

第二章 汇编语言



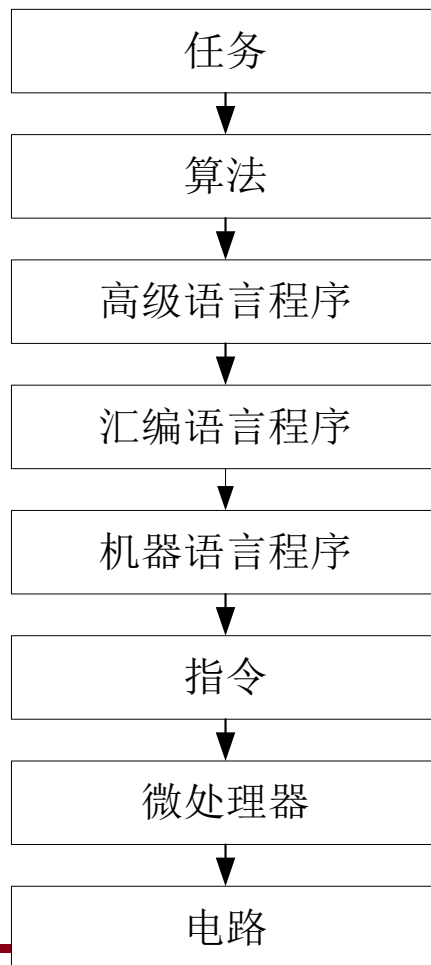
学习目标

- 了解汇编指令的构成以及不同指令集结构的特点
- 了解MIPS指令集不同指令的编码特点
- 熟悉常用MIPS汇编指令的功能和应用，能够利用汇编指令实现常用C语句功能
- 熟悉子程序的实现机理，并且熟练掌握程序运行过程中内存的变化过程
- 熟悉不同寻址方式的原理
- 了解程序的编译、链接、装载过程
- 编写简单MIPS汇编语言程序



2.1 计算机语言

● 控制计算机完成任务的过程





几种计算机语言的特点

● 高级语言：

- 是一类面向问题的程序设计语言，且独立于计算机硬件，对具体算法进行描述，所以又成为“算法语言”。它的特点是独立性、通用性和可移植性好。例如：BASIC，FORTRAN，PASCAL，C，C++等语言都是高级语言。

● 汇编语言：

- 是指使用助记符号和地址符号来表示指令的计算机语言，也称之为“符号语言”。每条指令有明显的标识，易于理解和记忆。

● 机器语言：

- 是最初级的计算机语言，它依赖于硬件，是由1、0组成的二进制编码形式的指令集合。不易被人识别，但可以被计算机直接执行。



几个基本概念

- 汇编：
 - 把汇编语言翻译为机器语言的过程
- 汇编程序：
 - 实现汇编过程的软件程序
- 汇编语言源程序：
 - 用户采用汇编语言编写的程序



2.2 计算机指令

- 计算机指令通常由两个部分构成：操作码和操作数。
 - 操作码指明计算机应该执行什么样的操作，
 - 操作数则指出该操作处理的数据或数据存放的地址。
 - 操作码和操作数在计算机中都采用二进制编码表示。



指令集结构

- 指令集结构中的指令到底要支持哪些类型的操作
 - 复杂指令集计算机（CISC）
 - 强化指令功能，实现软件功能向硬件功能转移
 - 精简指令集计算机（RISC）
 - 尽可能地降低指令集结构的复杂性，以达到简化硬件实现、提高性能的目的



CISC结构指令集特点

指令系统复杂庞大，指令数目一般多达2、3百条。

寻址方式多

指令格式多

指令字长不固定

可访存指令不加限制

各种指令使用频率相差很大

各种指令执行时间相差很大

- 适合于通用机
- Intel公司的X86系列CPU是典型的CISC体系的结构



RISC结构指令集的特点

- 精简了指令系统，流水线以及常用指令均可用硬件执行；
- 采用大量的寄存器，使大部分指令操作都在寄存器之间进行，提高了处理速度；
- 每条指令的功能尽可能简单，并在一个机器周期内完成；
- 所有指令长度均相同；
- 只有Load和Store操作指令才访问存储器，其它指令操作均在寄存器之间进行；
- 适合于专用机
- MIPS R3000、HP-PA8000系列，Motorola M88000

2.3 汇编指令

- 汇编指令是机器指令的符号表示，包括操作数和操作码
- MIPS汇编指令的一般格式为：
 - [标号:] 操作码 操作数1, 操作数2, 操作数3 [#注释]

- 标号代表指令在内存中的存储地址，

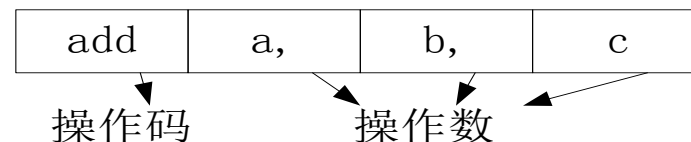
- 操作码表示执行什么操作，

- 操作数表示操作的对象（数据或地址），

- 注释解释指令的功能，为方便读者读懂程序的功能而添加的，汇编程序将忽略这部分内容。

- 加法运算汇编指令：

- add a,b,c





- 不同的微处理器对add，sub等操作码的符号表示不同，
- 不同的指令结构对操作数a,b,c,d,e,t0,t1等的存取方式也不同
- 32位MIPS处理器支持的操作数

操作数名称	举例	备注
寄存器	\$s0~\$s7,\$t0~\$t9,\$zero,\$a0~\$a3 , \$v0~\$v1,\$gp,\$fp,\$sp,\$ra,\$at	MIPS处理器中，算术运算的操作数都必须在寄存器中，寄存器\$zero=0,\$at保留给汇编程序处理大的常数
存储器	Memory[0],memory[4],....	只能用在数据传送指令中，32位数据占据4个字节的内存单元。存储器通常用来存储数据结构体，数组等



32位MIPS处理器常用的汇编指令

类型	指令	指令举例	含义	备注
算术运算	加法	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	三个寄存器操作数
	减法	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	三个寄存器操作数
	加立即数	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	用来加立即数
数据传送	读取字	lw \$s1,20(\$s2)	$\$s1 = \text{mem}[\$s2 + 20]$	从内存读字到寄存器
	存储字	sw \$s1,20(\$s2)	$\text{mem}[\$s2 + 20] = \$s1$	从寄存器写字到内存
	读取半字	lh \$s1,20(\$s2)	$\$s1 = \text{mem}[\$s2 + 20]$	从内存读半字到寄存器
	读取无符号半字	lhu \$s1,20(\$s2)	$\$s1 = \text{mem}[\$s2 + 20]$	从内存读半字到寄存器
	存储半字	sh \$s1,20(\$s2)	$\text{mem}[\$s2 + 20] = \$s1$	从寄存器写半字到内存
	读取字节	lb \$s1,20(\$s2)	$\$s1 = \text{mem}[\$s2 + 20]$	从内存读字节到寄存器
	读取无符号字节	lbu \$s1,20(\$s2)	$\$s1 = \text{mem}[\$s2 + 20]$	从内存读字节到寄存器
	存储字节	sb \$s1,20(\$s2)	$\text{mem}[\$s2 + 20] = \$s1$	从寄存器写字节到内存
	读取链接字	ll \$s1,20(\$s2)	$\$s1 = \text{mem}[\$s2 + 20]$	读字作为原子交换的第一半
	条件存储字	sc \$s1,20(\$s2)	$\text{mem}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ 或 } 1$	写字作为原子交换的第二半
	读取立即数到高半字	lui \$s1,20	$\$s1 = 20 * 2^{16}$	读取一个常数到高16位
	与	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	三个寄存器，位与
逻辑操作	或	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	三个寄存器，位或
	或非	nor \$s1,\$s2,\$s3	$\$s1 = \sim(\$s2 \$s3)$	三个寄存器，位或非
	与立即数	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	寄存器与立即数位与
	或立即数	ori \$s1,\$s2,20	$\$s1 = \$s2 20$	寄存器与立即数位或
	逻辑左移	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	左移常数次
	逻辑右移	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	右移常数次
	相等转移	beq \$s1,\$s2,25	If $(\$s1 = \$s2)$ goto PC+4+25*4	相等测试，转移
条件跳转	不相等转移	bne \$s1,\$s2,25	If $(\$s1 \neq \$s2)$ goto PC+4+25*4	不相等测试，转移
	小于设置	slt \$s1,\$s2,\$s3	If $(\$s2 < \$s3)$ $\$s1 = 1$ else $\$s1 = 0$	比较小于设置 $\$s1 = 1$
	低于设置	sltu \$s1,\$s2,\$s3	If $(\$s2 < \$s3)$ $\$s1 = 1$ else $\$s1 = 0$	比较低于设置 $\$s1 = 1$
	小于常数设置	slti \$s1,\$s2,20	If $(\$s2 < 20)$ $\$s1 = 1$ else $\$s1 = 0$	和常数比较小于设置 $\$s1 = 1$
	低于常数设置	sltiu \$s1,\$s2,20	If $(\$s2 < 20)$ $\$s1 = 1$ else $\$s1 = 0$	和常数比较低于设置 $\$s1 = 1$
无条件跳转	直接跳转	j 2500	goto 2500*4	跳转到目标地址
	间接跳转	jr \$ra	goto \$ra	用在分支和子程序返回
	跳转并链接	jal 2500	$\$ra = PC + 4$; goto 2500*4	用在子程序调用



2.4操作数类型

● 寄存器操作数

编号	名称	用途
\$0	\$zero	常量0(constant value 0)
\$2-\$3	\$v0-\$v1	函数调用返回值(values for results and expression evaluation)
\$4-\$7	\$a0-\$a3	函数调用参数(arguments)
\$8-\$15	\$t0-\$t7	临时寄存器(temporary)
\$16-\$23	\$s0-\$s7	存储寄存器，C语言中定义的变量可以保存在这些寄存器中。同时这些寄存器也可以保存内存单元的起始地址（基地址）
\$24-\$25	\$t8-\$t9	临时寄存器(temporary)
\$28	\$gp	全局指针(Global Pointer)
\$29	\$sp	栈指针(Stack Pointer)
\$30	\$fp	帧指针(Frame Pointer)
\$31	\$ra	子程序调用返回地址(return address)



存储器操作数

- 字节类型数组在内存中的存储

3	101
2	10
1	1
0	0

内存地址 内存中的数据

- 字类型数组在内存中的存储

- 采取字节对齐的策略去存放数据：即半字类型的数据从偶地址开始存放，而字类型的数据从4的整数倍地址开始存放

12	101
8	10
4	1
0	0

内存地址 内存中的数据



例2. 1假设数组A是一个具有100个元素的字类型数组，其在内存中的起始地址（基地址）保存在寄存器\$*s2*中，*g*和*h*分别是保存在寄存器\$*s0*和\$*s1*中的数据，要完成C语言中的以下语句：

g=*h*+*A*[8];

汇编指令：

lw \$*t0*,32(\$*s2*)

