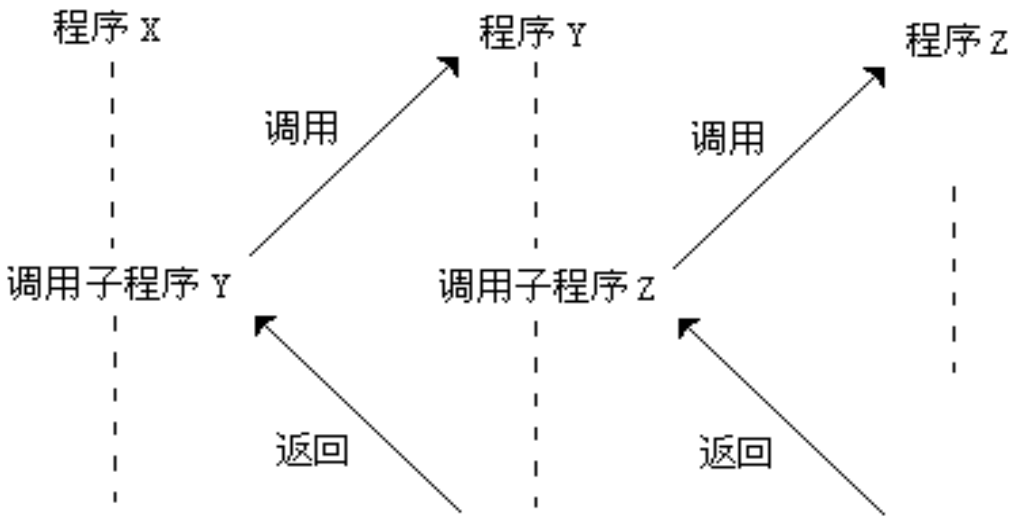
# 第二章 汇编语言

华中科技大学
电子信息与通信学院
School of Electronic Information and Communications

# 学习目标

- 子程序原理
  - –栈
  - –调用、返回过程

# 2.7 子程序实现



- 设计包含子程序的程序时，应解决的问题
  - 主程序与子程序之间的转返
  - 主程序与子程序间的信息传递
  - 主程序和子程序公用寄存器的问题

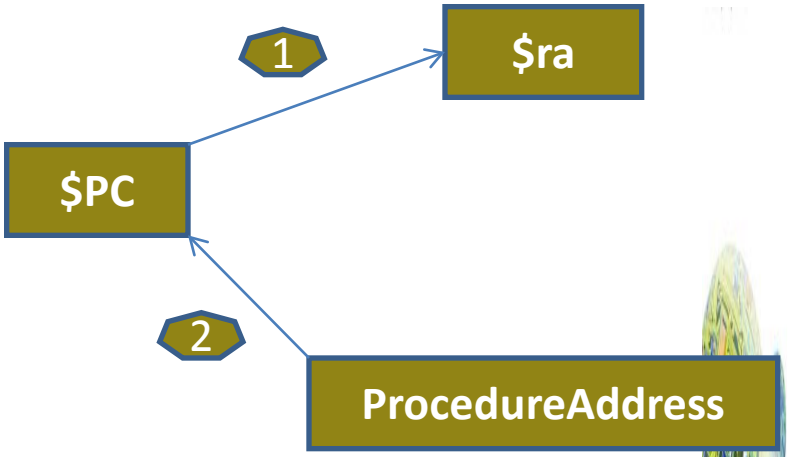
编译器约定：

- $a0~$a3:主程序传递给子程序的入口参数
- $v0,$v1:子程序传递给主程序的返回结果
- $ra:子程序返回到主程序的地址
- $t0~$t9：10个临时寄存器，在子程序调用时，子程序可以不保存其原始值就使用；
- $s0~$s7：8个存储寄存器，在子程序调用时，子程序必须保存其原始值之后才能使用。

