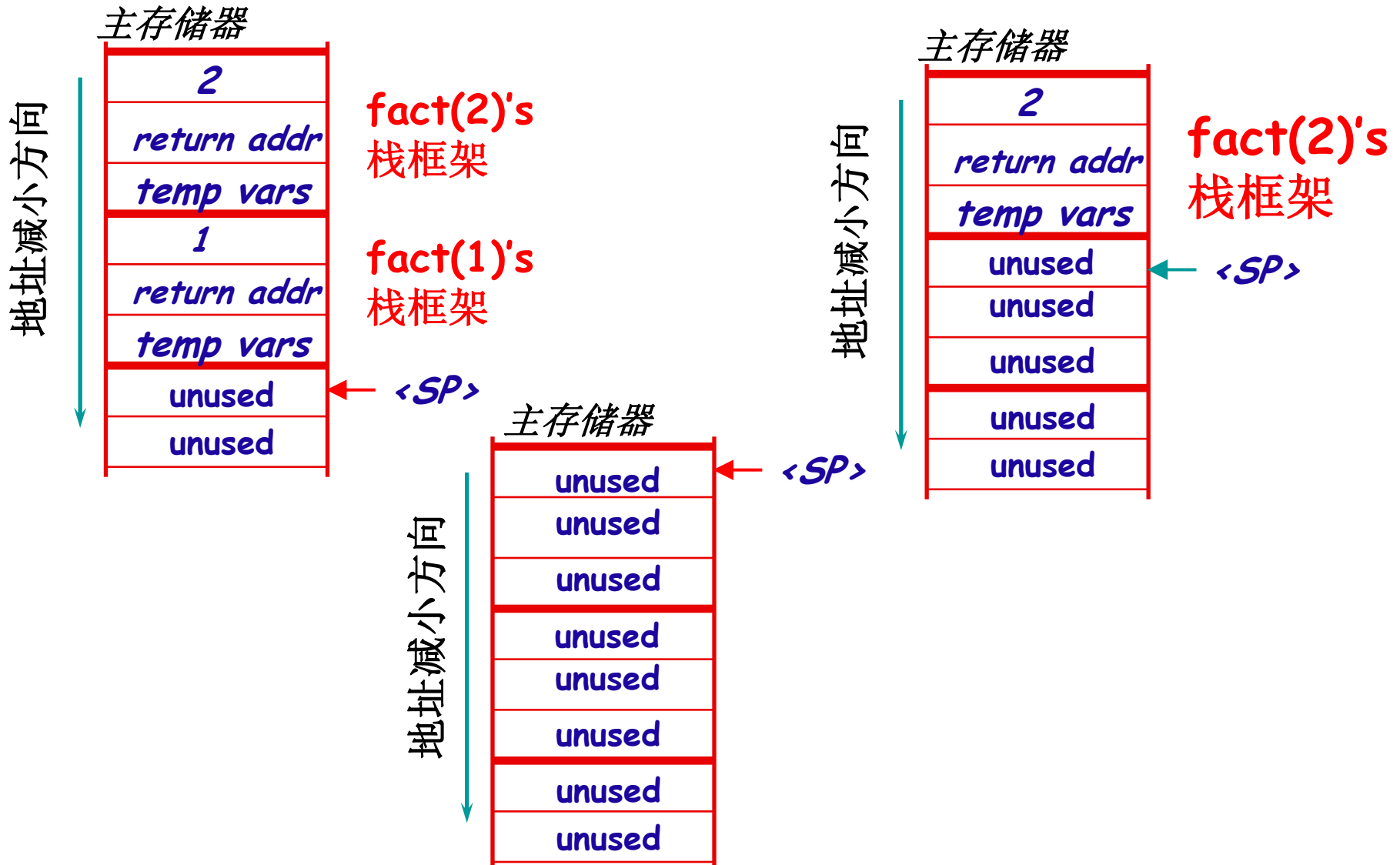
使用栈： 栈框架

- 需要存储：
 - 返回地址(old ra)
 - 参数 n in *fact(1)/fact(0)*
 - 临时/局部的变量 (*fact*执行过程中)
 - 被破坏的寄存器





使用栈： 出栈





MIPS计算机硬件对过程的支持

- 为新数据分配空间
 - 利用堆栈存储过程中不适合用寄存器保存的局部变量（如局部数组或结构）
 - **过程框架**：也叫活动记录，是指包含了过程保存的寄存器和局部变量的堆栈段
 - 通过下例了解过程调用**之前**、**之中**和**之后**的堆栈状态