#该指令从地址为\$*s2*+32的连续的4个内存单元取值保存到寄存器\$*t0*中

add \$*s0*,\$*s1*,\$*t0*

- 由于每个字类型的数据占据4个内存单元，所以A中的第8个元素的地址相对于该数组的基地址的偏移量为 8×4 ，而不是8。



立即数

- 立即数或常数。
- 立即数在处理器设计中，通常将它们和指令绑定在一起，成为指令的一部分，这样可以加快立即数操作指令的执行效率。
 - 如指令 `lw $t0, 32($s2)` 中的 32 即为一个立即数，
 - 指令 `addi $s1, $s2, 40` 中的 40 也是一个立即数，
 - 前者 32 是操作数地址中的立即数，后者 40 是立即数操作数



2.5 MIPS指令编码

● R型指令

■ 仅具有寄存器操作数的指令

op	rs	rt	rd	shamt	funct
6位（第一段）	5位（第二段）	5位（第三段）	5位（第四段）	5位（第五段）	6位（第六段）

- op：操作码的编码，表明该指令的基本功能
- rs：第一个源操作数寄存器的编码
- rt：第二个源操作数寄存器的编码
- rd：目的操作数寄存器的编码
- shamt：移位指令的移位次数编码
- funct：功能码，确定op域范围内的具体的指令功能

● I型指令

■ 含有立即数操作数的指令

op	rs	rt	常数地址 (constant address)
6位 (第一段)	5位(第二段)	5位 (第三段)	16位 (第四段)

- op : 操作码的编码, 表明该指令的基本功能;
- rs : 第二个操作数寄存器的编码;
- rt : 第一个操作数寄存器的编码;
- 常数地址 (constant address) : 常数或内存地址偏移量立即数的二进制编码。



● J型指令

- 为实现远距离的跳转，设计了一类伪直接跳转指令，这类指令为无条件跳转指令

6位操作码编码

26位跳转地址

- 实际内存地址为

PC的高4位

26位跳转地址

00



2.6常用MIPS汇编指令

● C语言函数

```
int sum_pow2(int b, int c)
{
    int pow2[8] = {1, 2, 4, 8, 16, 32, 64, 128};
    int a, ret;
    a = b + c;
    if (a < 8)
        ret = pow2[a];
    else
        ret = 0;
    return(ret);
}
```

已知变量pow2,a,ret保存在内存中，而参数b，c以及返回结果保存在寄存器中。



- 该函数执行哪些操作：

- 语句 $a=b+c$ ：首先执行 $b+c$ 的算术运算，然后再将该算术运算的结果保存到内存中。
- 语句 $\text{if}(a<8)$ ：比较判断内存值是否小于8，并且根据比较结果跳转到不同语句执行
- 函数的调用和返回

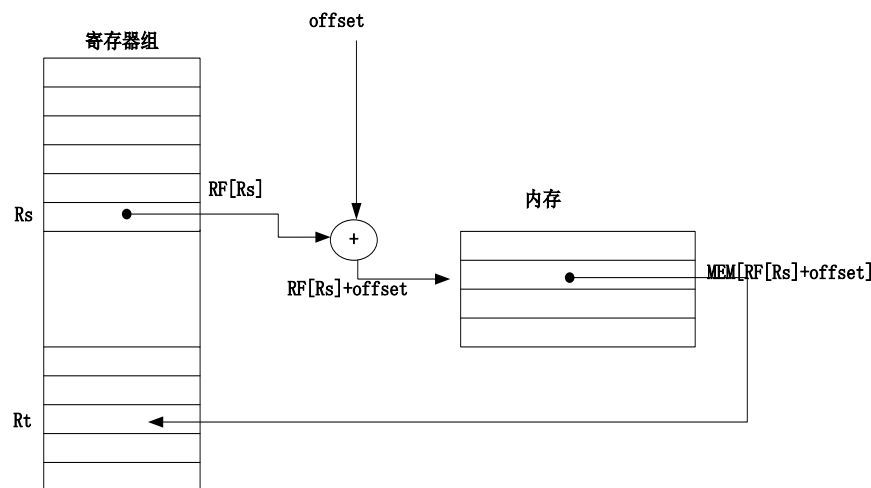
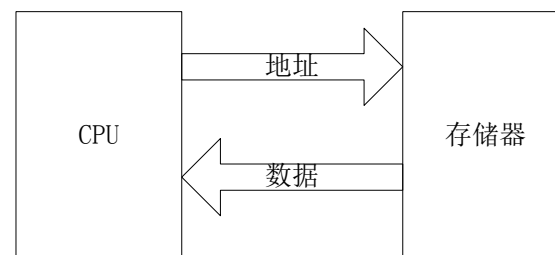
- 该C语言函数需要实现算术运算、数据存取以及程序控制等操作，这些操作都需要微处理器提供相应的指令来实现

数据传送指令

● 数据从存储器传送到寄存器称为装载（load）

$\text{lb Rt, offset(Rs) \# RF[Rt] = Mem[RF[Rs] + Offset]}$
 $\text{lbu Rt, offset(Rs) \# RF[Rt] = Mem[RF[Rs] + Offset]}$
 $\text{lh Rt, offset(Rs) \# RF[Rt] = Mem[RF[Rs] + Offset]}$
 $\text{lhu Rt, offset(Rs) \# RF[Rt] = Mem[RF[Rs] + Offset]}$
 $\text{lw Rt, offset(Rs) \# RF[Rt] = Mem[RF[Rs] + Offset]}$
 $\text{lwl Rt, offset(Rs) \# RF[Rt] = Mem[RF[Rs] + Offset]}$
 $\text{lwr Rt, offset(Rs) \# RF[Rt] = Mem[RF[Rs] + Offset]}$

- b表示字节传送，
- h表示半字传送，
- w表示字传送，
- u表示无符号扩展，不带u表示符号扩展





- 例3. 1假设寄存器\$*s0*=0x00000000，地址为0x00000000开始处内存单元存储的数据如图所示，执行以下指令之后，试问各*Rt*寄存器的值是多少？

lb \$*s1*,0(\$*s0*) \$*s1*=0xffffffff80

lbu \$*s1*,0(\$*s0*) \$*s1*=0x00000080

lh \$*s1*,0(\$*s0*) \$*s1*=0xffff8081.

lhu \$*s1*,0(\$*s0*) \$*s1*=0x00008081

lw \$*s1*,0(\$*s0*) \$*s1*=0x80818283

lwl \$*s1*,6(\$*s0*) \$*s1*=0x86858400

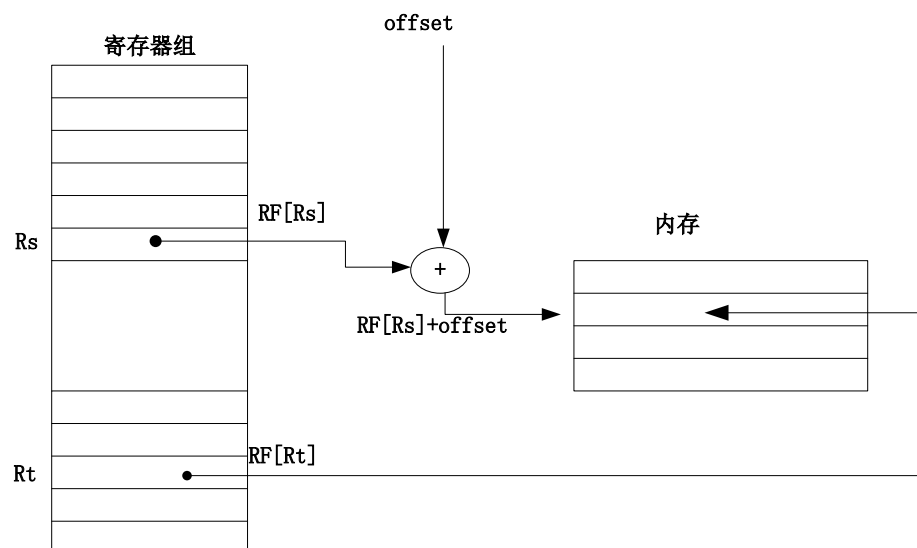
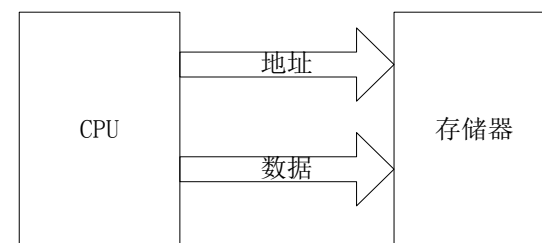
lwr \$*s1*,3(\$*s0*) \$*s1*=0x00000083

\$*s1*的初始值为0x00000000

0x00000000	0x80
0x00000001	0x81
0x00000002	0x82
0x00000003	0x83
0x00000004	0x84
0x00000005	0x85
0x00000006	0x86
0x00000007	0x87

● 数据从寄存器传送到存储器称为存储（store）

sb Rt, offset(Rs) # $\text{Mem}[\text{RF}[\text{Rs}] + \text{Offset}] = \text{RF}[\text{Rt}]$
sh Rt, offset(Rs) # $\text{Mem}[\text{RF}[\text{Rs}] + \text{Offset}] = \text{RF}[\text{Rt}]$
sw Rt, offset(Rs) # $\text{Mem}[\text{RF}[\text{Rs}] + \text{Offset}] = \text{RF}[\text{Rt}]$
swl Rt, offset(Rs) # $\text{Mem}[\text{RF}[\text{Rs}] + \text{Offset}] = \text{RF}[\text{Rt}]$
swr Rt, offset(Rs) # $\text{Mem}[\text{RF}[\text{Rs}] + \text{Offset}] = \text{RF}[\text{Rt}]$





- 假设寄存器\$*s*0=0x00000000,\$*s*1=0x81828384,地址为0x00000000开始处的内存单元存储的数据如图所示,执行以下指令之后,试问各内存单元的值是多少?

sb \$*s*1,0(\$*s*0)

sh \$*s*1,2(\$*s*0)

sw \$*s*1,4(\$*s*0)

swl \$*s*1,6(\$*s*0)

swr \$*s*1,3(\$*s*0)

0x00000000
0x00000001
0x00000002
0x00000003
0x00000004
0x00000005
0x00000006
0x00000007

0x00000000
0x00000001
0x00000002
0x00000003
0x00000004
0x00000005
0x00000006
0x00000007

0x00000000	0x84
0x00000001	0x81
0x00000002	0x83
0x00000003	0x84
0x00000004	0x84
0x00000005	0x83
0x00000006	0x81
0x00000007	0x84



● 特殊寄存器（low，high）到通用寄存器的数据传送指令

■ 将数据移出low或high寄存器

mfhi Rd # RF[Rd] = HIGH

mflo Rd # RF[Rd] = LOW

■ 将数据移入low或high寄存器

mthi Rs # HIGH = RF[Rs]

mtlo Rs # LOW = RF[Rs]