# 子程序相关指令

- 子程序的调用

**jal ProcedureAddress**



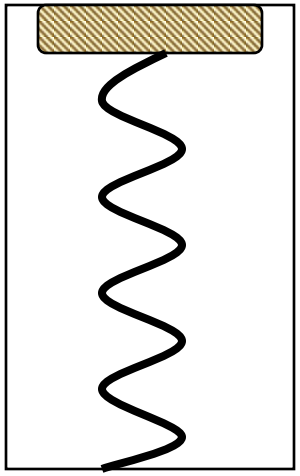

- 返回到主程序

**jr $ra**

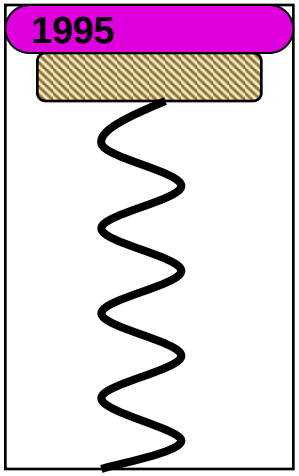
# 栈

- 栈是在存储器中开辟的一片数据存储区，这片存储区的一端固定，另一端活动，且只允许数据从活动端进出
- 栈中数据的存取遵循“先进后出”的原则
- 栈的活动端称为栈顶，固定端称为栈底
- 指示栈顶位置的寄存器，即栈顶地址的指示器$SP
- 栈的伸展方向既可以从高地址向低地址，也可以从低地址向高地址。
  - MIPS处理器以及80x86处理器都是采用从高地址向低地址方向生长
  - 往栈中存数据也叫压入数据（push），$sp的值递减；从栈中读数据也叫弹出数据（pop），$sp的值递增；
  - 不同处理器栈操作支持的数据类型不同，在MIPS处理器中，栈操作数据类型为字类型（32位），因此每次栈操作都将使$sp的值变化4

