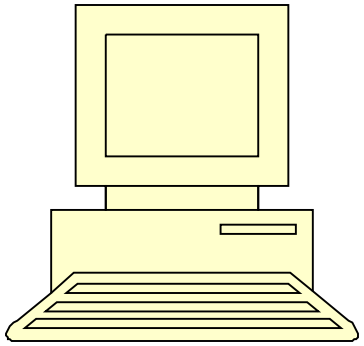
MIPS指令与汇编程序设计

- 在模拟器（MARS、SPIM）上调试如下程序：
  1. 循环分支程序：斐波那契数列生成器
  2. 系统调用程序：Hello World
  3. 过程调用程序：排序算法
- 写测试报告（截屏运行过程）

MIPS指令与汇编程序设计



所有的MIPS指令都是32位长

1. 指令格式
2. 寻址方式
3. 指令系统

- ✓R-format: 所有其他
- ✓I-format: 用于有立即数的指令,

  lw, sw, beq, bne
- ✓J-format: 无条件跳转 j, 并连接jal

关于指令格式的思考

MIPS 算术指令

指令	*例子*	*含义*	*说明*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 × $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 × $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

计算机指令系统(2)

- **MIPS模拟器**
- **MIPS汇编语言程序设计**
- 一个排序算法的实例

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Unix、Windows等多个操作系统平台
  - http://pages.cs.wisc.edu/~larus/spim.html