● 立即数到寄存器

`lui Rt, Imm` # $RF[Rt] = Imm \ll 16 \mid 0x0000$ Imm表示立即数



算术运算指令

● 加法运算

add Rd, Rs, Rt # $RF[Rd] = RF[Rs] + RF[Rt]$

addu Rd, Rs, Rt # $RF[Rd] = RF[Rs] + RF[Rt]$

addi Rt, Rs, Imm # $RF[Rt] = RF[Rs] + Imm$

addiu Rt, Rs, Imm # $RF[Rt] = RF[Rs] + Imm$

- 符号数加法结果产生溢出，微处理器将产生异常；
- 而无符号加法不会产生溢出异常。
- 立即数加法中
 - 如果是符号数加法，则16位立即数进行符号数扩充为32位之后再加；
 - 若为无符号加法则进行无符号扩充，即在高16位补0，然后再进行加法运算。



● 减法运算

sub Rd, Rs, Rt # $RF[Rd] = RF[Rs] - RF[Rt]$

subu Rd, Rs, Rt # $RF[Rd] = RF[Rs] - RF[Rt]$

● 乘法运算

mult Rs, Rt # High | Low = $RF[Rs] * RF[Rt]$

multu Rs, Rt # High | Low = $RF[Rs] * RF[Rt]$

● 除法运算

div Rs, Rt # Low = 商 ($RF[Rs] / RF[Rt]$); # High = 余数 ($RF[Rs] / RF[Rt]$)

divu Rs, Rt # Low = 商 ($RF[Rs] / RF[Rt]$); # High = 余数 ($RF[Rs] / RF[Rt]$)



逻辑运算指令

逻辑操作	C语言运算符	汇编指令	指令实例	指令含义
逻辑左移	<<	sll	sll \$s1,\$s2,10	将寄存器\$s2中的值左移10位，高10位移除，低10位补充0，结果保存到\$s1中
逻辑右移	>>	srl	srl \$s1,\$s2,10	将寄存器\$s2中的值右移10位，低10位移除，高10位补充0，结果保存到\$s1中
寄存器位与	&	and	and \$s1,\$s2,\$s3	将寄存器\$s2与\$s3中的值按位相与，结果保存到寄存器\$s1中
立即数位与	&	andi	andi \$s1,\$s2,40	将寄存器\$s2的值与40按位相与，结果保存到寄存器\$s1中
寄存器位或		or	or \$s1,\$s2,\$s3	将寄存器\$s2与\$s3中的值按位相或，结果保存到寄存器\$s1中
立即数位或		ori	ori \$s1,\$s2,40	将寄存器\$s2的值与40按位相或，结果保存到寄存器\$s1中
位或非	~	nor	nor \$s1,\$s2,\$s3	将寄存器\$s2与\$s3中的值按位相或非，结果保存到寄存器\$s1中
异或	^	xor	xor \$s1,\$s2,\$s3	将寄存器\$s2与\$s3中的值按位相异或，结果保存到寄存器\$s1中



● 移位指令机器指令格式

op (6位)	rs (5位)	rt (5位)	rd (5位)	shamt (5位)	func (6位)
操作码的编码	0	源操作数寄存器的编码	目的操作数寄存器的编码	移位次数的编码	操作码功能编码



程序控制类指令

● 条件跳转类指令：beq , bne

beq R1,R2,L1 #当寄存器R1的值与寄存器R2的值相等时，跳转到L1处执行指令。

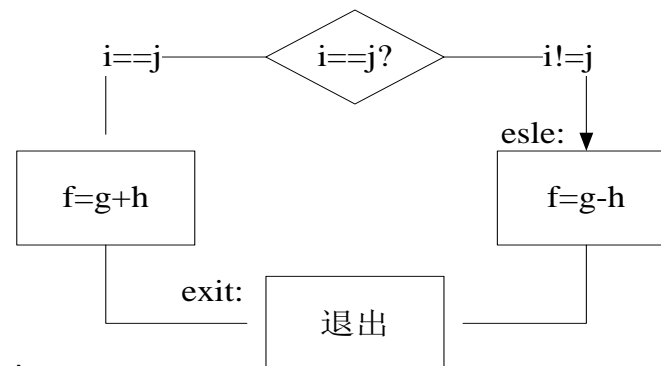
bne R1,R2,L1 #当寄存器R1的值与寄存器R2的值不相等时，跳转到L1处执行指令。

● 无条件跳转指令j

j L1#无条件跳转到L1处执行指令

● If 控制

```
if (i==j)
    f=g+h;
else f=g-h;
```



变量*i,j,f,g,h*分别对应寄存器*\$s0,\$s1,\$s2,\$s3,\$s4*

```
bne $s0,$s1,else # i!=j,跳转到else处，相等则顺序执行
add $s2,$s3,$s4
j exit      #处理完相等的情况后跳转到退出处
else: sub $s2,$s3,$s4
exit:
```


● for/while控制


```
while (save[i]==k)
    i+=1;
```

假定i, k为寄存器\$s0,\$s2, save的基地址保存在寄存器\$s3中

```
loop: sll $t1,$s0,2 #将i乘以4得到数组元素的偏移地址保存到暂存寄存器$t1中
      add $t1,$t1,$s3 #将数组元素在内存中的地址保存到暂存寄存器$t1中
      lw $t0,0($t1) #从内存中读取数组save中元素i的值保存到寄存器$t0中
      bne $t0,$s2,exit #如果save[i]不等于k则跳转到退出处, 否则顺序执行
      addi $s0,$s0,1 #数组元素加1得到下一个元素
      j loop #重复以上过程
exit:
```

条件判断

● =, != 

● <, >, <=, >= ?


● 比较设置指令

<u>Instruction</u>	<u>Example</u>	<u>Meaning</u>	<u>Comments</u>
set less than	<code>slt \$1, \$2, \$3</code>	$\$1 = (\$2 < \$3)$	comp less than signed
set less than imm	<code>slti \$1, \$2, 100</code>	$\$1 = (\$2 < 100)$	comp w/const signed
set less than uns	<code>sltu \$1, \$2, \$3</code>	$\$1 = (\$2 < \$3)$	comp < unsigned
set l.t. imm. uns	<code>sltiu \$1, \$2, 100</code>	$\$1 = (\$2 < 100)$	comp < const unsigned



if (i<j)

f=g+h;

else f=g-h;

\$t0=0.

slt \$t0, \$s0,\$s1 #若\$s0(i)小于\$s1(j), \$t0=1,否则

beq \$t0,\$zero,else

add \$s2,\$s3,\$s4

j exit

#处理完小于的情况后跳转到退出处

else: sub \$s2,\$s3,\$s4

exit:



● case/switch 控制

```
switch (i)
{
case 0:j=j+1;
break;
case 1:j=j+2;
break;
case 2:j=j+3;
break;
case 3:j=j+4;
break;
case 4: j=j+5;
break;
};
```

- i,j为寄存器 \$s0,\$s1,
- 常数0,1,2,3,4 分别保存在寄存器 \$t0,\$t1,\$t2,\$t3,\$t4中,
- 每个case对应的标号为: ca0,ca1,ca2,ca3,ca4。

逐次比较法

```
beq $s0,$t0,ca0 #比较i=0, 则跳转到ca0
beq $s0,$t1,ca1 #比较i=1, 则跳转到ca1
beq $s0,$t2,ca2 #比较i=2, 则跳转到ca2
beq $s0,$t3,ca3 #比较i=3, 则跳转到ca3
beq $s0,$t4,ca4 #比较i=4, 则跳转到ca4
j exit
ca0:addi $s1,$s1,1
j exit
ca1:addi $s1,$s1,2
j exit
ca2:addi $s1,$s1,3
j exit
ca3:addi $s1,$s1,4
j exit
ca4:addi $s1,$s1,5
exit:
```



● case/switch 控制

跳转表法

Jumptable:

```
.word ca0  
.word ca1  
.word ca2  
.word ca3  
.word ca4
```

偏移地址

0000	ca0
0004	ca1
0008	ca2
000C	ca3
0010	ca4

```
bltz $s0,exit    #小于0退出  
Slti $t0,$s0,5   #小于5设置为1  
Beq $t0,$zero,exit #大于等于5退出  
sll $t0,$s0,2     #将i*4得到标号在内存中的偏移地址  
add $s3,$s3,$t0  #偏移地址与基地址相加得到标号在内存中的地址  
lw $s4,0($s3)    #取出标号地址存放到寄存器$s4中  
jr $s4           #跳转到$s4所指示的标号处
```

Exit:



sum_pow2代码的MIPS汇编指令实现

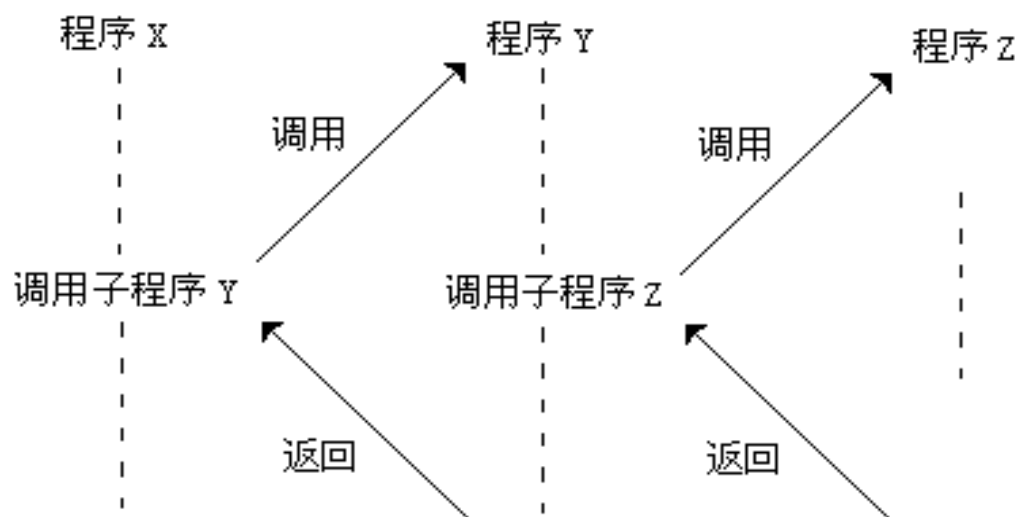
- 假定pow2的地址存储在寄存器\$V1中，b、c分别用寄存器\$t0,\$t1表示，a、ret用\$t2、\$t3表示

```
int sum_pow2(int b, int c)
{
    int pow2[8] = {1, 2, 4, 8, 16, 32, 64, 128};
    int a, ret;
    a = b + c;
    if (a < 8)
        ret = pow2[a];
    else
        ret = 0;
    return(ret);
}
```

```
add $t2,$t0,$t1          # a = b + c,
slti $v0,$t2,8           # $v0 = a < 8
beq $v0,$zero, Exceed   # 跳转到 Exceed if $v0 == 0
sll $v0,$t2,2            # $v0 = a*4
addu $v0,$v0,$v1        # $v0 = pow2 + a*4
lw $t3,0($v0)           # $t3 = pow2[a]
j Return                # 跳转到 Return
Exceed: addu $t3,$zero,$zero # $t3 = 0
Return:
```