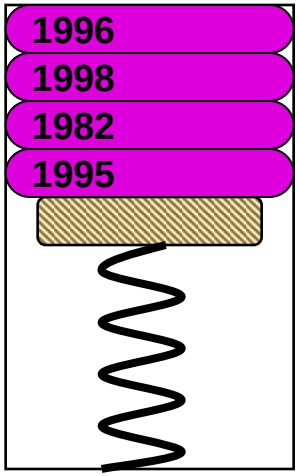
# 硬件栈



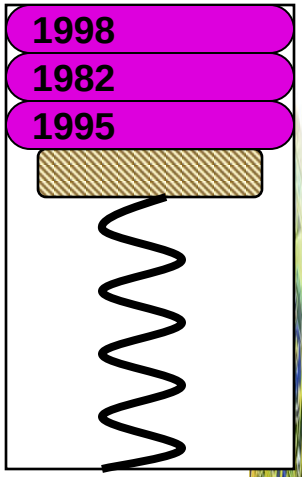
初始状态




压入一个数据



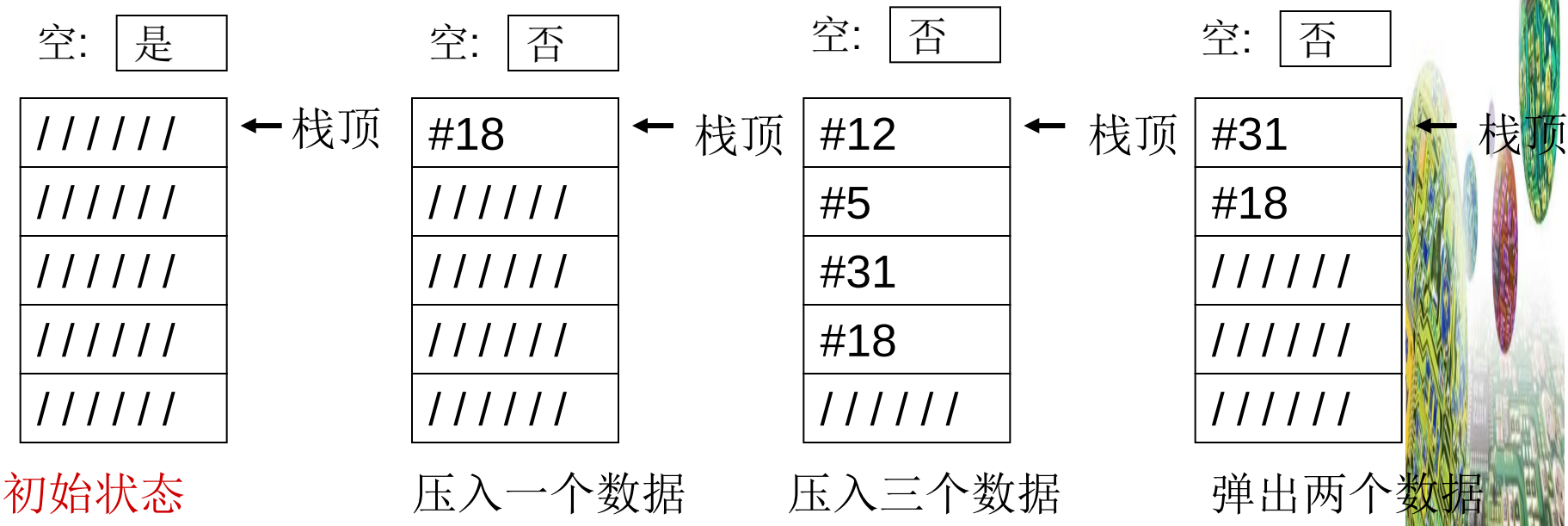

在压入三个数据




弹出一个数据

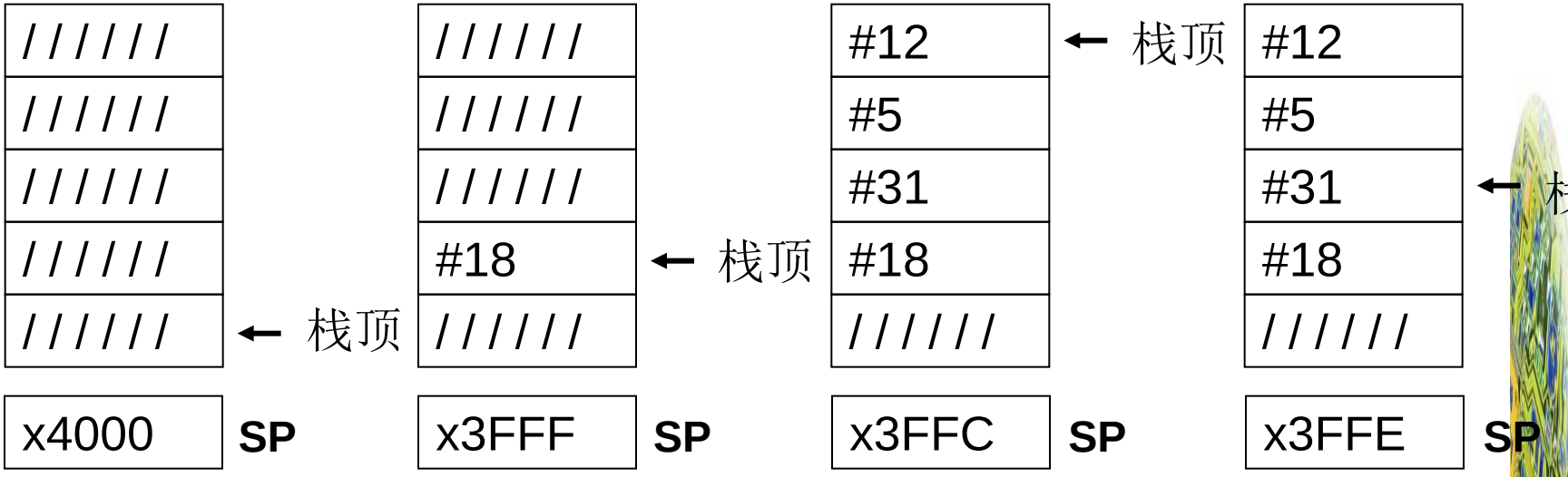
# 硬件栈的实现

- 数据在栈中移动



# 软件实现的栈

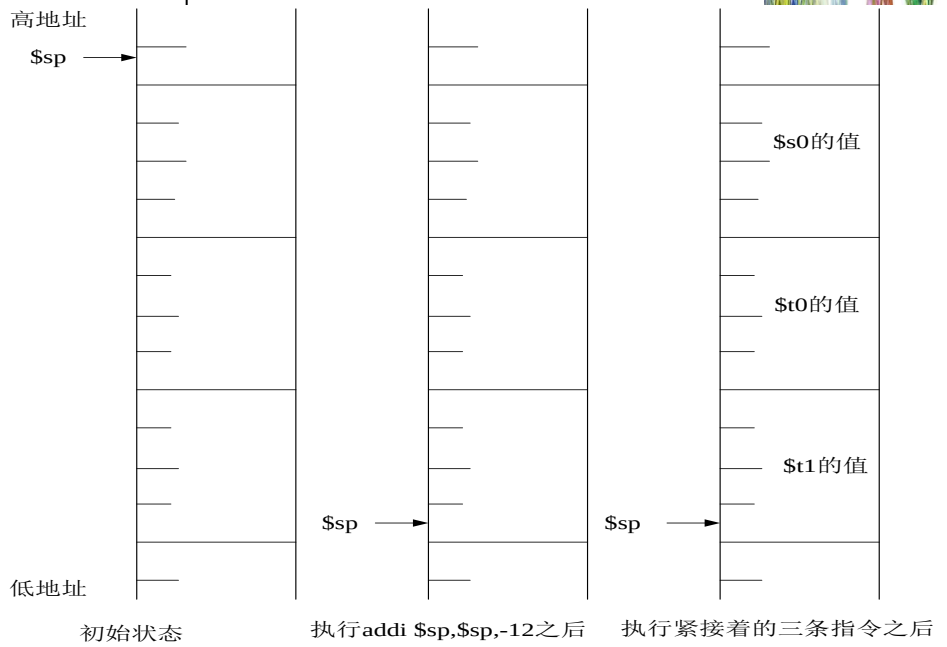
- 数据不在栈中移动，栈顶位置随着数据进出改变.




**SP寄存器 指示栈顶位置 Stack pointer.**

- 主程序和子程序公用寄存器问题的解决方法为：在子程序中事先保存其要使用的寄存器的值之后，再来使用该寄存器，在使用完之后，再恢复该寄存器的原始值。
- 如在子程序中要使用$s0,$t0,$t1这三个寄存器

```
addi $sp,$sp,-12  #修改$sp使其预留3个字的内存空间
sw $s0,8($sp)  #保存$s0的值到栈中
sw $t0,4($sp)  #保存$t0的值到栈中
sw $t1,0($sp)  #保存$t1的值到栈中
```



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