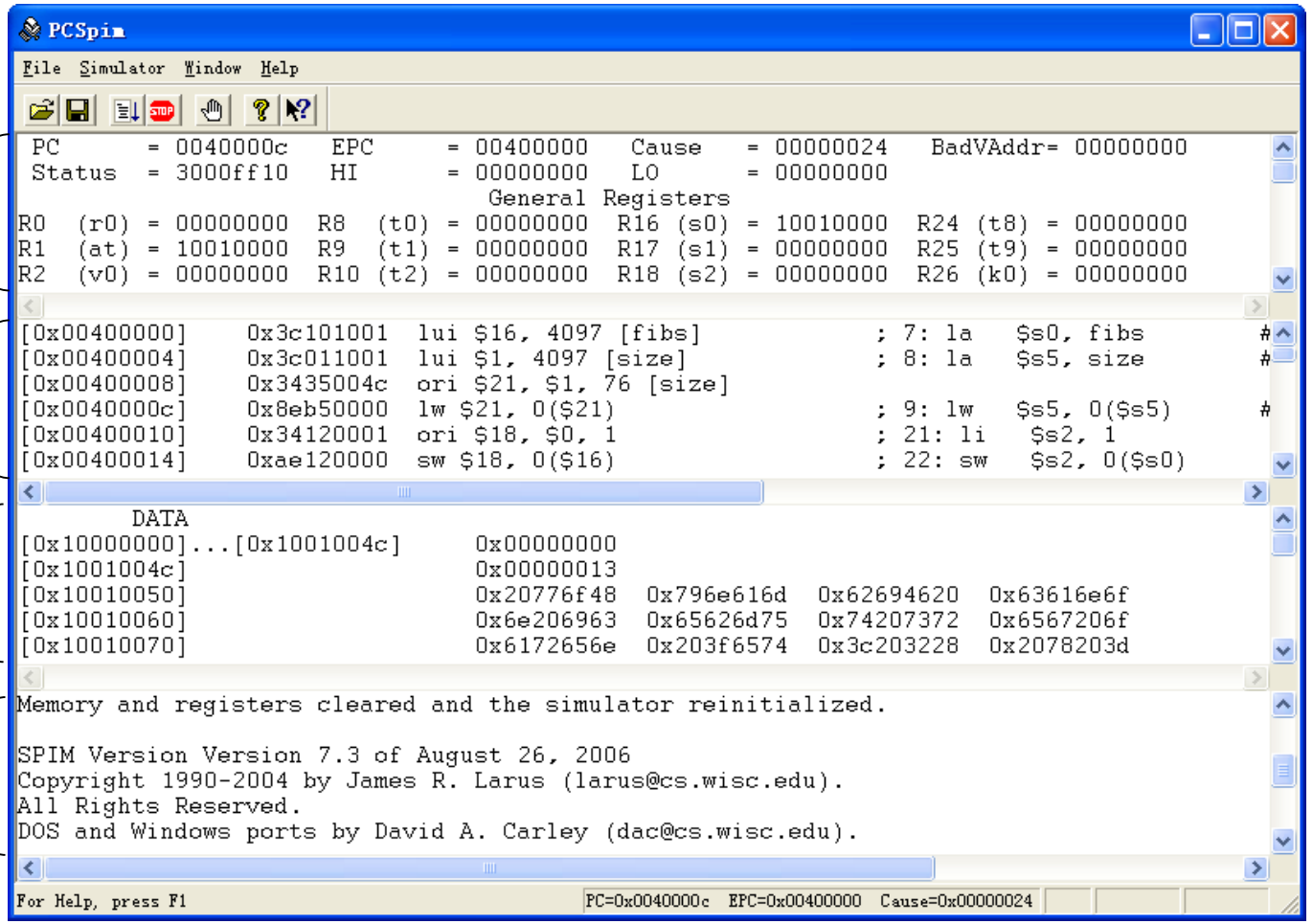
MIPS模拟器

寄存器窗口

正文段

数据与堆栈段

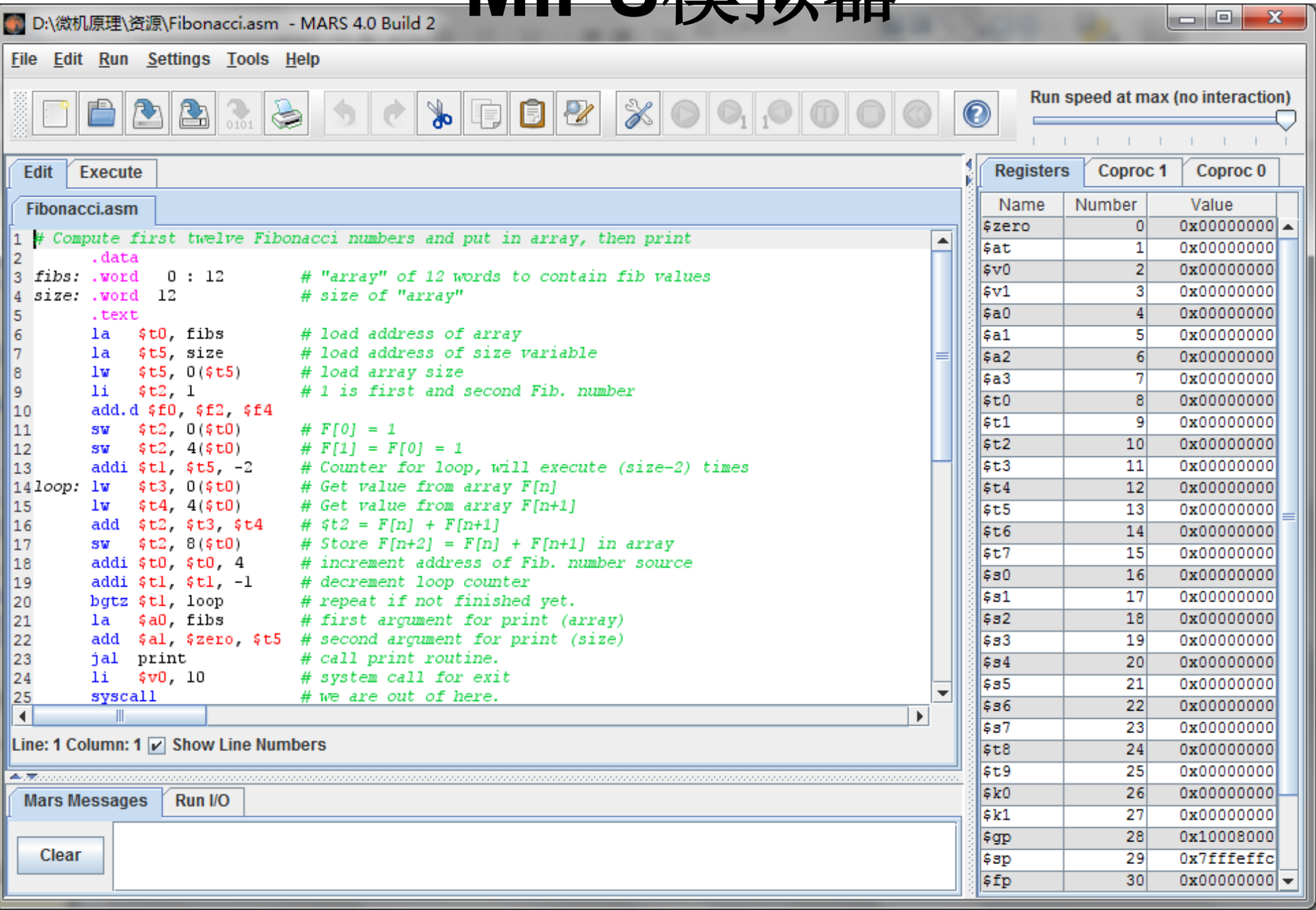
SPIM消息



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/

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)
Edit Execute
Fibonacci.asm
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
14 loop: 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
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
$t8 24 0x00000000
$t9 25 0x00000000
$k0 26 0x00000000
$k1 27 0x00000000
$gp 28 0x10008000
$sp 29 0x7fffeffc
$fp 30 0x00000000

MIPS汇编器：语法

- 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；
- 标号放在行首，后跟冒号（:），例如

```
        .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汇编语言程序设计：

MIPS汇编语言语句格式

- 指令与伪指令语句

  [Label:] <op> [arg1], [arg2], [arg3] [#comment]

- 汇编命令(directive)语句

  [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)
                         # [address]为可选的地址
.align  n                # 以 2^n字节边界对齐数据
                         # 只能用于数据段
```

MIPS汇编语言程序设计

汇编命令

.ascii <string> # 在内存中存放字符串

.asciiz <string> # 在内存中存放NULL结束的字符串

.word $w_1, w_2, \ldots, w_n$ # 在内存中存放n个字

.half $h_1, h_2, \ldots, h_n$ # 在内存中存放n个半字

.byte $b_1, b_2, \ldots, b_n$ # 在内存中存放n个字节

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

n! 程序

```
    .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
        addi $s1,$s1,-1
        bnez $s1,loop    #f=n!
        j   result

        .data 0x10000200
n:      .word 4
f:      .word 0
```

版本（2）

```
        .data 0x10000000        #下列语句行是数据代码行
        .word 4,0               #定义了两个字型立即数4和0
        .text                   #下列语句行是指令代码行
main:
        lui     $s6,$0,0x1000   #  获得数据起始地址
                                #  $s6=0x10000000
        addiu  $s5,$s6,0x0004   #  $s5=0x10000004
fact:   addiu  $s0,$0,1         #  循环计数器赋初值
        lw      $s1,0($s6)      #  把 word型数 4 载入 $s1
loop:   mul    $s0,$s1,$s0      #   $s0=n!, n=4
        addi   $s1,$s1,-1       #   $s1-1
        bnez   $s1,loop         #
        sw     $s0,0($s5)       #   f=n!=24
        jr $ra                  #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）
  - 因为我们通常有很多的变量要存，不止32个
- 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 and SW 只在一块代码开始和结束时使用
  - 节省指令
  - also, 事实上LW and SW 比寄存器操作要慢得多得多！
- 由于一条指令只能采用两个输入,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
main:   addu  $s3,$ra,$0
        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
        addi    $s1,$s1,-1
        bnez    $s1,loop
        sw      $s0,0($s5)      #s0 hold result, return result in s0
        addu  $ra,$s3,$0
        jr      $ra
```

在程序起始处保存ra是一种习惯，目的是避免在程序中有jal指令修改了ra，我们跳不回去了，本程序中没有用ra，可删除以节省寄存器和指令数。

lui $s6, 0x1000

编程指南：（2）分支

- 在符号汇编语句中,分支语句的目标位置是用绝对地址方式写的
  - e.g., beq $0,$0,fact
    means PC ← 0x00400100
- 不过在实现中,要用相对于PC的地址来定义
  - e.g., beq $0,$0,0x3f
    means PC ← 0x00400100

```
        .text
main:   addu  $s3,$ra,$0
        lui     $s6,0x1000
        addiu $s4,$s6,0x0200
        addiu $s5,$s6,0x0204
        beq     $0,$0, fact
result: sw      $s0,0($s5)
        addu  $ra,$s3,$0
        jr      $ra
        .text 0x00400100
fact:   sw      $ra,0($s7)
        addiu $s0,$0,1
        lw      $s1,0($s4)    #$s0=n!
loop:   mul     $s0,$s1,$s0
        addi    $s1,$s1,-1
        bnez    $s1,loop      #f=n!
        j       result
        .data 0x10000200
n:      .word 4
f:      .word 0
```

分支语句中的偏移量的使用

- 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
  - e.g., beq $0,$0,fact如果位于地址0x00400000 的话,
    word displacement=(target - (<PC>+4)) / 4
    =(0x00400100-0x00400004)/4
    =0xfc/4=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?
  - *relocatable* 代码(可重新定位的)
  - 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

- 分支
  - 如果和 0比较, 则直接使用**blez,bgez,bltz,bgtz, bnez**

    e.g., loop example before
  - 更复杂的比较, 采用比较指令（如**slt**）, 然后再用与**0**比较
- **Example:**

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

编程指南:分支 test2

```
        .data 0x10000000
        .word -6,0                  #x：-6，y：0
        .text
main:
        lui    $s6,0x1000           #计算内存中数据存地址$s6=x，
        addiu $s5,$s6,4             #$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                #$s2=1, 跳到done
        j 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) n=17
- 使用LW和SW的$S5和$S6

```
LW   $S4, const($S5)
SW   $S4, const($S6)
```

  - –将 *const* 作为数组偏移量
  - –将寄存器$S5和$S6作为数组中的开始地址(A[0]， B[0])

#编程指南:（3）数组array test3

```
        .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
        lui     $s5,0x1000          #$s5=A[ ]=0x10000000
        addiu $s6,$s5,0x0400  #$s6=B[ ]=0x10000400
        addiu $s0,$0,0x11       #Size(A)=Size(B)=0x11
loop:   addi   $s0,$s0,-1          # 计数
        addiu $s1,$0,4
        mul    $s2,$s1,$s0        # 换算地址
        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

- **SLL by k 等价于 MUL by 2^k**
- **SRL by k 等价于 DIV by 2^k**

只对无符号数成立

- **很有用，因为 MUL 和DIV一般都比SLL 和SRL慢**
  - *想想怎么实现MUL和DIV*
- **对于有符号数用 SRA**
  - **高位用符号位填充(在2的补码表示情况下)**
  - **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: addi  $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)  向下取整
      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