\$sp → 7fff ffff hex

MIPS 程序和数据的存储器空间使用约定

- 从顶端开始，对栈指针初始化为7fffffff，并向向下向数据段增长；
- 在底端，程序代码（文本）开始于00400000；
- 静态数据开始于10000000；
- 紧接着是由C中malloc进行存储器分配的动态数据，朝堆栈段向上增长

全局指针被设定为易于访问数据的地址，以便使用相对于\$gp的±16位偏移量

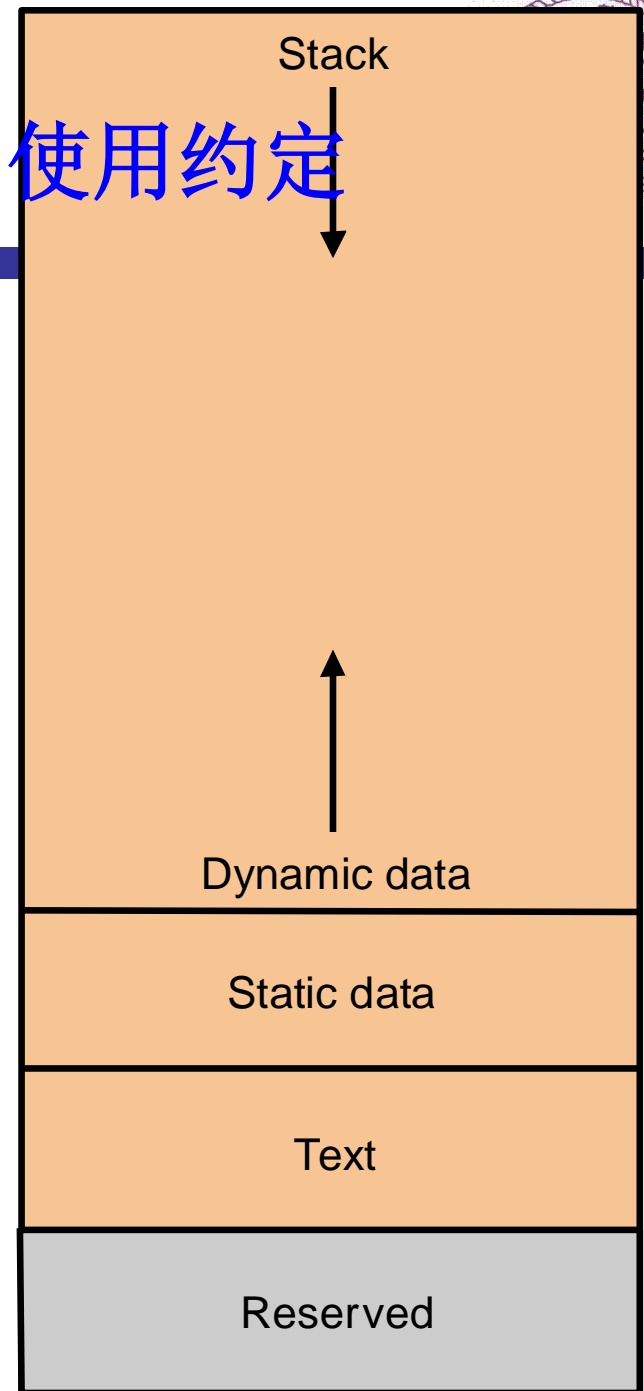
$10000000_{\text{hex}} - 1000ffff_{\text{hex}}$

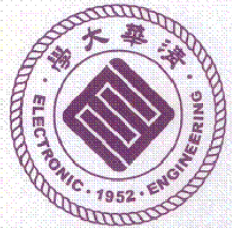
\$gp → 1000 8000 hex

1000 0000 hex

pc → 0040 0000 hex

0

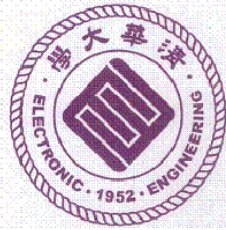




数据存在哪？

```
.data 0x10000000
.word 4,0
.text
main:  ori $s6,$0,0x1000
      sll $s6,$s6,16
      addiu $s5,$s6,4
fact:  addiu $s0,$0,1
      lw $s1,0($s6)           #s1 get 4
loop:  mul $s0,$s1,$s0
      subu $s1,$s1,1
      bnez $s1,loop
      sw $s0,0($s5)          #s0 hold result
      jr $ra                 #return result in s0
```





编程指南: (5) 阅读、改进汇编代码

.data

#x's address stored in \$a0

f's address stored in \$a1

.text

move \$s1,\$0

move \$s2,\$s1

addiu \$s3,\$0,1

lw \$s4,0(\$a0)

loop:

addi \$s4,\$s4,-1

beqz \$s4,done

move \$s1,\$s2

move \$s2,\$s3

addu \$s3,\$s2,\$s1

j loop

done:

sw \$s3,0(\$a1)

• 这段程序干了啥?