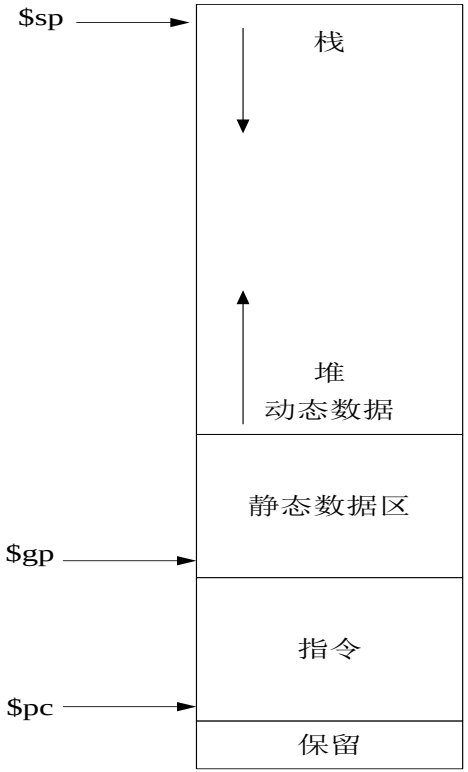
```
leaf_example:                #定义子程序标号
        addi $sp,$sp,-12     #修改$sp预留3个字的内存空间
        sw $s0,8($sp)        #保存$s0的值到栈中
        sw $t0,4($sp)        #保存$t0的值到栈中
        sw $t1,0($sp)        #保存$t1的值到栈中
        add $t0,$a0,$a1      #$t0=g+h
        add $t1,$a2,$a3      #$t1=i+j
        sub $s0,$t0,$t1      #$s0=(g+h)-(i+j),
        add $v0,$s0,$zero    #$v0=$s0+0，
        lw $t1,0($sp)        #恢复$t1的值
        lw $t0,4($sp)        #恢复$t0的值
        lw $s0,8($sp)        #恢复$s0的值，
        addi $sp,$sp,12      #恢复$sp的值
        jr $ra               #返回到主程序处
```