Side Note #1: 用移位代替乘法

```
    add $s0,$0,$a2       # i = N
loop:
    addi   $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的补码表示情况下)

编程指南:（4）过程调用

- 用汇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！

- 大多数有关过程调用的记录集中在一个叫做过程调用帧的内存块中，实现：
  - 以参数形式保存调用值
  - 保存主调过程的运行现场（寄存器）
  - 为程序的变量提供足够的内存空间（局部变量、临时变量）
- 程序的调用和返回严格遵从后进先出（**LIFO**）原则

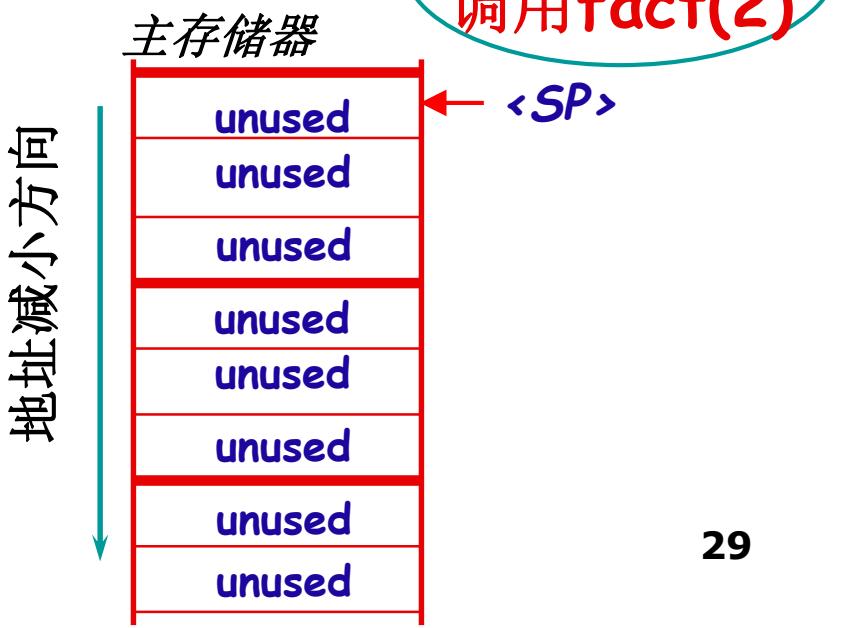
编程指南:（4）过程调用

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *n* in *fact(n)*
  - 临时/局部的变量 （在*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）
  - 缩减栈(POP 出来) （每次返回时）
  - 每次调用都有自己的“栈框架”

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

例如
调用fact(2)

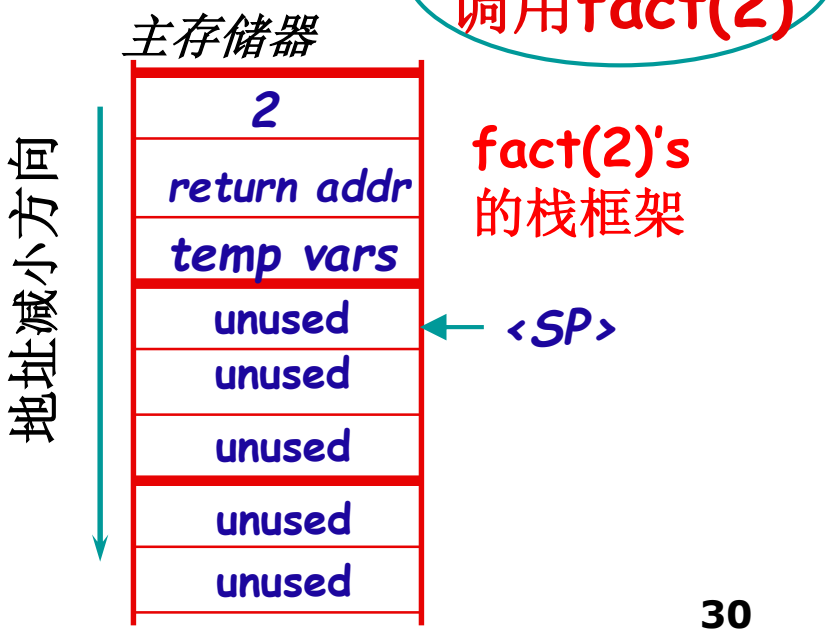


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *n* in *fact(n)*
  - 临时/局部的变量 （在*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）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的“栈框架”

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

例如
调用fact(2)