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

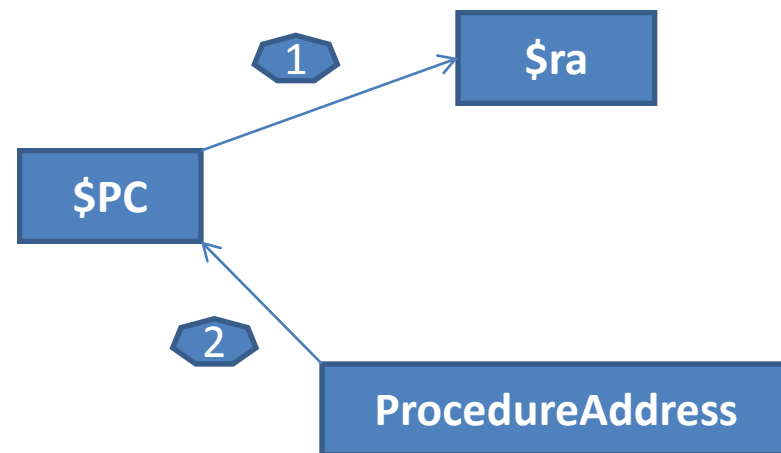
□\$t0~\$t9: 10个临时寄存器，在子程序调用时，子程序可以不保存其原始值就使用；

□\$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`

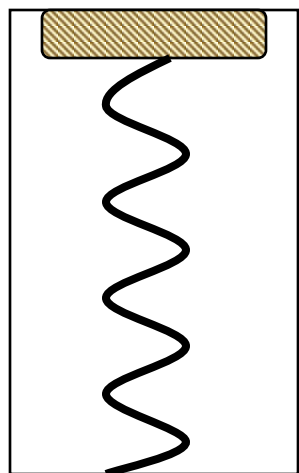




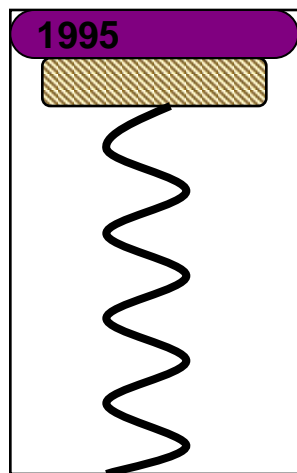
栈

- 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

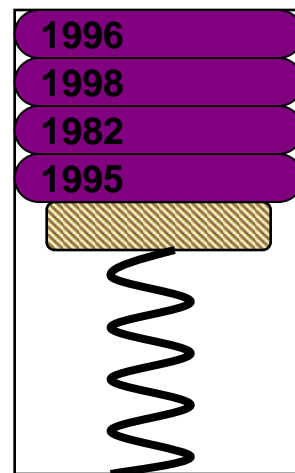
硬件栈



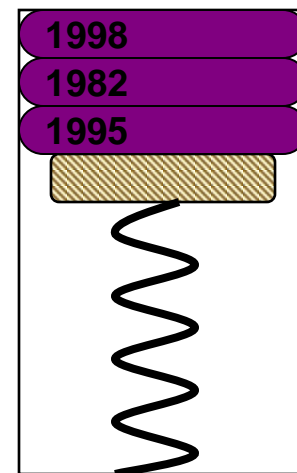
初始状态



压入一个数据



在压入三个数据

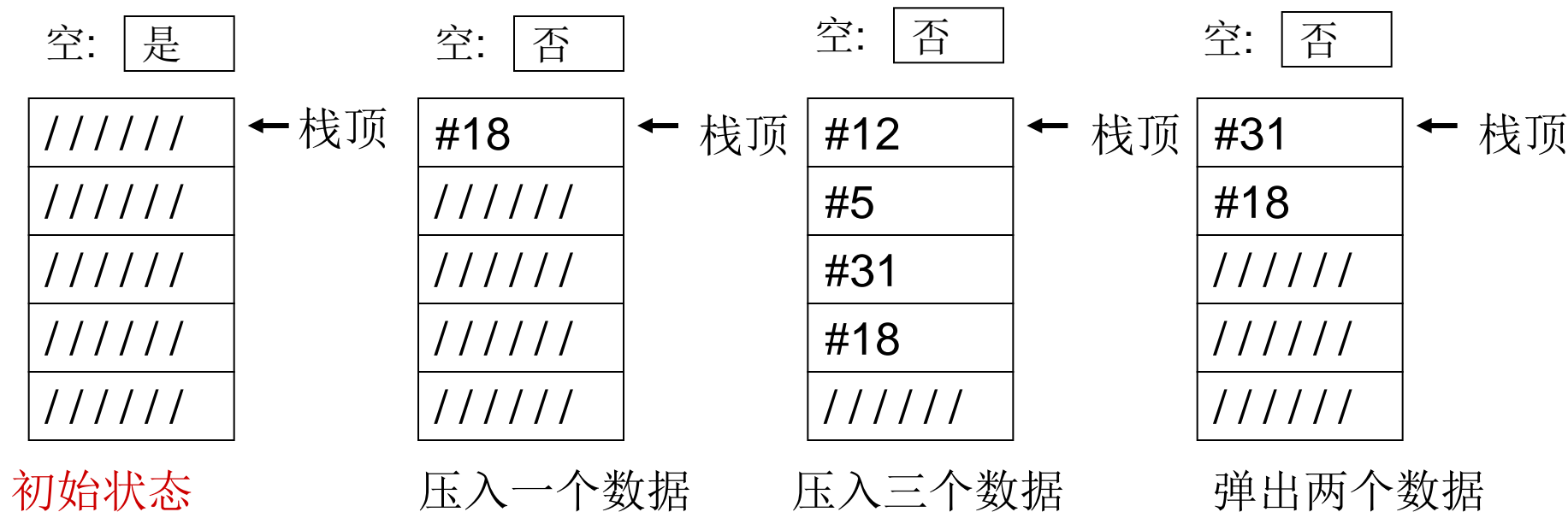


弹出一个数据



硬件栈的实现

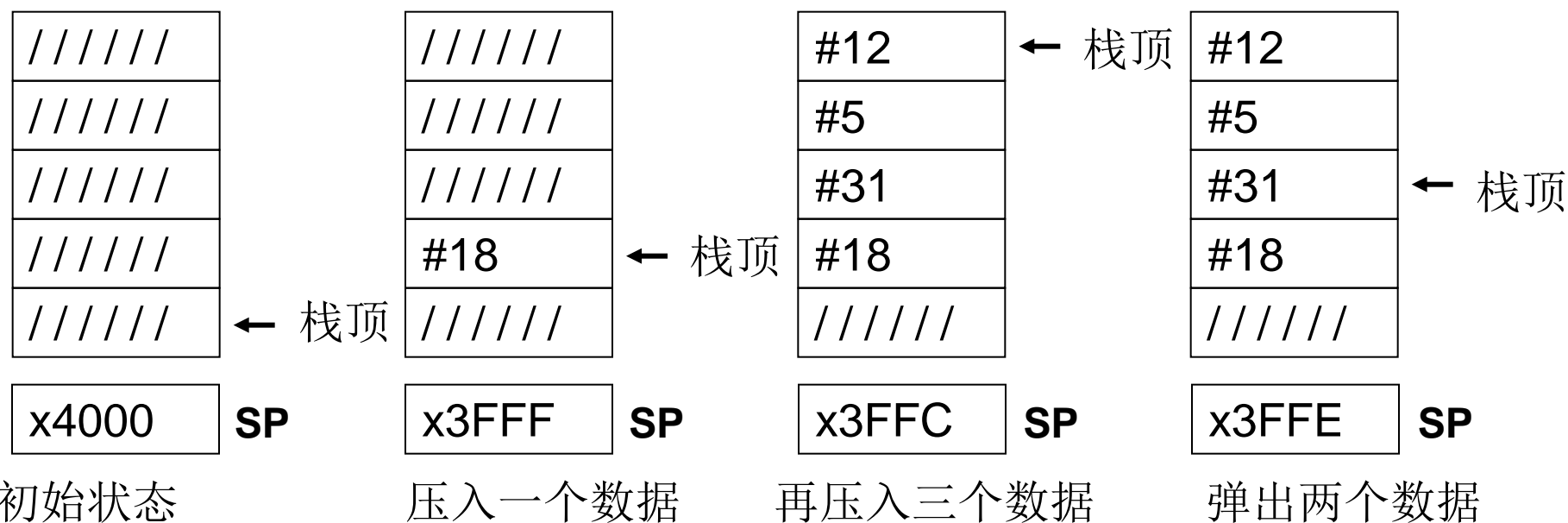
● 数据在栈中移动





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.



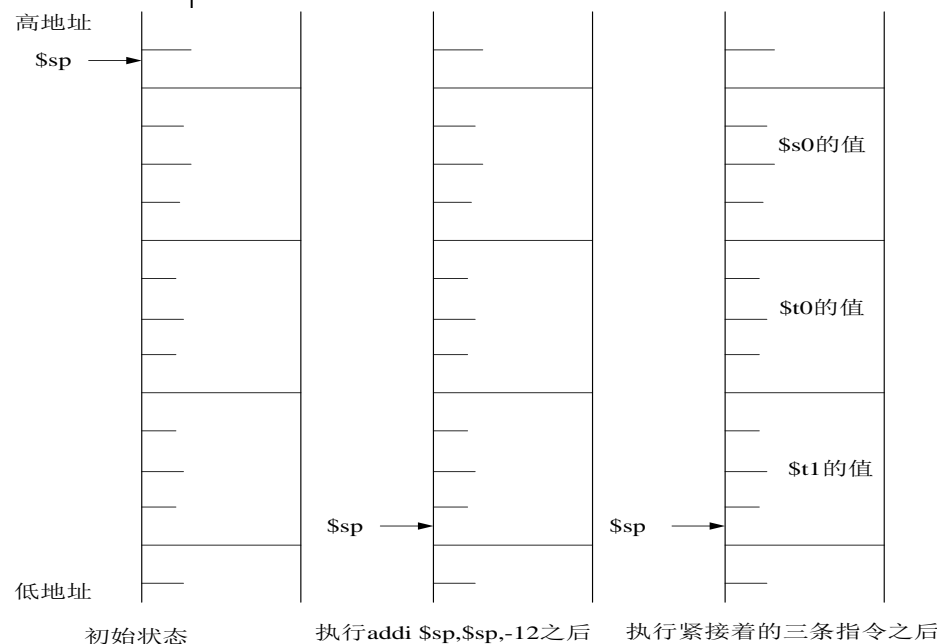
- 主程序和子程序公用寄存器问题的解决方法为：在子程序中事先保存其要使用的寄存器的值之后，再来使用该寄存器，在使用完之后，再恢复该寄存器的原始值。
- 如在子程序中要使用\$*s0*,\$*t0*,\$*t1*这三个寄存器

```
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)
```