# 子程序的嵌套调用

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

```
fact:
        addi $sp,$sp,-8 #修改栈顶指针，
        sw $ra,4($sp）#保存$ra的值到栈中
        sw $a0,0($sp）#保存$a0的值到栈中
        slti $t0,$a0,1   # $a0是否小于1，小于1则设置$t0为1
        beq $t0,$zero,L1#检测$t0不为0则转移到标号L1处
        addi $v0,$zero,1#返回1到出口参数寄存器$v0中
        addi $sp,$sp,8   #恢复栈指针
        jr $ra          # 返回
L1:     addi $a0,$a0,-1#计算n-1的值
        jal fact# 嵌套调用,结果保存在$v0中
        lw $a0,0($sp）#从栈中获取当前子程序的入口参数
        lw $ra,4($sp）#从栈中获取当前子程序的返回地址
        mult $v0,$a0#计算返回结果实现n*fact(n-1)
        mflo $v0
        addi $sp,$sp,8
        jr $ra#返回
```

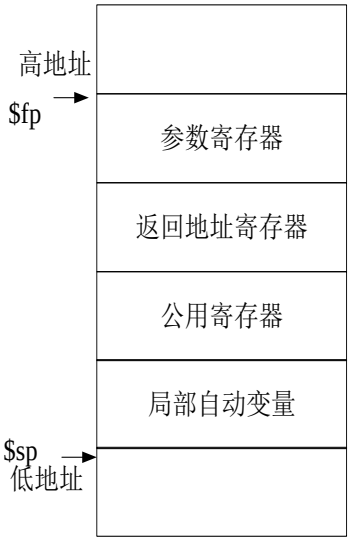
# 程序的内存映像




公用寄存器、
参数寄存器、
返回地址寄存器
自动变量

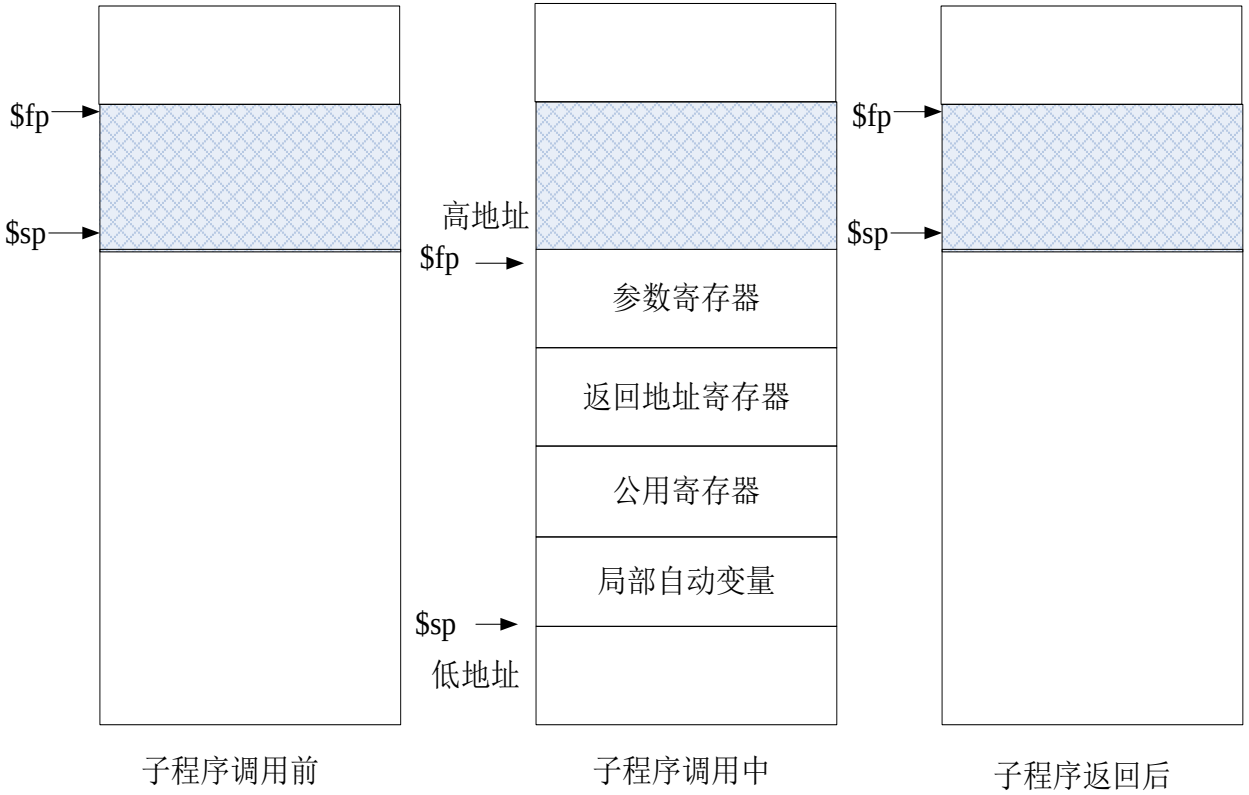
链表