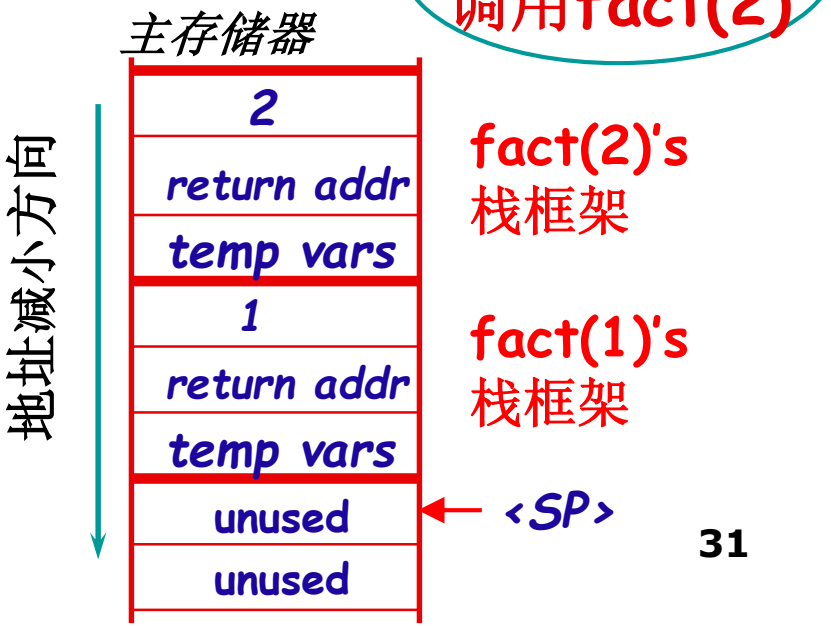


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *n* in *fact(n)*
  - 临时/局部的变量 （在*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）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的“栈框架”

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

例如
调用fact(2)

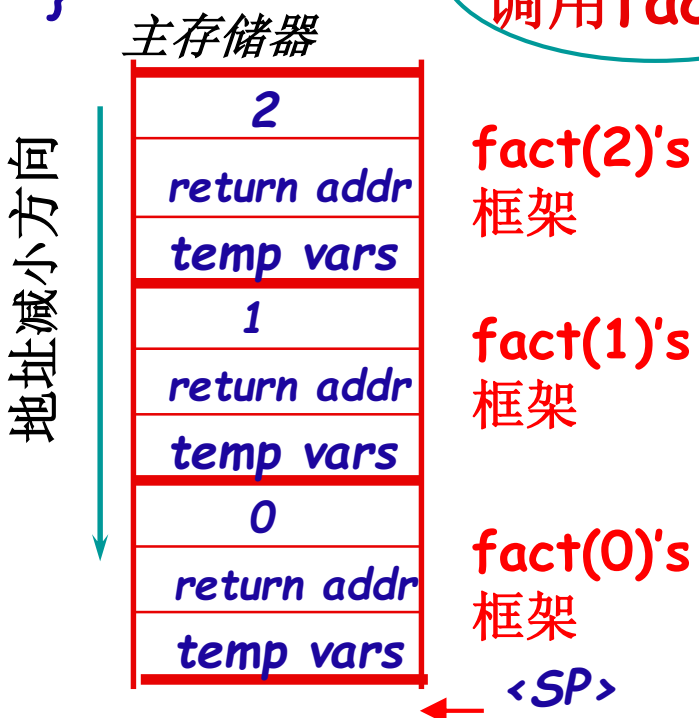


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *n* in *fact(n)*
  - 临时/局部的变量 （在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）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的“栈框架”

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

例如
调用fact(2)

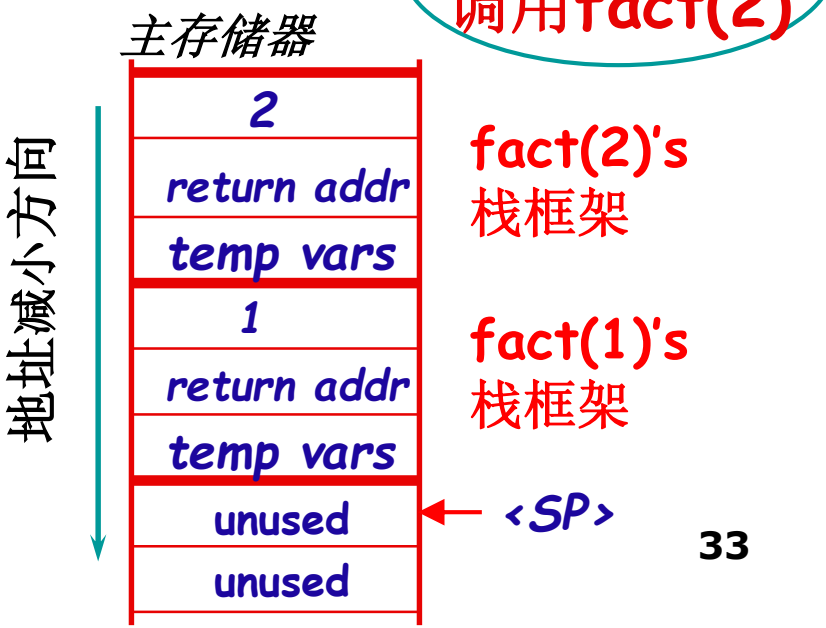


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *n* in *fact(n)*
  - 临时/局部的变量 （在*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）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的“栈框架”

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

例如
调用fact(2)

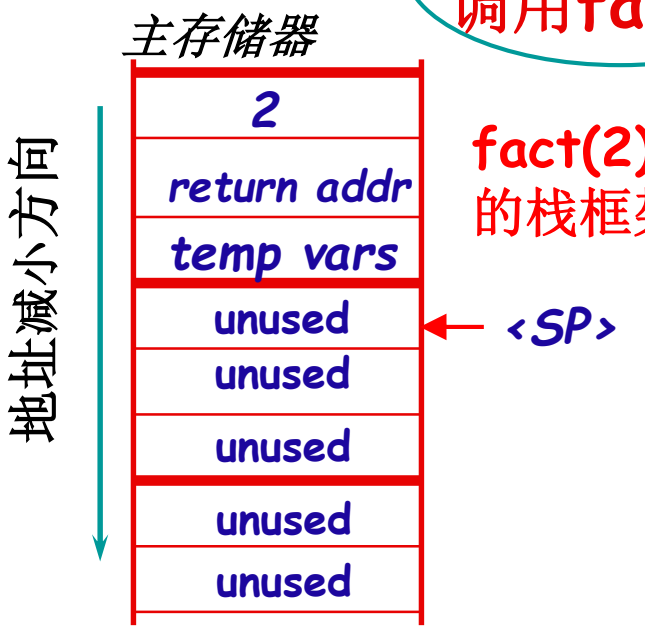


