函数

实现函数的6个步骤

- 调用者把参数放置在某个地方以便函数能访问
- 调用者转移控制给被调用的函数
- 函数获取局部变量对应的空间
- 函数执行具体功能
- 函数把返回值放置在某个地方，然后恢复使用的资源
- 返回控制给调用者

函数相关寄存器

- $a0-$a3：4个传递参数的寄存器
- $v0-$v1：2个传递返回值的寄存器
- $ra：返回地址寄存器，保存着调用者的地址

编号	名称	用途
0	$zero	常量0
1	$at	汇编器保留
2-3	$v0-$v1	返回值
4-7	$a0-$a3	参数
8-15	$t0-$t7	临时变量
16-23	$s0-$s7	程序变量
24-25	$t8-$t9	临时变量
28		
29		
30		
31	$ra	返回地址

函数调用指令

- jal：Jump and Link (jal)
  - 用途是调用函数

    jal label
  - 把jal的下一条指令的地址保存在$ra，然后跳转到label (即函数地址)
- Jump Register (jr)
  - jr src
  - 无条件跳转到保存在src的地址 (通常就是$ra)
  - jr的用途是从函数返回

PC

- PC（program counter）是一个特殊寄存器，用于保存当前执行指令下一条指令的地址
  - PC是冯式体系结构计算机的关键环节
  - 在MIPS中，PC对于程序员不可见，但可以被jal访问
- 注意：保存在$ra的不是当前指令，而是下一条指令，即PC的值
  - 否则当函数返回的时候，就会再次返回到jal本身了