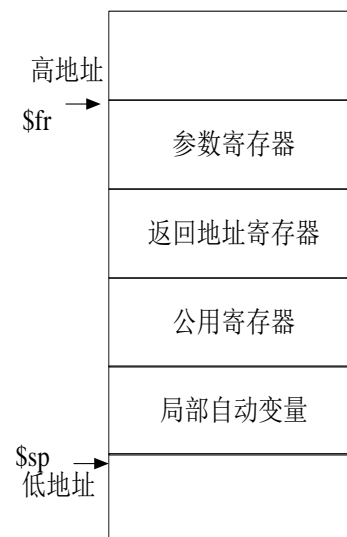
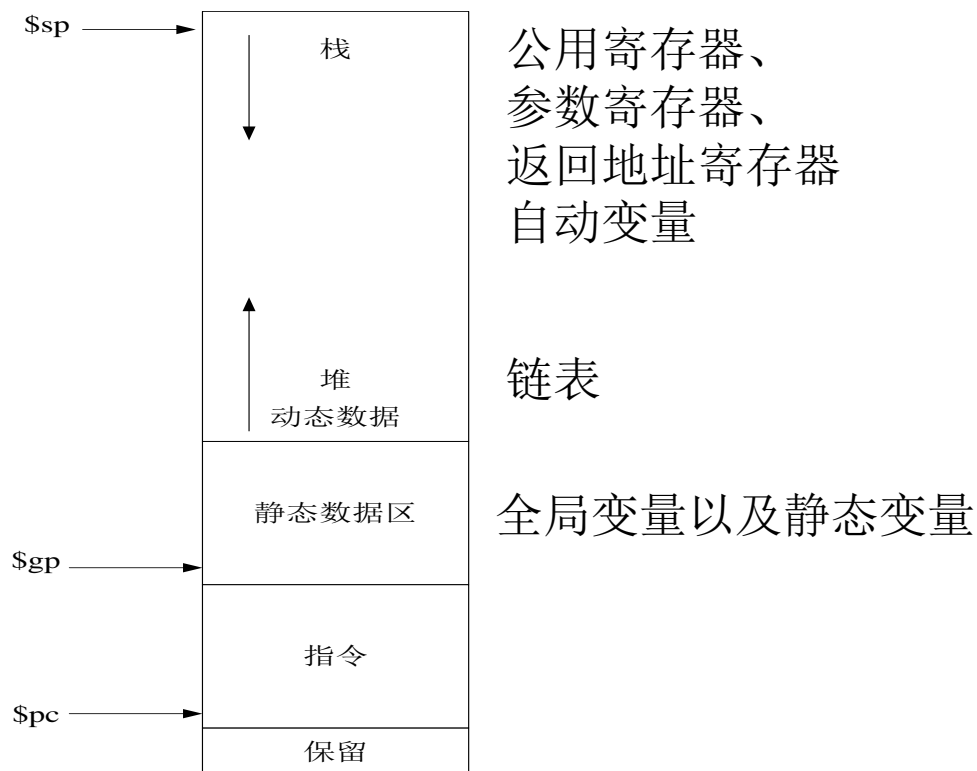
```
{  
    if(n<1) return (1);  
    else return (n*fact(n-1));  
}
```

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  
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
```

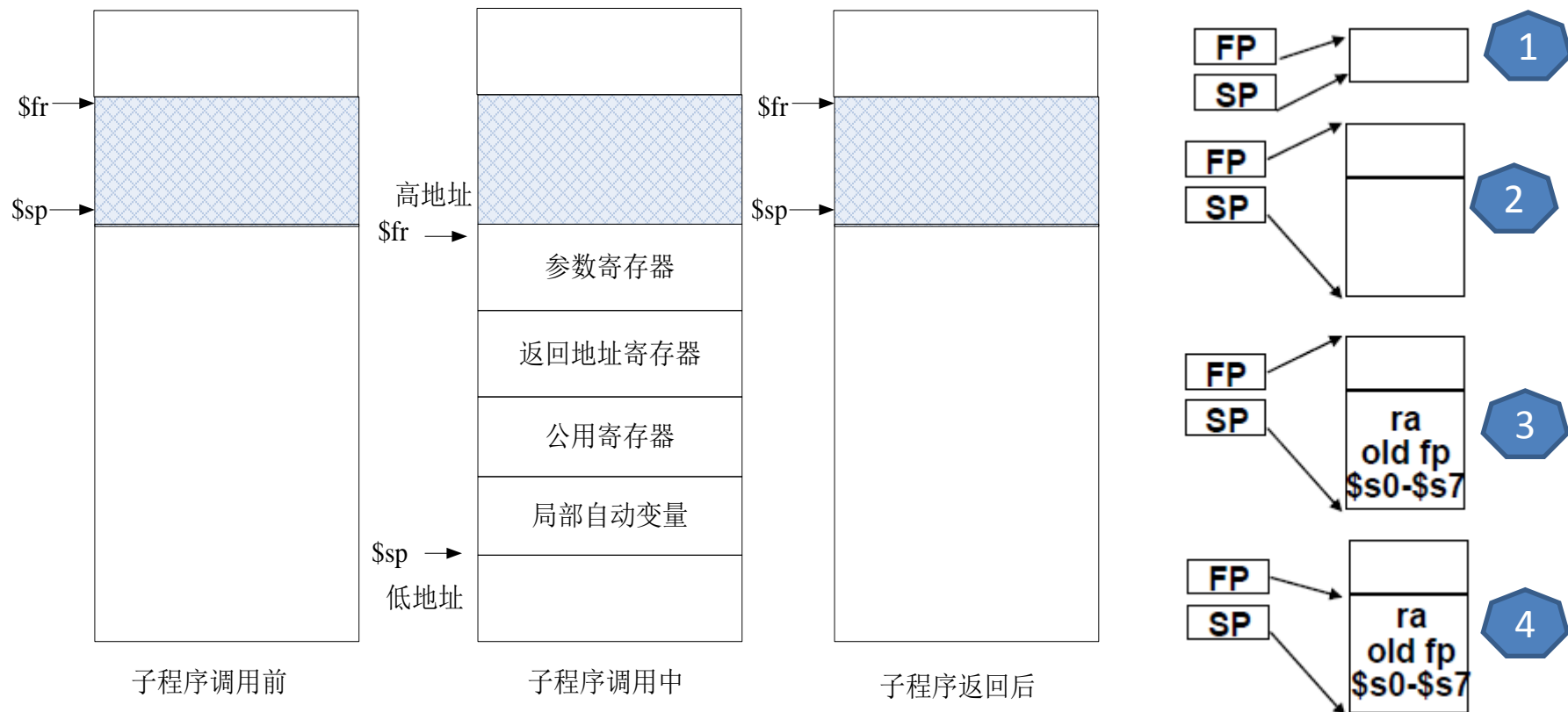
```
L1: addi $a0,$a0,-1#计算n-1的值  
jal fact# 嵌套调用,结果保存在$v0中  
lw $a0,0($sp) #从栈中获取当前子程序的入口参数  
lw $ra,4($sp) #从栈中获取当前子程序的返回地址  
mult $v0,$a0#计算返回结果实现n*fact(n-1)  
mflo $v0  
addi $sp,$sp,8  
jr $ra#返回
```

程序的内存映像



子程序被调用时栈的内存映像

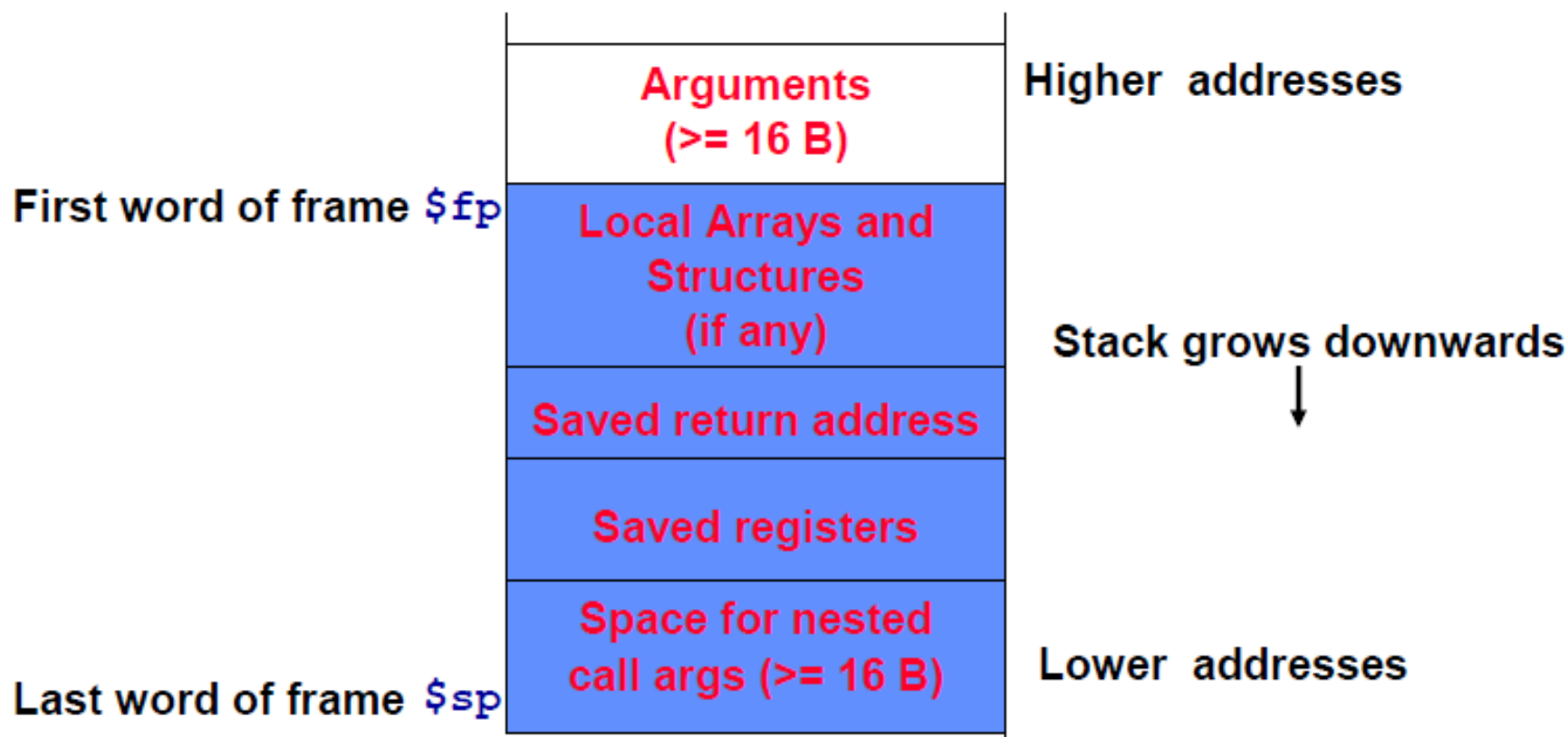
子程序嵌套调用栈的变化





GCC编译器栈的管理

- Each procedure creates an activation record on the stack
 - At least 32 bytes by convention to allow for \$a0-\$a3, \$ra, \$fp and be double double word aligned (16 byte multiple)





GCC寄存器使用规则

0	\$zero	Zero constant (0)	
1	\$at	Reserved for assembler	
2	\$v0	Expression results	
3	\$v1		
4	\$a0	Arguments	
5	\$a1		
6	\$a2		
7	\$a3		
8	\$t0	Caller saved	
...			
15	\$t7		
16	\$s0	Callee saved	
...			
23	\$s7		
24	\$t8	Temporary (cont'd)	
25	\$t9		
26	\$k0	Reserved for OS kernel	
27	\$k1		
28	\$gp	Pointer to global area	
29	\$sp	Stack Pointer	Callee saved
30	\$fp	Frame Pointer	
31	\$ra	Return address	



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 ($\$sp = \$sp - \text{frame}$)
 2. Save callee-saved registers $\$s0-\$s7, \$fp, \ra
 3. Set frame pointer ($\$fp = \$sp + \text{frame size} - 4$)
- Callee Return
 - Place return value in $\$v0$ and $\$v1$
 - Restore any callee-saved registers
 - Pop stack ($\$sp = \$sp + \text{frame size}$)
 - Return by `jr $ra`



GCC编译结果

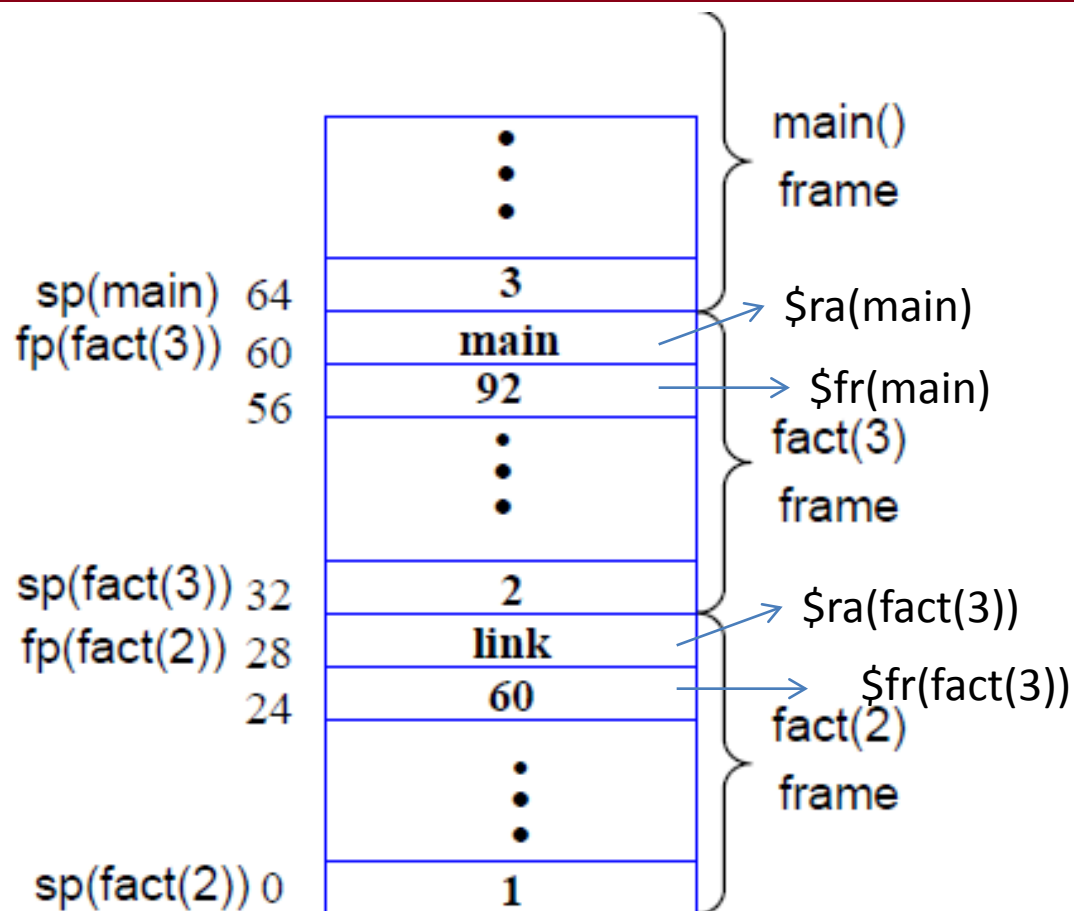
```
int fact(int n)
{
    if (n <= 1)
        return(1);
    else
        return(n*fact(n-1));
}
```