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *n* in *fact(n)*
  - 临时/局部的变量 （在*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）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的“栈框架”

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

例如
调用fact(2)

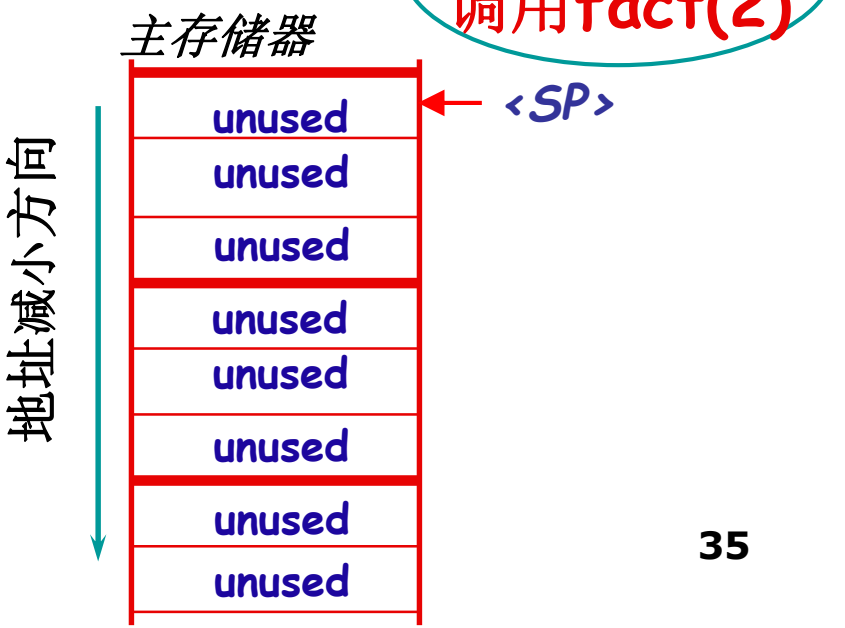


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *n* in *fact(n)*
  - 临时/局部的变量 （*在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）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的“栈框架”

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

例如
调用fact(2)



MIPS计算机硬件对过程的支持

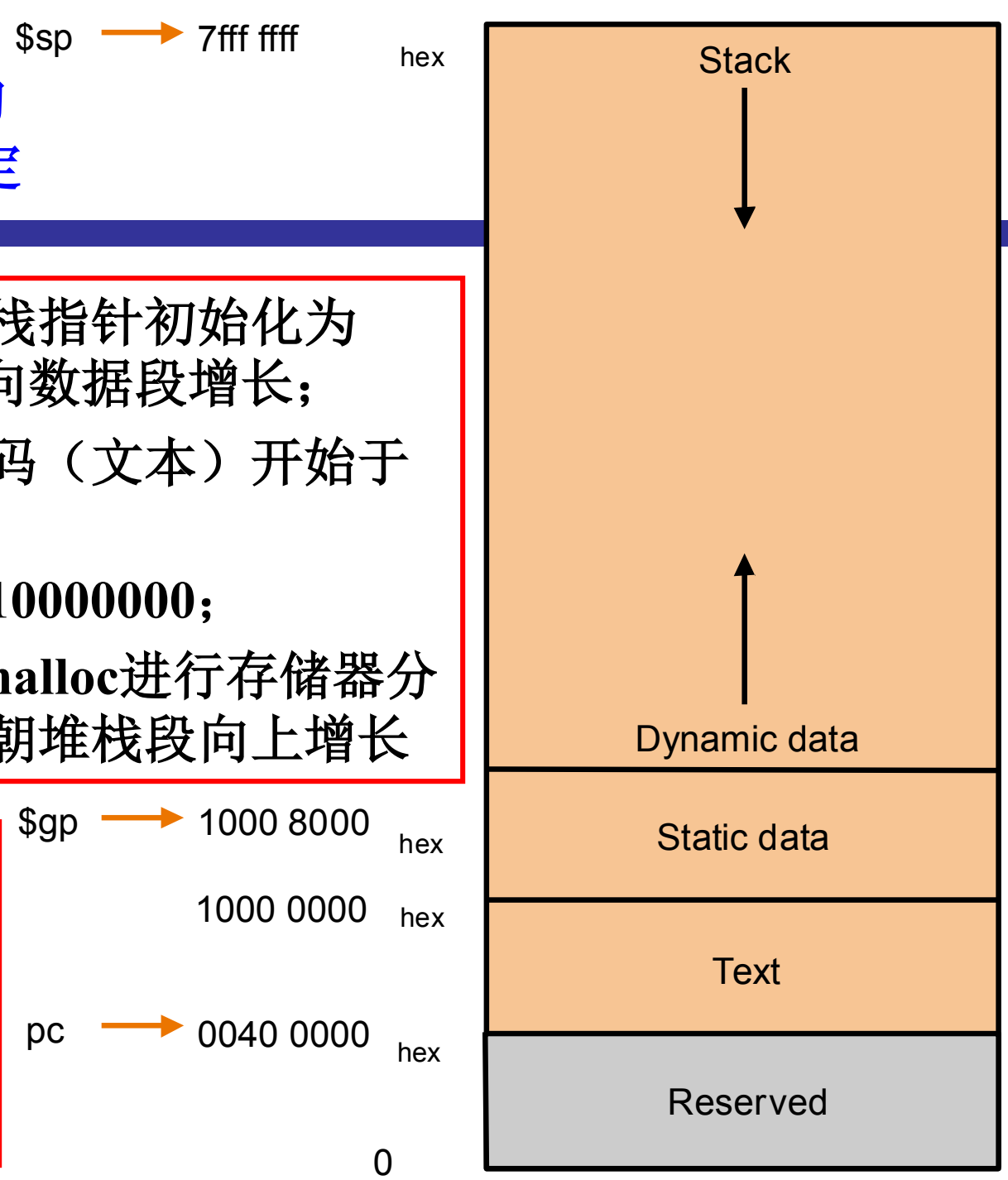
- 为新数据分配空间
  - 利用堆栈存储过程中不适合用寄存器保存的局部变量（如局部数组、或结构）
  - 过程框架：也叫活动记录，是指包含了过程保存的寄存器和局部变量的堆栈段。
  - 通过下例了解过程调用之前、之中和之后的堆栈状态

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

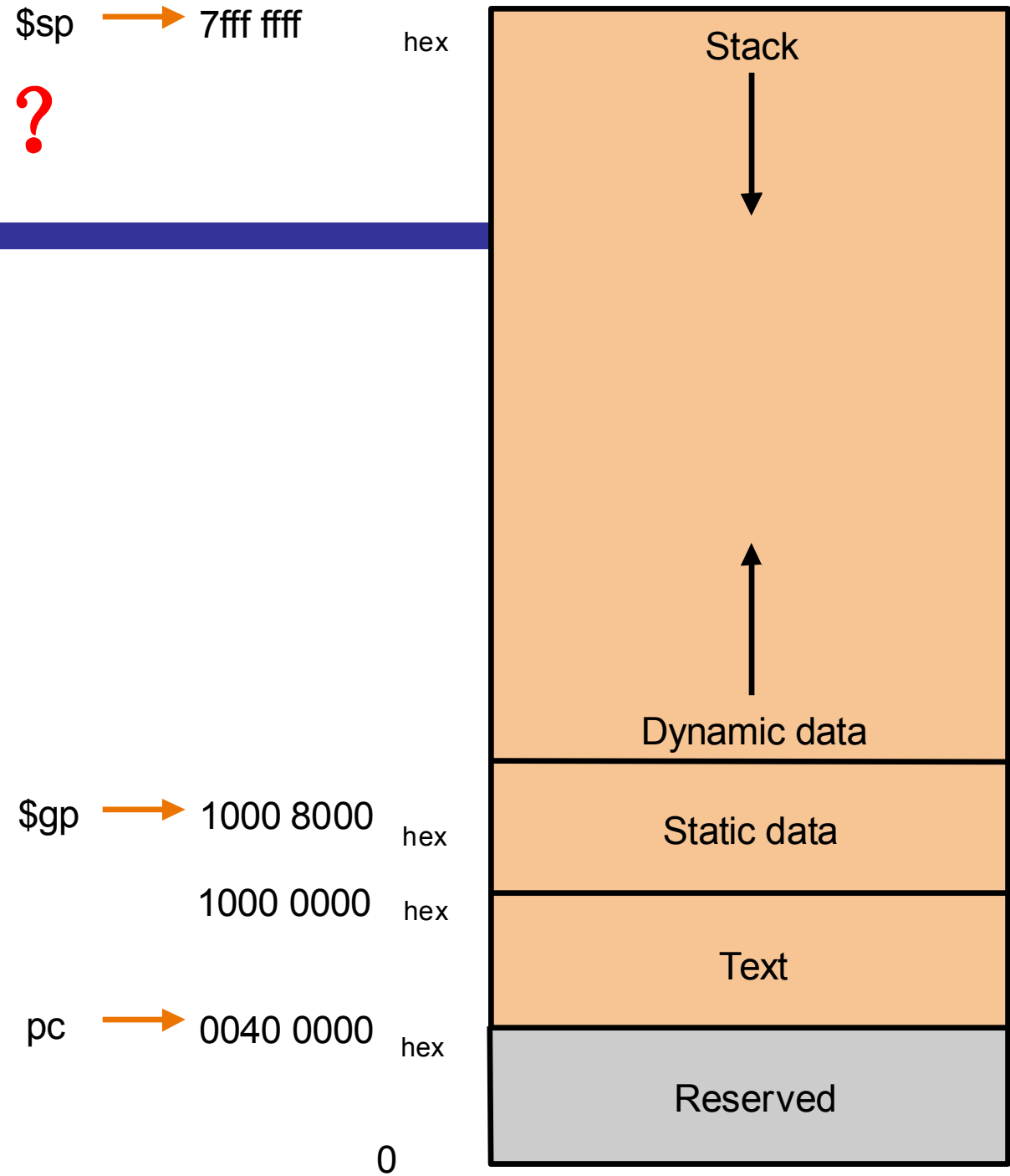
全局指针被设定为易于访问数据的地址，以便使用相对于$gp的±16位偏移量

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



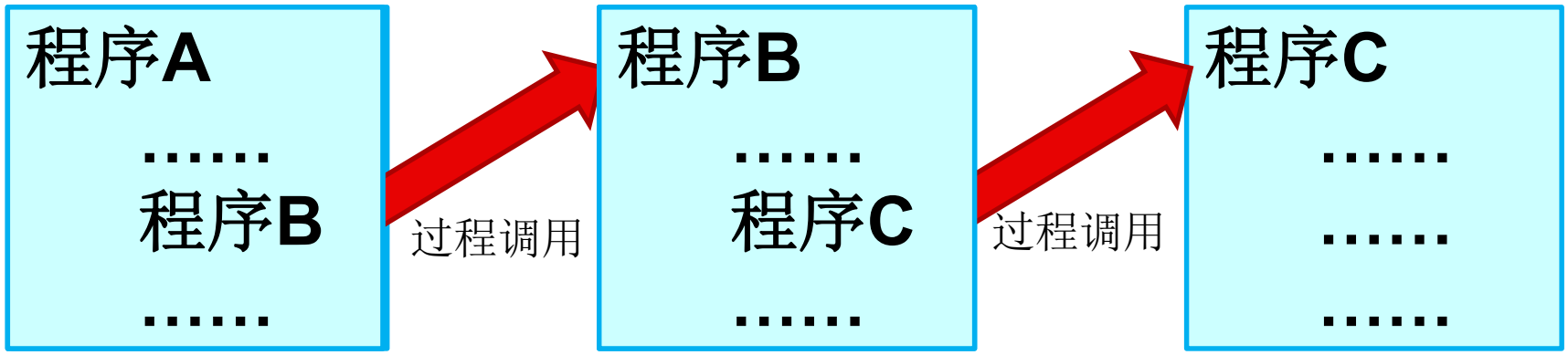
数据存在哪？