函数调用示例：叶子函数

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i -40);
  return f;
}
```

  - 参数 g, …, i in $a0, …, $a2
  - f in $s0 (这里用到了 $s0, 所以在使用前要保存 $s0 进栈)
  - 结果用$v0返回

示例的汇编代码

- MIPS code:

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    addi $t1, $a2, -40
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

实现函数的6个步骤

- 调用者把参数放置在某个地方以便函数能访问
  - **$a0~$a3**
- 调用者转移控制给被调用的函数
  - **jal**
- 函数获取局部变量对应的空间？？
- 函数执行具体功能
- 函数把返回值放置在某个地方，然后恢复使用的资源
  - **$v0~$v1**
- 返回控制给调用者jr

保存和恢复寄存器

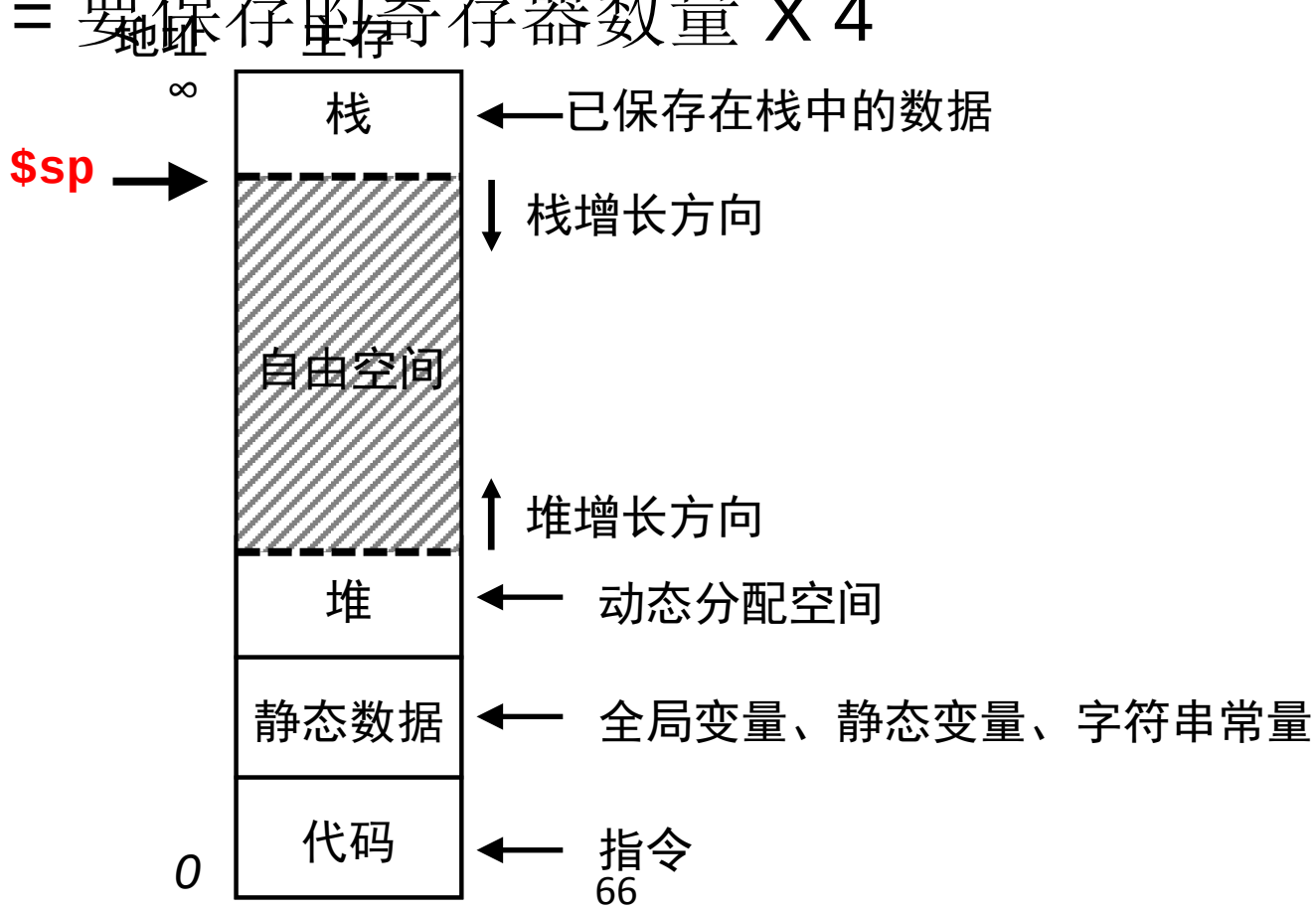
- Q：为什么需要保存寄存器？
  - 理由1：寄存器数量太少，不可能只用寄存器就能编写实用程序
  - 理由2：如果被调用函数继续调用函数会发生什么？

    ($ra 被覆盖了！)
- Q：寄存器保存在什么地方？
  - 栈！一段特殊的内存区域 FILO/LIFO
- $sp：栈指针寄存器 指针指向栈底
  $fp：桢指针寄存器 配合$sp

编号	名称	用途
0	$zero	常量0
1	$at	汇编器保留
2-3	$v0~$v1	返回值
4-7	$a0~$a4	参数
8-15	$t0-$t7	临时变量
16-23	$s0-$s7	程序变量
24-25	$t8-$t9	临时变量
28		
29	$sp	栈指针
30	$fp	桢指针
31	$ra	返回地址

主存分配的基本方案

- 栈帧stack frame：函数为获得保存寄存器而将$sp向0方向调整的空间
  - 栈帧容量 = 要保存的寄存器数量 X 4



非叶子函数

- 函数里又调用了其他函数
- 对于嵌套调用，调用者需要保存到栈里的有：
  - 返回地址
  - 调用后需要用到的任何参数或临时寄存器
- 调用结束后从栈里恢复

非叶子函数举例：递归

- C code:

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

  - 参数 n 在 $a0
  - 结果放在 $v0

MIPS code

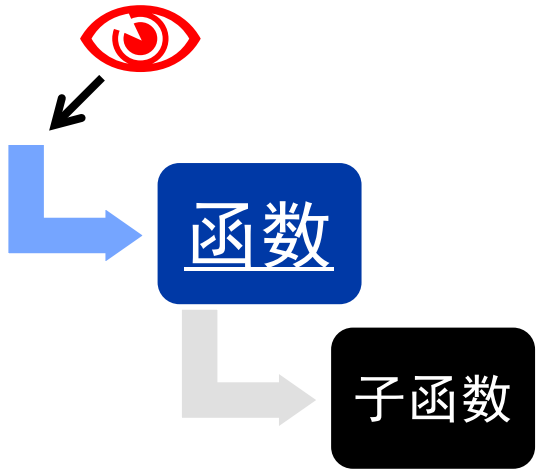
```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       #   pop 2 items from stack
    jr   $ra               #   and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