#\$s1=0

#\$s2=\$s1

#\$s3=1

#\$s4=x

#\$s4=\$s4-1

#if(\$s4=0)done

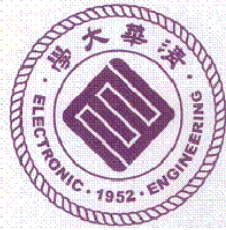
#\$s1=\$s2

#\$s2=\$s3

#\$s3=\$s2+\$s1

#goto loop

#store result in f



编程指南: (5) 阅读、改进汇编代码

.data

#x's address stored in \$a0

f's address stored in \$a1

.text

move \$s1,\$0 **#\$s1=0**

move \$s2,\$s1 **#\$s2=\$s1**

addiu \$s3,\$0,1 **#\$s3=1**

lw \$s4,0(\$a0) **#\$s4=x**

loop:

addi \$s4,\$s4,-1 **#\$s4=\$s4-1**

beqz \$s4,done **#if(\$s4=0)done**

move \$s1,\$s2 **#\$s1=\$s2**

move \$s2,\$s3 **#\$s2=\$s3**

addu \$s3,\$s2,\$s1 **#\$s3=\$s2+\$s1**

j loop **#goto loop**

done:

sw \$s3,0(\$a1) **#store result in f**

• 这段程序干了啥?

a = 0; // a = s1

b = a; // b = s2

res = 1;

i = x;

while (--i != 0)

// decrease i then check if 0

{ a=b;

b=res;

res=a+b;

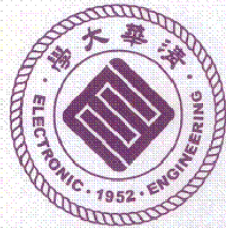
}

f = res;

// note x itself isn't changed

x = 1 2 3 4 5 6 7 8 9 10

f =



编程指南: (5) 阅读、改进汇编代码

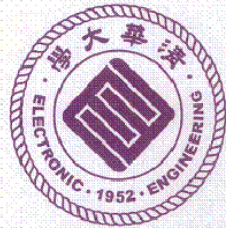
```
.data
#x's address stored in $a0
# f's address stored in $a1
.text
    move $s1,$0      #$s1=0
    move $s2,$s1      #$s2=$s1
    addiu $s3,$0,1     #$s3=1
    lw $s4,0($a0)      #$s4=x
loop:
    addi $s4,$s4,-1    #$s4=$s4-1
    beqz $s4,done      #if($s4=0)done
    move $s1,$s2        #$s1=$s2
    move $s2,$s3        #$s2=$s3
    addu $s3,$s2,$s1    #$s3=$s2+$s1
    j loop              #goto loop
done:
    sw $s3,0($a1)      #store result in f
```

```
a = 0; // a = s1
b = a; // b = s2
res = 1;
i = x;
while ( --i != 0 )
    // decrease i then check if 0
{ a=b;
  b=res;
  res=a+b;
}
f = res;
// note x itself isn't changed
```

x =	1	2	3	4	5	6	7	8	9	10
f =	1	1	2	3	5	8	13	21	34	55

前两项均为1，从第三项起，每一项都是其前两项的和

f = fib(x) (for x > 0) (Fibonacci)



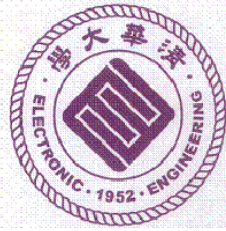
编程指南: (5) 阅读、改进汇编代码

```
.data
#x:  $a0
#f:  $a1
.text
move $s1,$0      #$s1=0
move $s2,$s1     #$s2=$s1
addiu $s3,$0,1    #$s3=1
lw    $s4,0($a0)  #$s4=x

loop:
addi  $s4,$s4,-1  #$s4=$s4-1
beqz  $s4,done    #if($s4=0)done
move  $s1,$s2     #$s1=$s2
move  $s2,$s3     #$s2=$s3
addu  $s3,$s2,$s1 #$s3=$s2+$s1
j     loop        #goto loop

done:
sw    $s3,0($a1)  #store result in f
```

- 如何使本部分代码运行更快?
- 目前运行时间:
 $= \text{const} + 6x$
- 能否达到
 $\text{const} + 5x$?