```
        .data 0x10000000
        .word 4,0
        .text
main:   addu $s7,$ra,$0
        lui $s6,0x1000
        addiu $s5,$s6,4
fact:   addiu $s0,$0,1
        lw $s1,0($s6)
loop:   mul $s0,$s1,$s0
        subu $s1,$s1,1
        bnez $s1,loop
        sw $s0,0($s5)
        addu $ra,$0,$s7
        jr $ra
```



过程调用

- **MIPS的过程调用遵循如下约定：**
  - 通过$a0～$a3四个参数寄存器传递参数
  - 通过$v0～$v1两个返回值寄存器传递返回值
  - 通过$ra寄存器保存返回地址



过程调用

- 子程序调用通过跳转与链接指令 **jal** 进行

```
jal Procedure  # 将返回地址保存在$ra寄存器中，
               # 程序跳到过程 Procedure处执行
```

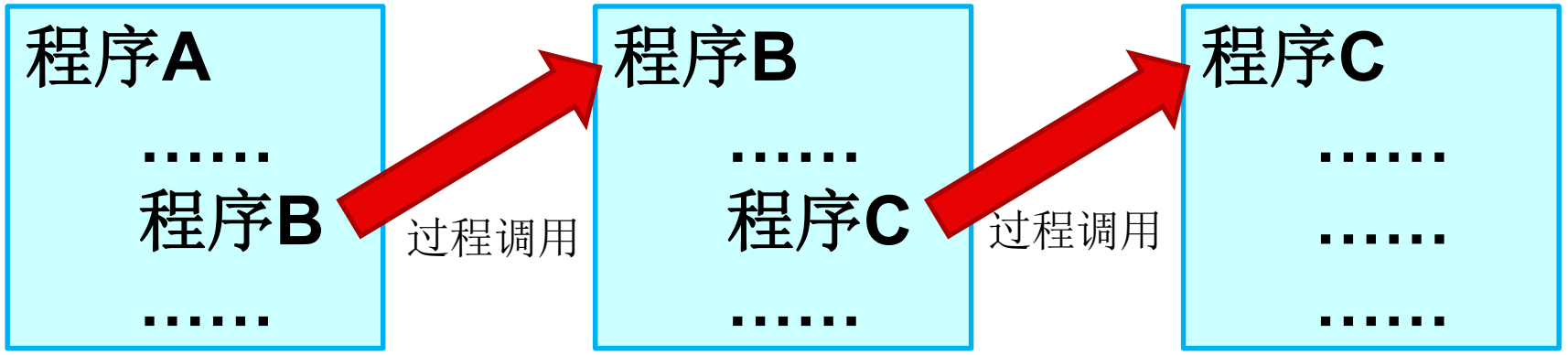
- 子程序返回通过寄存器跳转指令 **jr** 进行