```
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
```



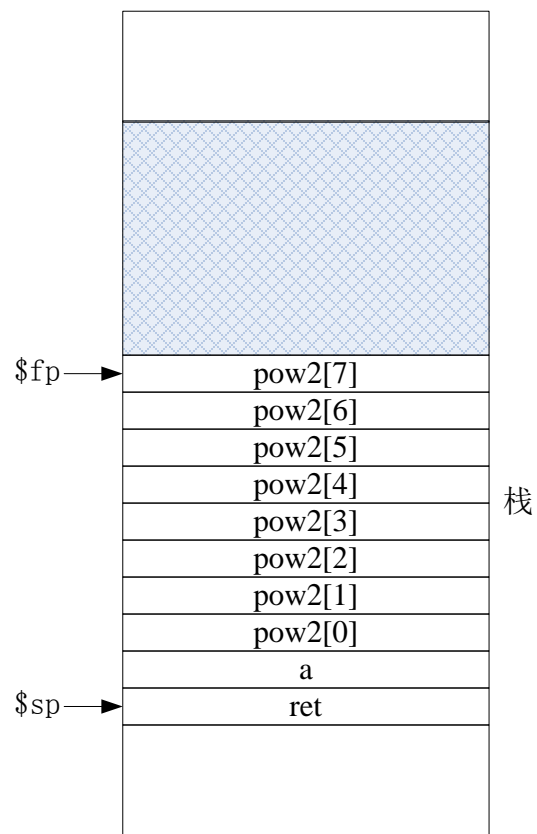
Fact函数递归调用栈变化示例

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





sum_pow2子程序栈内存映像



进入子程序中

sum_pow2按照编译规则得到的汇编代码

```

sum_pow2:                # $a0 = b, $a1 = c
    addu $a0,$a0,$a1      # a = b + c, $a0 = a
    slti $v0,$a0,8        # $v0 = a < 8
    beq $v0,$zero, Exceed # 跳转到Exceed if $v0 == 0
    addiu $v1,$sp,8       # $v1 = pow2 地址
    sll $v0,$a0,2         # $v0 = a*4
    addu $v0,$v0,$v1      # $v0 = pow2 + a*4
    lw $v0,0($v0)         # $v0 = pow2[a]
    j Return              # 跳转到Return

Exceed:  addu $v0,$zero,$zero # $v0 = 0

Return:  jr ra            # sum_pow2子程序返回
    
```



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,$a1 #将X字符串中的字符地址保存在$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
```



2.9寻址原理

- 寻址指微处理器获取其操作对象，包括数据和指令存储位置的方式
 - 操作数寻址
 - 指令寻址



操作数寻址

● 立即数寻址

- 立即数直接编码在机器指令中，因此与指令存储在一起

lui \$t0,32

001111	00000	01000	0000 0000 0010 0000
--------	-------	-------	---------------------

立即数

● 寄存器寻址

- 指令中的操作数为寄存器时，称为寄存器寻址

- add \$s0,\$t0,\$zero

● 基址寻址

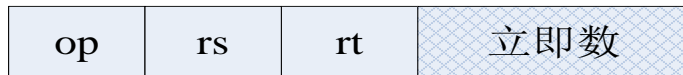
- 通过某个寄存器加上一个立即数来指示该内存单元的地址，这种方式就称为基址寻址

- lw \$t0,4(\$sp)



操作数寻址数据存放位置示例

1. 立即数寻址 `addi $t0,$s0,56`



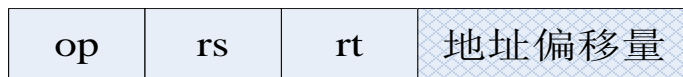
2. 寄存器寻址 `add $t0,$s0,$s1`



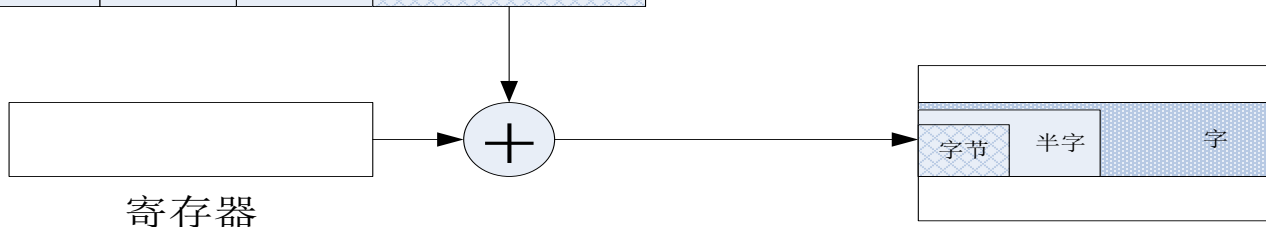
寄存器



3. 基址寻址 `lw $t0,4($s1)`



内存





指令寻址

● PC相对寻址

- 要寻找的下一条指令的地址由PC寄存器的值和一个相对偏移量构成
- `beq $t2,$zero,L2;`

6位操作码编码	5位第一寄存器编码	5位第二寄存器编码	16位偏移量编码
---------	-----------	-----------	----------

`beq $t2,$zero,L2`#如果字符的值为0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

跳转范围为16位有符号数所能表示的范围*4,
即 $(-2^{16} \sim 2^{16} - 1) * 4$

新PC的值=PC原始值+16位立即数*4

000100	01010	00000	0000 0000 0000 0010
--------	-------	-------	---------------------