编程指南: (5) 阅读、改进汇编代码

```
move $s1,$0      # $s1=0
move $s2,$s1     # $s2=$s1
addiu $s3,$0,1   # $s3=1
lw $s4,0($a0)    # $s4=x
subu $s4,$s4,1   # $s4=$s4-1
beqz $s4,done    # if($s4=0) done

loop:
move $s1,$s2     # $s1=$s2
move $s2,$s3     # $s2=$s3
addu $s3,$s2,$s1 # $s3=$s2+$s1
addi $s4,$s4,1   # $s4=$s4-1
bnez $s4,loop    # while($s4!=0) goto loop

done:
sw $s3,0($a1)    # store result in f

...
```

- 运行时间从 **const+6x** 到 **const+5x**
- Trick: 将 SUB 和分支检查到后面, 减掉额外的分支指令
- 当x比较大时, 减少了运行时间 (constant有所增加)
- 目前的编译器一般都可以支持这类优化





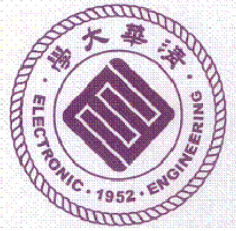
设计实例

- Hello world

```
        .text
main:
        la      $a0, str
        li      $v0, 4
        syscall                                # print string
        li      $v0, 10
        syscall                                # exit

        .data

str:
        .asciiz "Hello world."
```

MIPS汇编语言程序设计

系统调用

- **MIPS**模拟器提供了通过系统调用指令(**syscall**) 提供了一组类似操作系统的服务
- 调用方法:
 - 将系统调用代码装入**\$v0(\$2)**寄存器
 - 将参数(如果有)装入**\$a0(\$4)~\$a3(\$7)**或**\$f12**寄存器
 - **Syscall**
 - 返回值保存在**\$v0(\$2)**或**\$f0**寄存器中



MIPS汇编语言程序设计

系统调用

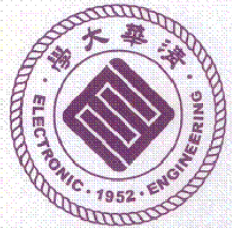
代码	系统调用	参数	结果
1	print integer	\$a0	
2	print float	\$f12	
3	print double	\$f12	
4	print string	\$a0	
5	read integer		integer in \$v0
6	read float		float in \$f0
7	read double		double in \$f0
8	read string	\$a0=buffer, \$a1=length	
9	sbrk	\$a0=amount	address in \$v0
10	exit		



MIPS汇编语言程序设计

系统调用

代码	系统调用	参数	结果
11	print char	\$a0	
12	read char		char in \$v0
13	open	\$a0=file name(string), \$a1=flags, \$a2=mode	file descriptor (fd) in \$v0
14	read	\$a0 =fd, \$a1=buffer, \$a2=length	num chars read in \$v0
15	write	\$a0 =fd, \$a1=buffer, \$a2=length	num chars write in \$v0
16	close	\$a0 =fd	
17	exit2	\$a0=result	