```
jr $ra         # 跳转到寄存器指定的地址
```

jal 和 j 的区别：jal将跳转的地址写入$ra

叶过程

不调用其他过程的过程



叶过程

叶过程

不调用其他过程的过程

```
int  leaf_example (int g,
int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

问题：g+h，i+j这种中间变量需要保存吗？

```
leaf_example:
    addi    $sp,  $sp,  -12 //开辟栈空间
    sw      $t1, 8($sp)//将原来$t1的值进行保存，下同
    sw      $t0, 4($sp)
    sw      $s0, 0($sp)
    add     $t0, $a0, $a1//计算过程
    add     $t1, $a2, $a3
    sub     $s0, $t0, $t1
    add     $v0, $s0, $zero //返回值
    lw      $s0, 0($sp)//将原来$t1的值进行恢复，下同
    lw      $t0, 4($sp)
    lw      $t1, 8($sp)
    addi    $sp, $sp, 12 //释放栈空间
    jr      $ra //跳回上层程序
```

叶过程

```
int  leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

临时寄存器不必保存
保存寄存器必须保存

```
leaf_example:
  addi    $sp,  $sp,  -4
  sw      $t1, 8($sp)
  sw      $t0, 4($sp)
  sw      $s0, 0($sp)
  add     $t0, $a0, $a1
  add     $t1, $a2, $a3
  sub     $s0, $t0, $t1
  add     $v0, $s0, $zero
  lw      $s0, 0($sp)
  lw      $t0, 4($sp)
  lw      $t1, 8($sp)
  addi    $sp, $sp, 4
  jr      $ra
```

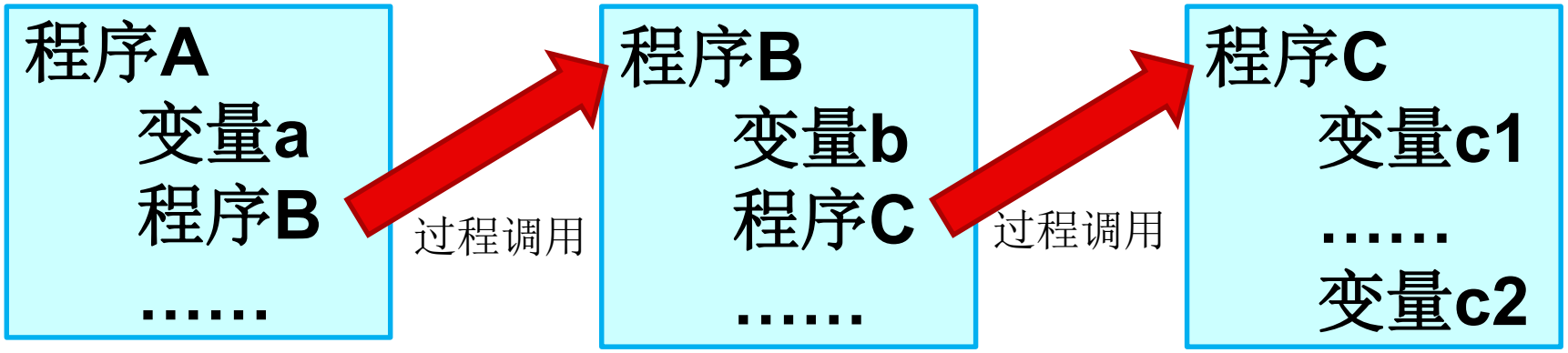
嵌套过程调用

- 主调过程将调用后还需要使用的参数寄存器$a0～$a3和临时寄存器$t0～$t9压栈

- 被调过程将返回地址寄存器ra和在被调过程中修改了的保存寄存器$s0～$s7压栈

嵌套过程调用

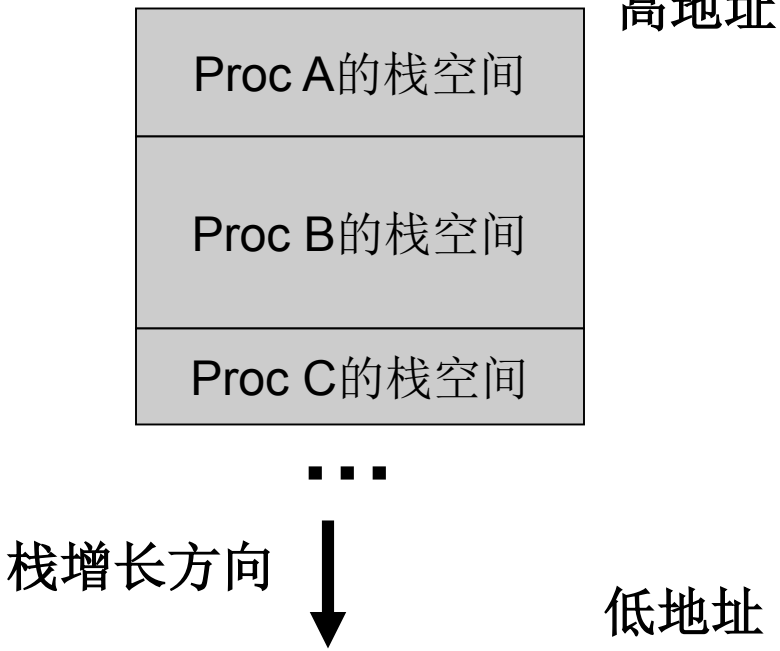
- 主调过程将调用后还需要使用的参数寄存器$a0～$a3和临时寄存器$t0～$t9压栈
- 被调过程将返回地址寄存器$ra和在被调过程中修改了的保存寄存器$s0～$s7压栈



问题：这些变量都是怎么存的？？？

嵌套过程调用

上层程序调用下层程序时，将自己的变量压栈保存



Proc A

调用 Proc B

调用 Proc C

从Proc C返回

从Proc B返回

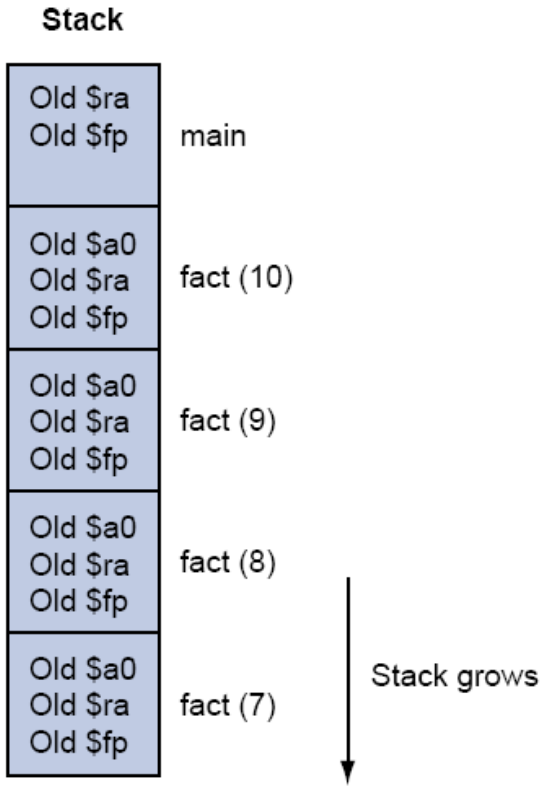
从Proc A返回

嵌套过程调用（ref.2/3.6）

```
fact:
        addi    $sp, $sp, -8
        sw      $ra, 4($sp)
        sw      $a0, 0($sp)
        slti    $t0, $a0, 1
        beq     $t0, $zero, L1
        addi    $v0, $zero, 1
        addi    $sp, $sp, 8
        jr      $ra
L1:
        addi    $a0, $a0, -1
        jal     fact
        lw      $a0, 0($sp)
        lw      $ra, 4($sp)
        addi    $sp, $sp, 8
        mul     $v0, $a0, $v0
        jr      $ra
```

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

- 计算**n!**



```
fact:
        addi    $sp, $sp, -8        # 调整栈指针
        sw      $ra, 4($sp)         # 保存返回地址
        sw      $a0, 0($sp)         # 保存参数n
        slti    $t0, $a0, 1         # 判断n是否小于1
        beq     $t0, $zero, L1      # n≥1则调至L1
        addi    $v0, $zero, 1       # 否则返回1
        addi    $sp, $sp, 8         # 从栈中弹出两个值
        jr      $ra                 # 返回值jal后
L1:
        addi    $a0, $a0, -1        # n≥1，参数变成（n-1）
        jal     fact                # 用（n-1）调用fact
        lw      $a0, 0($sp)         # 从jal返回，恢复参数n
        lw      $ra, 4($sp)         # 恢复返回地址
        addi    $sp, $sp, 8         # 调整堆栈指针以弹出两个栈元素
        mul     $v0, $a0, $v0       # 返回n*fact(n-1)
        jr      $ra                 # 返回到调用者
```

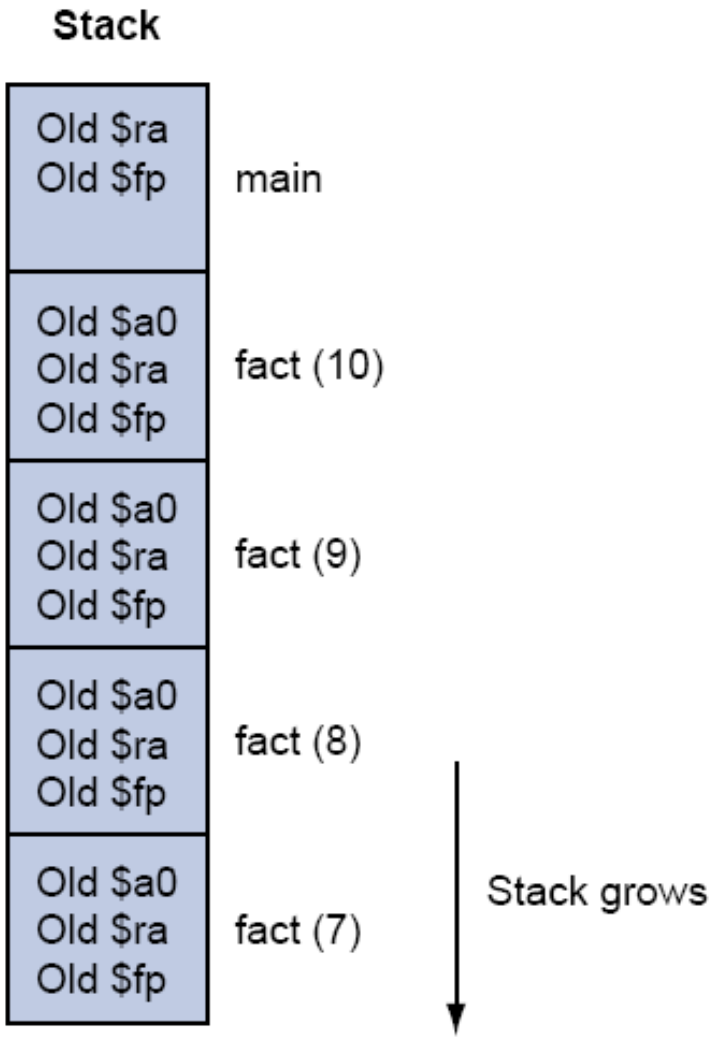
过程调用

- 过程调用保持情况（从**caller**的视角）

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp	Argument registers: $a0–$a3
Return address register: $ra	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer

为新数据分配空间

- 框架指针**$fp**指向框架的第一个字，通常是保存的参数寄存器；
- 栈指针**$sp**指向栈顶，在程序执行的过程中栈指针有可能改变；
- 因此通过固定的框架指针来访问变量要比用栈指针更简便。
- 如果一个过程的栈中没有局部变量，编译器将不设置和恢复框架指针，以节省时间。
- 当需要框架指针时，以调用时的**$sp**值作为框架指针的初值，调用返回时，根据**$fp**恢复**$sp**值



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

阅读汇编代码

```
        .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?