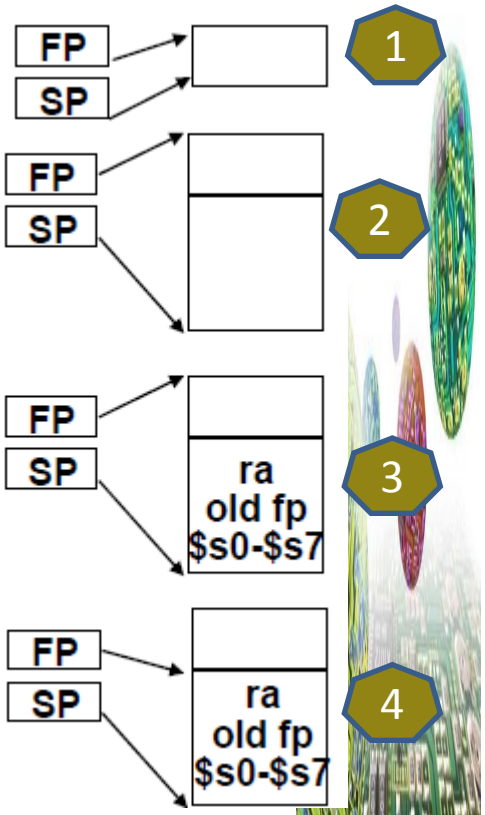
全局变量以及静态变量




子程序被调用时栈的内存映像

# 子程序嵌套调用栈的变化



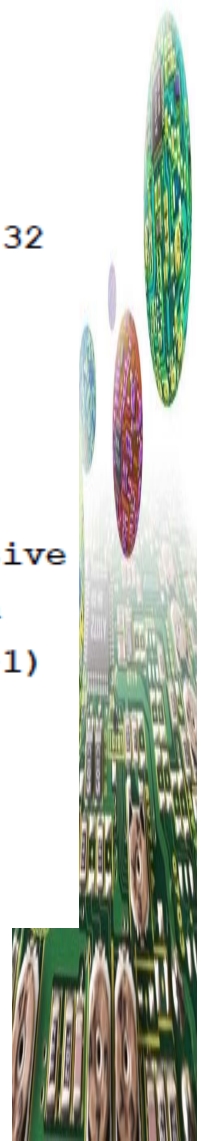
# 子程序嵌套调用和返回规则

- Caller
  - Save caller-saved registers `$a0-$a3`, `$t0-$t9`
  - Load arguments in `$a0-$a3`, rest passed on stack
  - Execute `jal` instruction
- Callee Setup
  1. Allocate memory for new frame (`$sp` = `$sp` - frame)
  2. Save callee-saved registers `$s0-$s7`, `$fp`, `$ra`
  3. Set frame pointer (`$fp` = `$sp` + frame size - 4)
- Callee Return
  - Place return value in `$v0` and `$v1`
  - Restore any callee-saved registers
  - Pop stack (`$sp` = `$sp` + frame size)
  - Return by `jr $ra`

# GCC编译结果

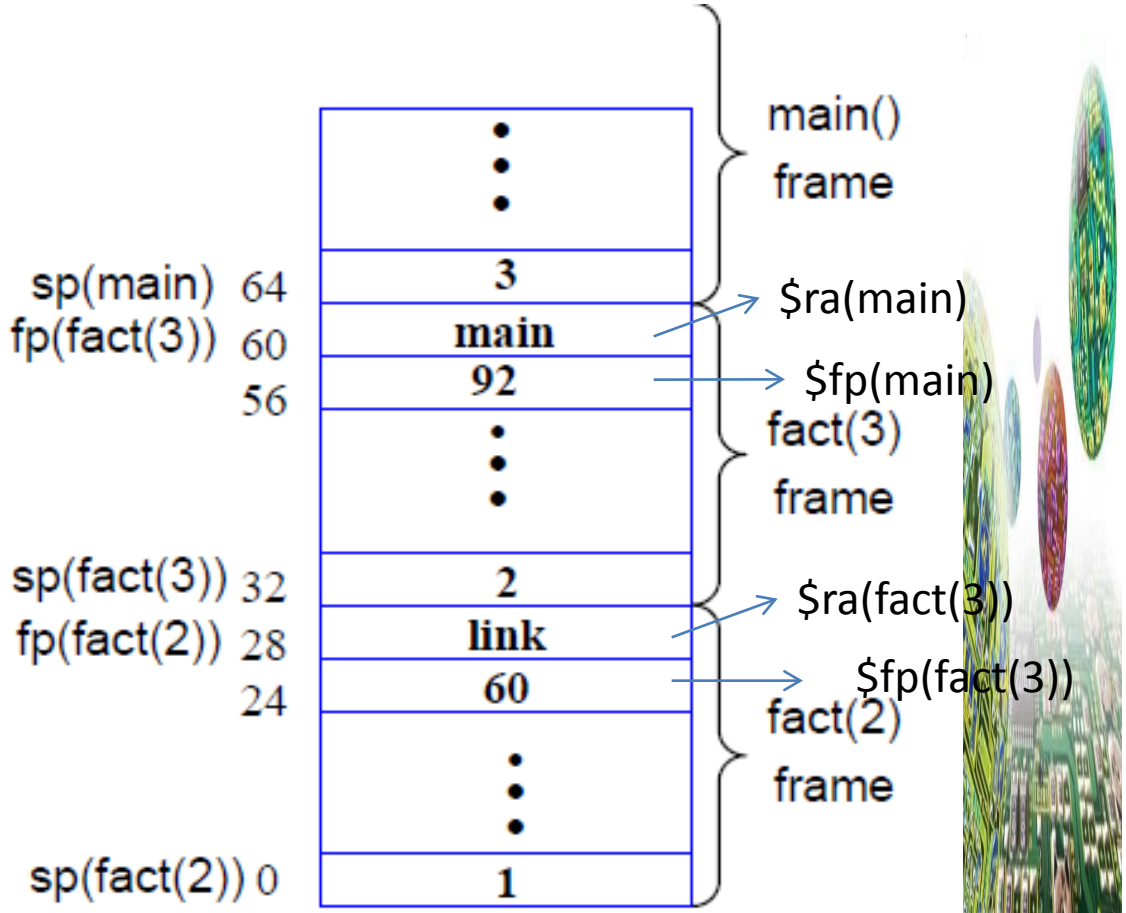
```
int fact(int n)
{
   if (n <= 1)
      return(1);
   else
      return(n*fact(n-1));
}
```

```
Fact: slti    $t0, $a0, 2       # a0 < 2
      beq     $t0, $zero, skip  # goto skip
      ori     $v0, $zero, 1     # Return 1
      jr      $ra               # Return
Skip: addiu   $sp, $sp, -32     # $sp down 32
      sw      $ra, 28($sp)      # Save $ra
      sw      $fp, 24($sp)      # Save $fp
;     addiu   $fp, $sp, 28      # Set up $fp
      sw      $a0, 20($sp)      # Save n
      addui   $a0, $a0, -1      # n - 1
      jal     Fact              # Call recursive
Link: lw      $a0, 32($sp)      # Restore n
      mul     $v0, $v0, $a0     # n * fact(n-1)
      lw      $ra, 28($sp)      # Load $ra
      lw      $fp, 24($sp)      # Load $fp
      addiu   $sp, $sp, 32      # Pop stack
      jr      $ra               # Return
```