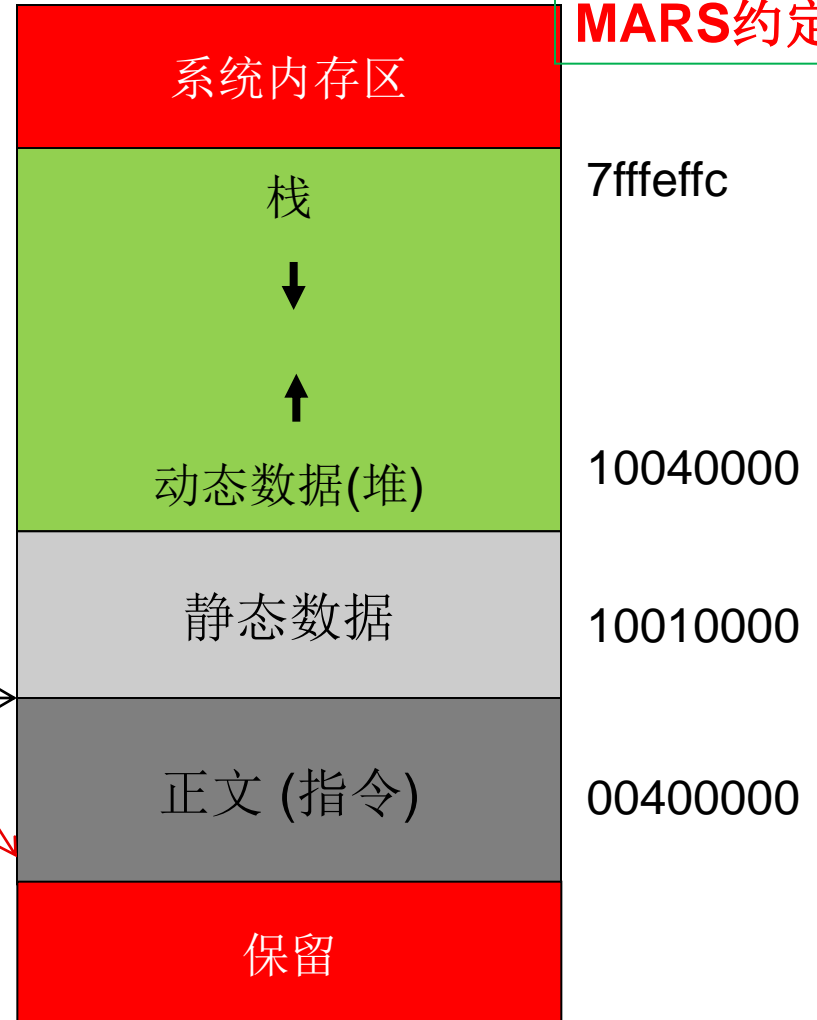
MIPS汇编语言程序设计

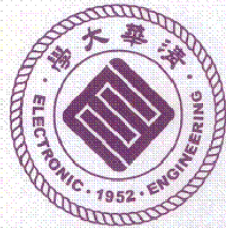
MARS约定

```
.text
main:
    la    $a0, str
    li    $v0, 4
    syscall

    li    $v0, 10
    syscall

.data
str:
    .asciiz "Hello world."
```





MIPS汇编语言程序设计

- 从键盘输入两个数，计算并输出这两个数的和

```
.data
    str1: .asciiz "Enter 2 numbers:"
    str2: .asciiz "The sum is "

.text
main:
    li $v0, 4
    la $a0, str1
    syscall

    li $v0, 5
    syscall
```

```
add $t0, $v0, $zero
li $v0, 5
syscall

add $t1, $v0, $zero
li $v0, 4
la $a0, str2
syscall

li $v0, 1
add $a0, $t1, $t0
syscall
```



MIPS汇编语言程序设计

- 计算 $1^2+2^2+\dots+100^2$

```
        .text
main:
        li      $t0, 1
        li      $t8, 0
loop:
        mul     $t7, $t0, $t0
        add     $t8, $t8, $t7
        addi    $t0, $t0, 1
        ble     $t0, 100, loop

        la      $a0, str
```

```
        li      $v0, 4
        syscall
        li      $v0, 1
        move    $a0, $t8
        syscall
        li      $v0, 10
        syscall

        .data
        .align 2
str:     .ascii "The sum of square from
1 to 100 is "
```



程序与指令系统

1

基本概念

2

MIPS体系结构

3

计算机性能评价

4

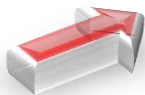
MIPS模拟器

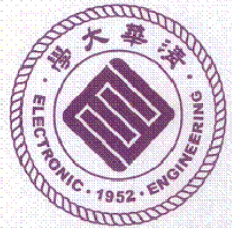
5

MIPS汇编语言程序设计

6

一个排序算法的实例





一个排序算法的实例

```
void sort (int v[], int n)
{
    int i, j;
    for (i=0; i<n; i++) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j--) {
            swap (v,j);
        }
    }
}
```

• Ref.2/2.13

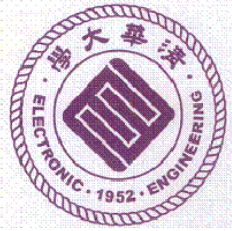
```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

从C程序手动转化为汇编程序:

- ✓为程序中变量分配寄存器
- ✓把过程体转化为汇编代码
- ✓保存跨过程调用的寄存器

对数组中的元素进行排序

叶过程: 交换数组中的元素



swap过程

- 寄存器分配:
- ✓ **\$a0**——参数**v**
- ✓ **\$a1**——参数**k**
- ✓ **\$t0**——变量**temp**