偏移指令条数

● 伪直接寻址

指令编码

6位操作码编码	26位跳转地址
---------	---------

新的PC值

PC的高4位	26位跳转地址	00
--------	---------	----

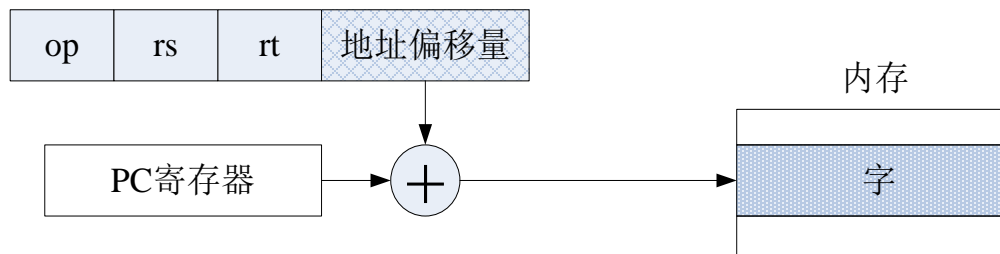
● 寄存器间接寻址

- 利用寄存器保存指令内存地址，如指令 `jr $ra`
- 直接将指令中所表示的寄存器值赋给 `$pc`

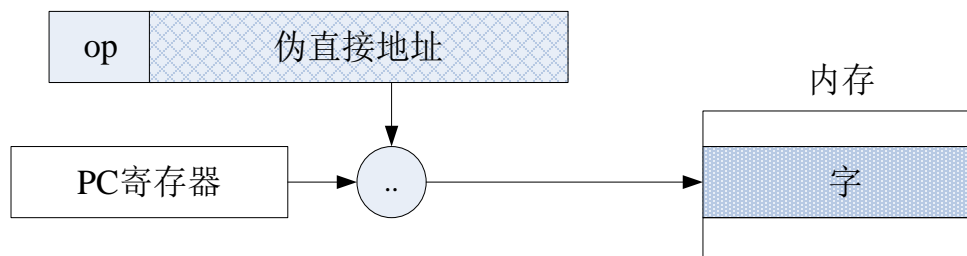


指令寻址方式原理

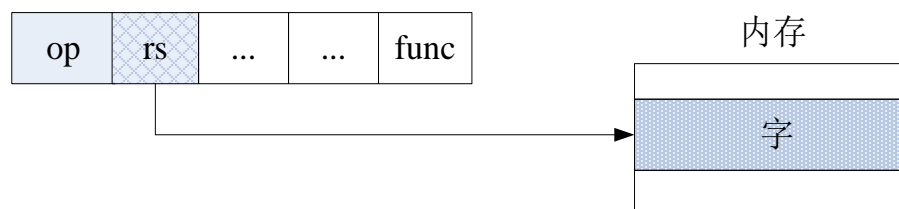
1. PC相对寻址 beq \$s0,\$s1,L2



2. 伪直接寻址 J L2



3. 寄存器间接寻址 jr \$ra

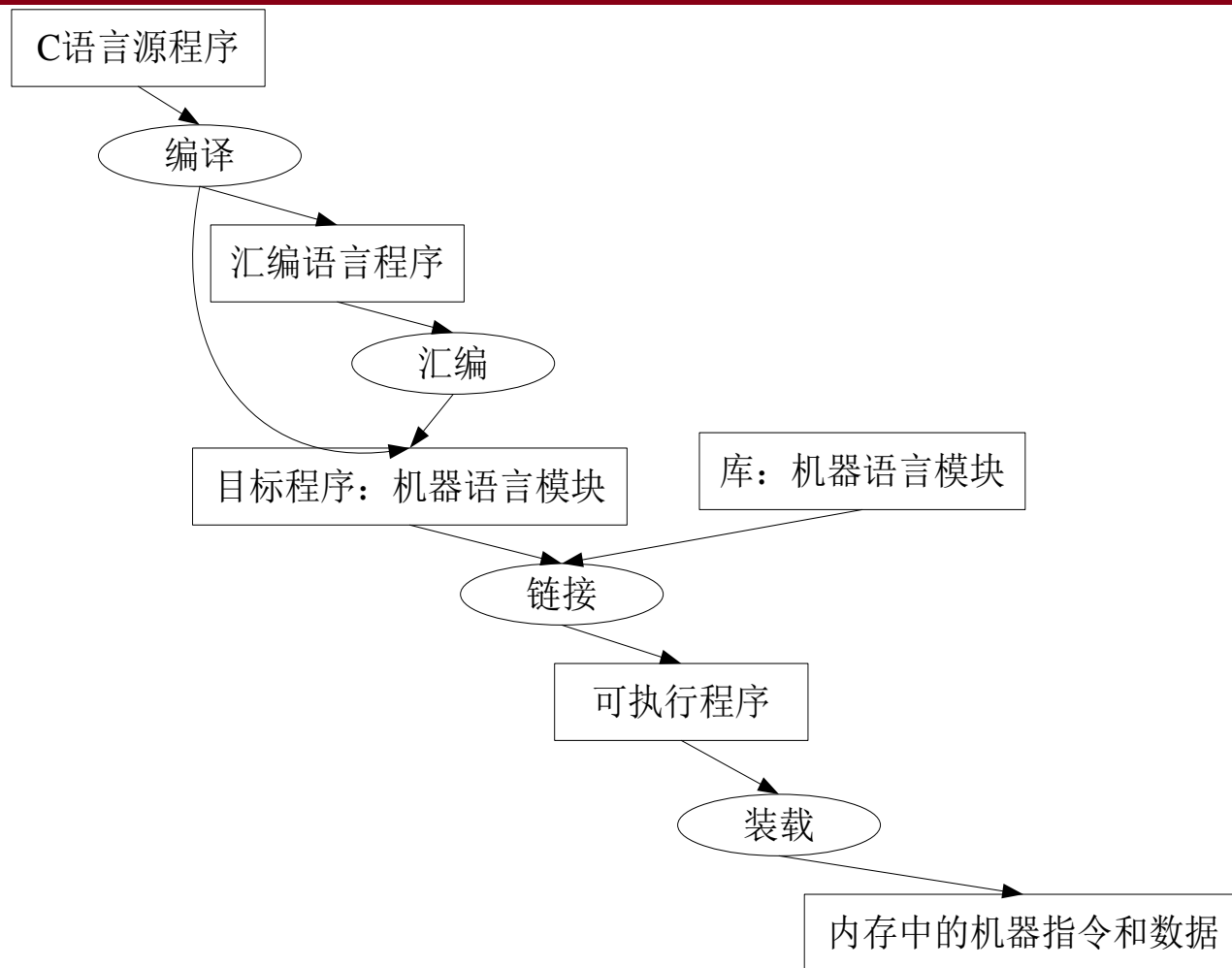




2.10编译、汇编、链接、装载过程

- 高级语言源程序：是指采用某种计算机高级语言编写的可以被人读懂的计算机程序。
- 汇编语言程序：是指采用汇编语言编写的可以被人读懂的计算机程序。此两种程序都不能被计算机直接执行。
- 编译器：是指将高级语言程序翻译成汇编语言程序的软件。
- 汇编程序：是指将汇编语言程序翻译为机器指令序列的软件。
- 目标程序：是指经汇编程序翻译而获得的机器指令序列的程序。此程序仍然不能被计算机执行。
- 链接：是指将各个相互关联的目标程序（包括库文件）连接起来组成可被计算机直接执行的程序的过程。
- 可执行程序：可以被系统软件调用并被计算机直接执行的程序。
- 程序装载：是指将可执行程序装载到内存，并实现相关内存、寄存器等初始化工作，并使得微处理器跳转到可执行程序的入口的过程。

高级语言源程序到计算机执行所经历的过程





2.11 汇编程序设计

- 宏汇编语言有3类基本指令：符号指令、伪指令和宏指令。
 - 伪指令只为汇编程序将符号指令翻译成机器指令提供信息，没有与它们对应的机器指令
 - 把一个指令序列定义为一条宏指令



spim MIPS仿真器汇编伪指令

- `.align n` 紧接着的内存地址以 2^n 的地址实现字节边界对齐。
 - 如`.align 2` 表示下一个内存地址以字实现边界对齐；
 - 而`.align 0` 则关闭`.half`, `.word`, `.float` 以及`.double`等伪指令的自动边界对齐，直到碰到下一个`.data` 或 `.kdata`。
- `.ascii str` 在内存中存储`str`字符串，不包含字符串结束符`null`。
- `.asciiz str`在内存中存储`str`字符串，且包含字符串结束符`null`。
- `.byte b1, ..., bn` 在`n`个连续的内存空间中依次存储字节`b1, ..., bn`
- `.word w1, ..., wn`在内存中连续存放`n`个字`w1, ..., wn`
- `.space n` 分配`n`个连续的字节存储空间
- `.globl sym` 声明全局变量`sym`，这样`sym`就可以被外部文件使用
- `.data <addr>`定义用户数据段，`addr`为可选参数，`addr`用来定义用户数据段的起始地址。紧接着定义的内容将存放在用户数据段
- `.text <addr>`定义用户代码段，`addr`为可选参数，`addr`用来定义用户代码段的起始地址。紧接着定义的内容将存放在用户代码段。在`spim`中紧接着只能为指令。
- `.kdata <addr> .ktext <addr>`分别定义系统内核数据段和代码段。



● 数据

- 字符串采用双引号括起来，
- 特殊字符的定义与C语言的规范基本一致，如：
 - 换行符 `\n`
 - Tab `\t`
 - 引号 `\` ”
- 数值的不同进制表示与C语言基本一致，
 - 任何前缀的数为十进制数，
 - 十六进制数以0x开头。

- 数据在内存中的地址通过变量来表示，变量在汇编语言中的定义方式与标号类似，即在定义数据的伪指令前，写上变量名，并且采用冒号隔开。
- 变量与标号的区别在于：
 - 变量表示数据在内存中的地址，
 - 标号表示指令在内存中的地址。

```
.data 0x10014000
.align 2
str: .ascii "abcd"
strn: .asciiz "abcdefg"
b0: .byte 1,2,3,4,5
h0: .half 1,2,3,4
w0: .word 1,2,3,4
```

		str									strn									b0			
0x1001	4000	0x61	0x62	0x63	0x64	0x61	0x62	0x63	0x64	0x65	0x66	0x67	0x0	0x1	0x2	0x3	0x4						
		h0									w0												
0x1001	4010	0x5	0x0	0x0	0x1	0x0	0x2	0x0	0x3	0x0	0x4	0x0	0x0	0x0	0x0	0x0	0x1						
0x1001	4020	0x0	0x0	0x0	0x2	0x0	0x0	0x0	0x3	0x0	0x0	0x0	0x4										



地址表达式

```
.data 0x10014000  
.align 2  
str: .ascii "abcd"  
strn: .ascii "abcdefg"  
b0: .byte 1,2,3,4,5  
h0: .half 1,2,3,4  
w0: .word 1,2,3,4  
w1: .word str,strn,b0,h0,w0
```

	str	strn											b0				
0x1001 4000	0x61	0x62	0x63	0x64	0x61	0x62	0x63	0x64	0x65	0x66	0x67	0x0	0x1	0x2	0x3	0x4	
	h0												w0				
0x1001 4010	0x5	0x0	0x0	0x1	0x0	0x2	0x0	0x3	0x0	0x4	0x0	0x0	0x0	0x0	0x0	0x1	
													w1				
0x1001 4020	0x0	0x0	0x0	0x2	0x0	0x0	0x0	0x3	0x0	0x0	0x0	0x4	0x10	0x01	0x40	0x00	
0x1001 4030	0x10	0x01	0x40	0x04	0x10	0x01	0x40	0x0c	0x10	0x01	0x40	0x12	0x10	0x01	0x40	0x1c	



宏指令

- **取变量或标号的地址:**

- $l a \text{ Rd, Label}$

- **立即数初始化**

- $l i \text{ Rd, value}$



系统功能调用

● 实现键盘输入和显示器输出等人机接口

功能描述	功能号 (\$v0)	入口参数	出口参数	备注
输出一个十进制整数	1	\$a0=需要输出的数值	无	
输出字符串	4	\$a0=字符串首地址	无	输出的字符串以字符串结束符标志
从键盘缓冲区读入一个十进制整数	5	无	\$v0=输入的整数	回车表示输入结束
从键盘缓冲区读入字符串	8	\$a0=保存字符串的首地址 \$a1=预留的内存空间的大小	输入的字符的ASCII顺序存放在以\$a0开始的连续的内存单元中	回车表示输入结束，实际能输入的字符个数为\$a1-1，实际存放在内存中的字符串为用户输入的字符串+回车符0x0a
退出	10	无	无	