局部变量和数组

- 由于寄存器只有32个，因此编译器绝大多数情况下不可能把函数需要的所有局部变量都分配在寄存器
- 局部变量存放在函数自己的栈帧中
  - 这样当函数返回时，随着$sp的回调，局部变量自然就被释放了
- 其他局部变量，如数组、结构等，同样也是存储在栈帧中
- 为局部变量分配空间的方法是与保存寄存器是完全相同
  - 将$sp向下调整，产生的空间用于存储局部变量

寄存器的保护分析

- $ra：函数返回地址
  - 保护前提：如果继续调用子函数，那么就必须保护
    - 否则函数就不能返回至父函数
- $s0~$s7：程序员变量
  - 分析：所有函数都要使用程序员变量
    - 程序员变量的生命周期与函数生命周期相同，即进入函数就有效，退出函数后无效
  - 保护前提：用哪些，保护哪些
- 保护动作：刚进入函数时保存；退出函数前恢复，即函数内或者说被调用者保护



寄存器的保护分析

- $t0~$t9：临时变量（MIPS约定其服务于表达式计算）
  - 无需保护：表达式中不调用子函数，故不存在被子函数修改的可能

```
e = a + b + c + d ;  // $t0 = a + b, $t1 = c + d, e=$t0 + $t1
f1() ;
```

  - 需要保护：表达式中调用子函数，故存在被子函数修改的可能

```
z = x + y + f2() ;    // $t0 = x + y
                      // $t0有可能被f2()修改
```

  - 保护前提：如果其值在子函数调用前后必须保持一致
  - 保护动作：调用子函数前保存；在调用子函数后恢复，即函数外保护或者说调用者保护

寄存器的保护分析

序号	名称	MIPS汇编约定的用途	分类	保护的前提条件	何时保护与恢复
16~23	$s0~$s7	程序员变量	强保护	如果需要使用	进入函数时保存 退出函数前恢复
31	$ra	函数返回地址		如果调用子函数	
2~3	$v0~$v1	函数返回值	弱保护	如果要求其值在调用子函数前后必须保持一致	调用子函数前保存 子函数返回后恢复
4~7	$a0~$a3	函数参数			
8~15, 24~25	$t0~$t9	临时变量			

- Saved Register：$s0~$s7，$ra
  - 英文：preserved register；中文：保护寄存器，保存寄存器
- Volatile Register：$t0~$t9，$a0~$a3，$v0~$v1
  - 英文：non-preserved register；中文：非保护寄存器，非保存寄存器

小结：函数调用

- 函数机制
  - 用jal实现函数调用，用jr实现函数返回
  - 用$a0-$a3传递参数，用$v0-$v1传递返回值
  - 寄存器保护
    - 从用途看，寄存器可以分为强保护和弱保护两大类
    - 强保护寄存器：一进函数就保护，退出时恢复
    - 弱保护寄存器：调用子函数时保护，子函数一返回就恢复
  - 栈用于保存寄存器、局部变量等

小测试

- 下列哪个陈述是错误的？
  - MIPS使用jal指令来调用函数，并使用jr指令函数返回
  - jal保存PC+1到$ra
  - 函数如果不担心$t*i*值在调用子函数后发生改变，则无需保存和恢复它们
  - 函数如果要使用($s*i*)，就必须保存和恢复它们

- Thanks ！