```
swap:  sll    $t1, $a1, 2
        add    $t1, $a0, $t1
        lw     $t0, 0($t1)
        lw     $t2, 4($t1)
        sw     $t2, 0($t1)
        sw     $t0, 4($t1)
        jr     $ra
```

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

从内存中取出

将v[k]和

v[k+1]的值交换写入内存中

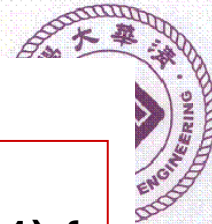
- 因为不使用**\$s0-\$s7**，并且此过程为叶过程，不会重用**\$a0**和**\$a1**，所以无需保存和恢复寄存器

sort过程

```
void sort (int v[], int n)
{
    int i, j;
    for (i=0; i<n; i++) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j--) {
            swap (v,j);
        }
    }
}
```

- 寄存器分配:
 - ✓ \$a0——参数v
 - ✓ \$a1——参数n
 - ✓ \$s0——变量i
 - ✓ \$s1——变量j
- 过程体包括两个嵌套的for循环和一个带参数的swap调用
- 外循环框架

```
loopbody1:    move    $s0, $zero        # 循环初始化
              bge     $s0, $a1, exit1   # 伪指令i≥n, 则转移至exit1
              ... 内循环循环体 ...
              addi    $s0, $s0, 1       # i=i+1
              j       loopbody1
exit1:
```



sort过程

- 内循环框架

```
for (i=0; i<n; i+=1) {  
    for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {  
        swap (v,j);  
    }  
}
```

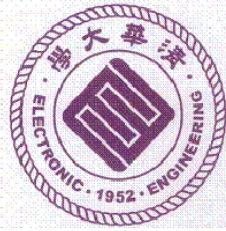
```
loopbody2:    addi    $s1, $s0, -1  
              blt     $s1, $zero, exit2    #伪指令j<0则转移至exit2  
              sll     $t1, $s1, 2  
              add     $t2, $a0, $t1  
              lw      $t3, 0($t2)  
              lw      $t4, 4($t2)  
              ble     $t3, $t4, exit2    #伪指令v[j]≤v[j+1]则转移至exit2  
              ... 调用swap...  
              addi    $s1, $s1, -1  
              j       loopbody2  
exit2:
```

sort调用swap

```
for (i=0; i<n; i+=1) {  
    for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {  
        swap (v,j);  
    }  
}
```

- 在调用**swap**前需要将参数**v**和**j**送入寄存器**\$a0**和**\$a1**，但是**sort**过程需要使用**\$a0**和**\$a1**的值，因此在调用**swap**前必须保存**\$a0**和**\$a1**的值，可将其复制到**\$s2**和**\$s3**中

```
move    $s2, $a0  
move    $s3, $a1  
...  
move    $a0, $s2    # $s2=v  
move    $a1, $s3    # $s3=j  
jal     swap
```



在sort中保存与恢复寄存器

- **sort**过程修改了**\$s0~\$s3**，并且调用了**swap**过程，因此在过程头必须保存**\$s0~\$s3**和**\$ra**，在过程尾恢复这些寄存器

```
sort:  addi    $sp, $sp, -20
        sw     $ra, 16($sp)
        sw     $s3, 12($sp)
        sw     $s2, 8($sp)
        sw     $s1, 4($sp)
        sw     $s0, 0($sp)
        ...
```

```
...
exit1: lw     $s0, 0($sp)
        lw     $s1, 4($sp)
        lw     $s2, 8($sp)
        lw     $s3, 12($sp)
        lw     $ra, 16($sp)
        addi    $sp, $sp, 20
        jr      $ra
```



课后作业8

- 国际象棋中的皇后可以吃掉与它在同一行、同一列、同一对角线上的棋子。八皇后问题，即在 8×8 的国际象棋棋盘上放置8个皇后，要求任意两个皇后不能在同一行、同一列或同一条对角线上。求出如此放置方法的种数
- 一种解决问题的思路是一行放置皇后，如果当前放置的皇后与前面的皇后不存在冲突时，则继续摆下一个皇后，否则跳到上一个皇后，重新摆置
- 编写一个**MIPS**软件，要求和作业指南中的程序功能相同，尤其请注意其中的键盘输入和打印功能也需要实现
- 网上提交时间：**4月30日**