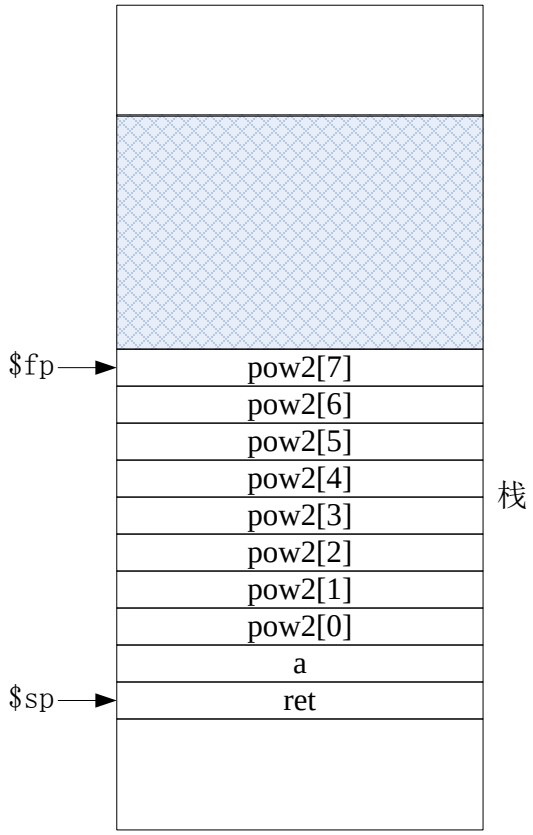


# Fact函数递归调用栈变化示例

```
main()
{
    fact(3);
}
```



# sum_pow2子程序栈内存映像




进入子程序中

**查看MicroBlaze的程序内存映像**

# 2.8 字符数据处理

- 字符串拷贝

ASCII码字符为无符号数据，
采用lbu，sb指令进行存取

```
void strcpy(char X[], char Y[])
{
        int i;
        i=0;
        while((X[
        i+=1;
}
```

```
strcpy:
        add $sp,$sp,-4
        sw $s0,0($sp)
        add $s0,$zero,$zero #$s0为字符索引
L1:     add $t1,$s0,$a1 #将Y字符串中的字符地址保存在$t1中
        lbu $t2,0($t1)    #获取Y字符串中的字符保存到$t2中
        add $t3,$s0,$a0 #将X字符串中的字符地址保存在$t3中
        sb $t2,0($t3)     #将$t2中的字符保存到X中，
        beq $t2,$zero,L2#字符串结束则转移到L2标号处
        addi $s0,$s0,1    #否则修改索引指向到下一个字符
        j L1              #返回到循环处
L2:     lw $s0,0($sp)     #恢复$s0的值
        addi $sp,$sp,4    #恢复$sp的值
        jr $ra            #返回
```

# 作业

- 11 C语言函数原型

```
int fib(int n,int f)
 { if(n==0)
     f=0;
    else if(n==1)
     f=1;
          else
           f=fib(n-1,f)+fib(n-2,f);
 return f;
}
```