系统功能调用步骤为：

- ❑ 将功能号赋给 \$v0;
- ❑ 设置好入口参数;
- ❑ syscall;

输出整数256,

- ❑ li \$v0,1 # 将功能号1赋给 \$v0
- ❑ li \$a0,256 # 将要显示的数据256赋给入口参数 \$a0
- ❑ syscall



功能调用举例

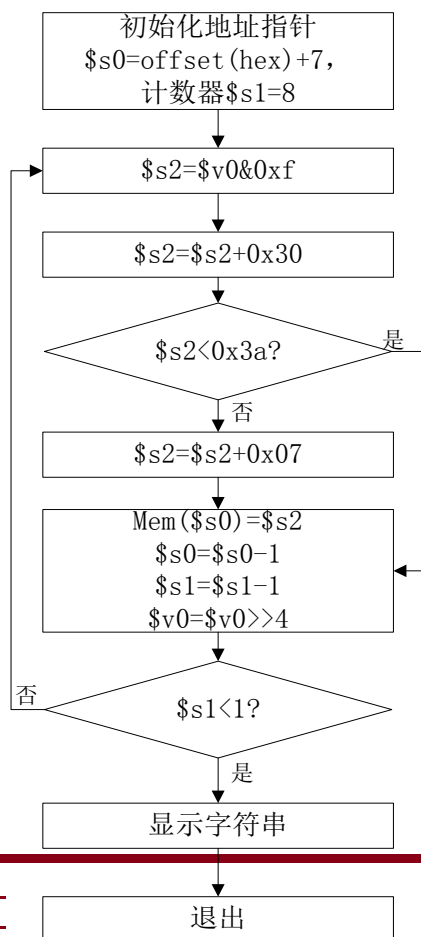
```
.data 0x10014000
.align 2
str: .ascii "abcd"
strn: .asciiz "ABCDEFGH"
b0: .byte 1,2,3,4,5
.text
main:li $v0,1
      li $a0,0x200
      syscall #输出十进制数据512①
      li $v0,4
      la $a0,str
      syscall #输出字符串str②
      la $a0,strn
      syscall #输出字符串strn③
      li $v0,5
      syscall #输入十进制整数④
      li $v0,8
      la $a0,b0
      li $a1,5
      syscall #输入字符串⑤
      li $v0,10
      syscall #程序结束退出
```



汇编程序设计举例

● 显示16进制数

■ 以16进制形式显示\$v0寄存器中的32位二进制数



```
.data
str: .byte 0x20,0x30,0x78 #16进制显示前缀
hex: .space 8 #预留8个内存单元
     .byte 0x00 #字符串结束符
.text
main:
    li $v0,0x12AF5678
    la $s0,hex
    addi $s0,$s0,7
    li $s1,8 #初始化
rep:  andi $s2,$v0,0xf #获取低4位
     addi $s2,$s2,0x30 #转换为ASCII码
     slti $t0,$s2,0x3a
     bne $t0,$0,less
     addi $s2,$s2,0x07
less: sb $s2,0($s0) #存储转换结果
     addi $s0,$s0,-1
     addi $s1,$s1,-1
     srl $v0,$v0,4
     slti $t0,$s1,1
     beq $t0,$0,rep
    li $v0,4 #显示整个转换后的字符
    la $a0,str
    syscall
    li $v0,10 #退出
    syscall
```

● 从键盘输入16



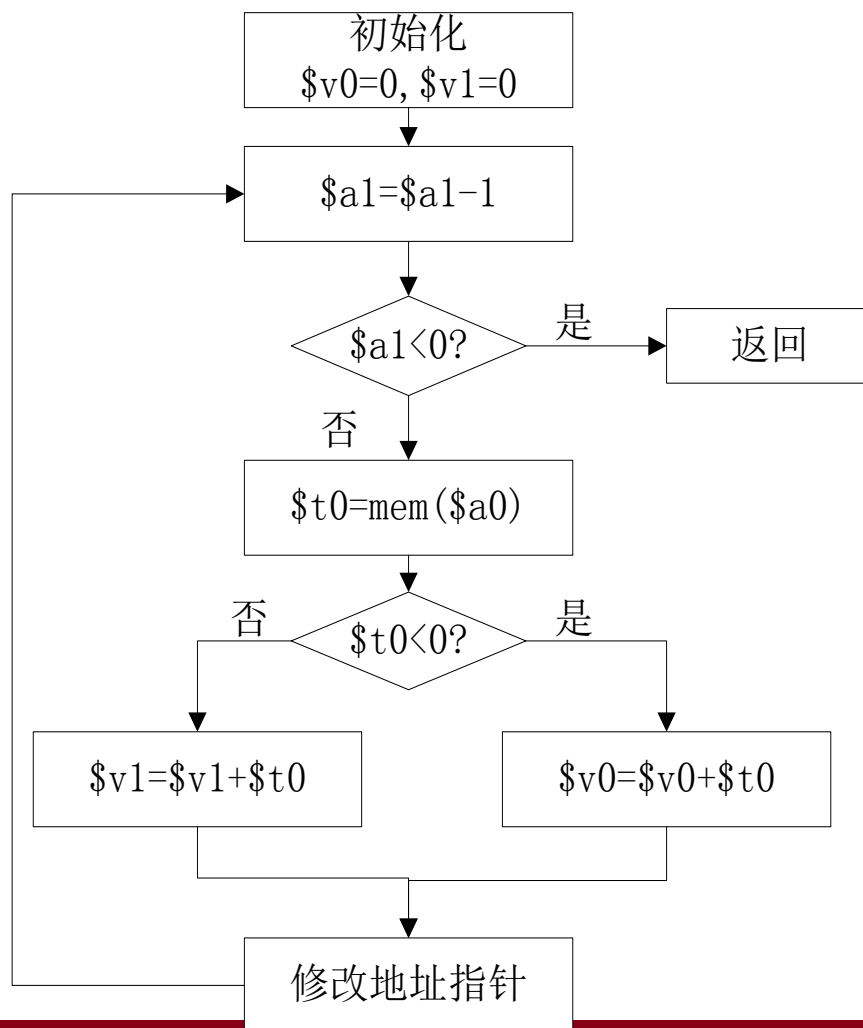


子程序设计示例

- 计算某数组中所有正数的和和所有负数的和，并分别显示结果。要求采用子程序计算数组中所有正数的和和所有负数的和。
 - 有两个输入参数：地址指针和计数器
 - 两个出口参数：正数的和和负数的和
 - \$a0,\$a1分别作为地址指针和计数器，\$v0,\$v1分别传递正数的和和负数的和



求和子程序流程





假定该数组为字类型的数据

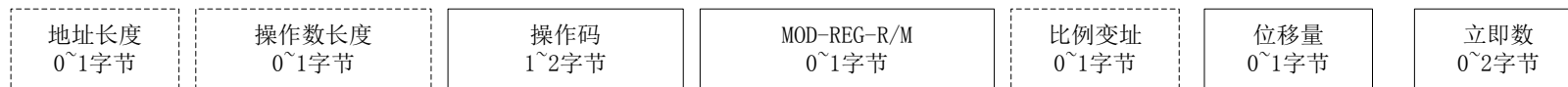
```
.data
array: .word -1,3,4,-5
posi: .ascii "\nthe sun of positive numbers a
nega: .ascii "\nthe sun of negative numbers
.text
main:
    li $v0,4
    la $a0,posi
    syscall #显示字符串posi
    la $a0,array #准备好入口参数
    li $a1,4
    jal sum #调用求和子程序
    add $a0,$v0,$0
    li $v0,1
    syscall #显示正数的和
    li $v0,4
    la $a0,nega
    syscall #显示字符串nega
    add $a0,$v1,$0
```

```
li $v0,1
syscall #显示负数的和
li $v0,10
syscall #退出
sum:
    li $v0,0
    li $v1,0
loop:
    blez $a1, retzz # 如果(a1 <= 0) 则返回
    addi $a1, $a1, -1 # 计数器减1
    lw $t0, 0($a0) # 从数组中获取一个元素
    addi $a0, $a0, 4 # 指向下一个元素
    bltz $t0, negg # 如果是复数跳转到negg
    add $v0, $v0, $t0 # 加到正数的和$v0
    b loop # 重复处理下一个元素
negg:
    add $v1, $v1, $t0 # 加到负数的和$v0
    b loop # 重复处理下一个元素
retzz:
    jr $ra # 返回
```


2.12 intel x86微处理器指令集简介



(a) 16位指令格式



(b) 32位指令格式

EAX		AH	AX	AL	累加器 (Accumulator)
EBX		BH	BX	BL	基址寄存器 (Base register)
ECX		CH	CX	CL	计数寄存器 (Count register)
EDX		DH	DX	DL	数据寄存器 (Data register)
ESP		SP			堆栈指示器 (Stack Point)
EBP		BP			基址指示器 (Base Point)
ESI		SI			源变址寄存器 (Source Index)
EDI		DI			目的变址寄存器 (Destination Index)
EIP		IP			指令指示器 (Instruction Point)
EFLAGS		F			状态标志寄存器 (status Flags)
		CS			代码段寄存器 (Code Segment)
		DS			数据段寄存器 (Data Segment)
		SS			堆栈段寄存器 (Stack Segment)
		ES			附加段寄存器 (Extra Segment)
		FS			
		GS			



寻址方式

寻址类型	指令示例	源	目的
寄存器寻址	MOV AX,BX	BX	AX
立即寻址	MOV CH,3AH	3AH	CH
直接寻址	MOV [1234H], AX	AX	DS:[1234H]
寄存器间接寻址	MOV [BX], CL	CL	DS:[BX]
基址加变址寻址	MOV [BX+SI],BP	BP	DS:[BX+SI]
寄存器相对寻址	MOV CL, [BX+4]	DS:[BX+4]	CL
基址加变址寻址	MOV ARRAY[BX+51],DX	DX	DS:[ARRAY+BX+51]
比例变址寻址	MOV [EBX+2 * ESI],AX	AX	DS:[EBX+2*ESI]



常用指令

指令		含义
程序控制类指令	jz,jnz	条件跳转，判断z标志位为0否再决定跳转
	jmp	无条件跳转
	call	调用子程序
	ret	子程序返回
	loop	根据ECX的值循环执行一段指令
数据传输	mov	寄存器间，寄存器与存储器间数据传输
	push,pop	栈操作，数据在栈与寄存器间传输
	les	装在内存数据到通用寄存器和ES段寄存器
算术逻辑运算	add,sub	加减运算，不区分符号数和无符号数
	cmp	比较两操作数的大小，不回送结果，仅改变标志位
	shl,shr,rcr	逻辑左移，逻辑右移，带进位循环右移
	cbw	字节（8位）符号数扩展为字（16位）符号数
	test	位与运算，但不回送结果，仅改变标志位
	inc,dec	加1，减1
	or,xor	位或，位异或
字符串操作	movs	字符串从一块内存区域传送到另一块内存区域
	lods	字符串中的字符从内存拷贝到EAX寄存器

作业



● 9.10.11, 14