阅读汇编代码

```
 .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)

优化代码

```
      .data
#x:    $a0
#f:    $a1
      .text
       move $s1,$0        #$s1=0
       move $s2,$s1       #$s2=0
       addiu $s3,$0,1     #$s3=1
       lw     $s4,0($a0)  #$s4=x
loop:
       addi   $s4,$s4,-1  #计数器$s4减1
       beqz  $s4,done     #if($s4=0)done
       move $s1,$s2       #$s1=0
       move $s2,$s3       #$s2=1
       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：
  = const + 6x
- 能否达到
  const + 5x?

Optimizing Code

```
        move $s1,$0        #$s1=0
        move $s2,$s1       #$s2=$s1
        addiu $s3,$0,1     #$s3=1
        lw     $s4,0($a0) #$s4=x
        addi   $s4,$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
        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 addi和bnez移到后面，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

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

MIPS程序设计（一）系统调用

- **Hello world**

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

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

MIPS程序设计（一）系统调用

系统调用

- **MIPS模拟器通过系统调用指令(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程序设计（一）系统调用

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程序设计（一）系统调用

系统调用

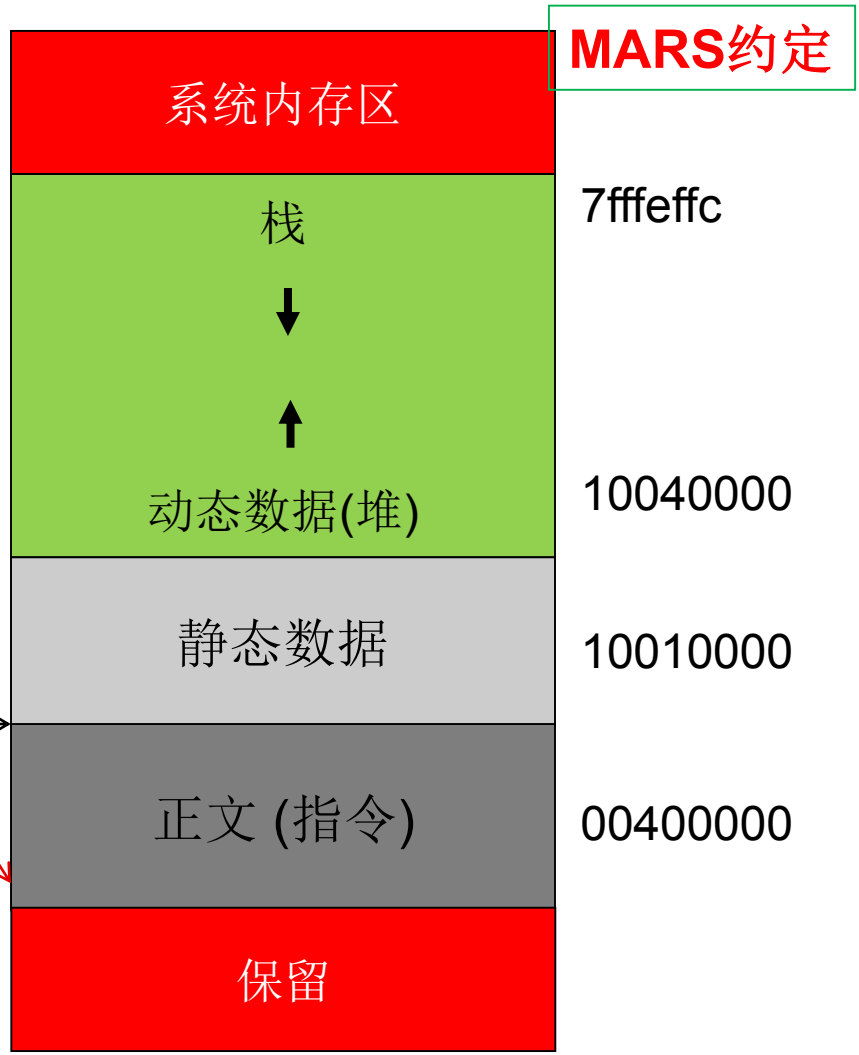
代码	系统调用	参数	结果
11	print char	$a0	
12	read char		char in $v0
13	open	$a0=file name(string), $a1=flags, $a2=mode	file descriptor (fd) in $v0
14	read	$a0 =fd, $a1=buffer, $a2=length	num chars read in $v0
15	write	$a0 =fd, $a1=buffer, $a2=length	num chars write in $v0
16	close	$a0 =fd	
17	exit2	$a0=result	

MIPS程序设计（一）系统调用

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

        li      $v0, 10
        syscall

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



MIPS程序设计（一）系统调用

- 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       # print string
        la  $a0, str1
        syscall

        li   $v0, 5       # read integer
        syscall

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

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

        li    $v0, 1     # print integer
        add  $a0, $t1, $t0
        syscall
```

MIPS程序设计（一）系统调用

- 计算$1^2+2^2+\ldots+100^2$

```
        .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    # print string
        syscall
        li      $v0, 1    # print integer
        move    $a0, $t8
        syscall
        li      $v0, 10   # exit
        syscall

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

MIPS程序设计（二）过程调用

- 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);
        }
    }
}
```

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

swap过程

- 寄存器分配：$a0——参数v[j-1]的地址，
  $a1——参数k，即j-1，
  $t0 ——变量temp，即v[j]的值
- 因为不使用$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

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

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：$a0——参数v[ ]的地址，$a1——参数n，
  i——$s0，j——$s1
- 外循环框架：

```
                move   $s0, $zero        # 循环初始化
loopbody1:      bge    $s0, $a1, exit1   # n≤0 跳到exit1
                … 内循环循环体 …
                addi   $s0, $s0, 1       # i=i+1
                j      loopbody1
exit1:          addi   $s1, $s0, -1
```

sort过程

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

- 内循环框架：

```
loopbody2:    blt    $s1, $zero, exit2    # j<0, 跳到exit2
              sll    $t1,  $s1, 2         # t1= j×4
              add    $t2, $a0, $t1        # 计算V[t1]地址
              lw     $t3, 0($t2)
              lw     $t4, 4($t2)
              ble    $t3, $t4, exit2      #v[j] <=v[j+1] , 跳到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，例如将其复制到$s2和$s3中

```
move    $s2, $a0        #$s2=v
move    $s3, $a1        #$s3=n
…
move    $a0, $s2        #$s2=v
move    $a1, $s1        #$s1=j
jal     swap
```

寄存器分配：$a0——参数v，$a1——参数n，i——$s0，j——$s1

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
```