数字逻辑与处理器基础

第七讲：计算机指令集系统 (2/2)

李学清

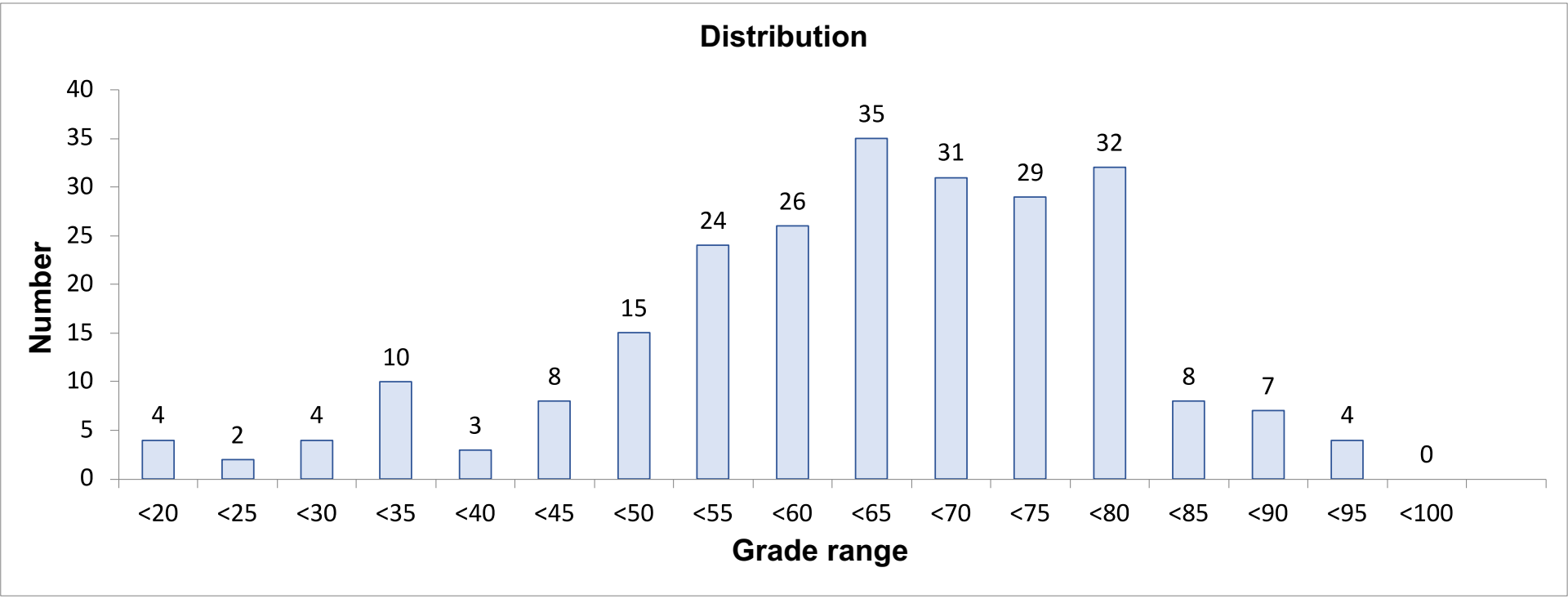
xueqingli@tsinghua.edu.cn

腾讯会议: 983-4578-6584

4/12/2022

期中考试

- 三个班共242人参加考试，平均分61.05，最高分94



作业

❖ 要求

- ✓ ddl: 8pm, 4/12/2022
- ✓ 标记学号，姓名和班级
- ✓ 不要抄袭或迟交
- ✓ 线下上课的同学交纸质版，线上上课同学交电子版

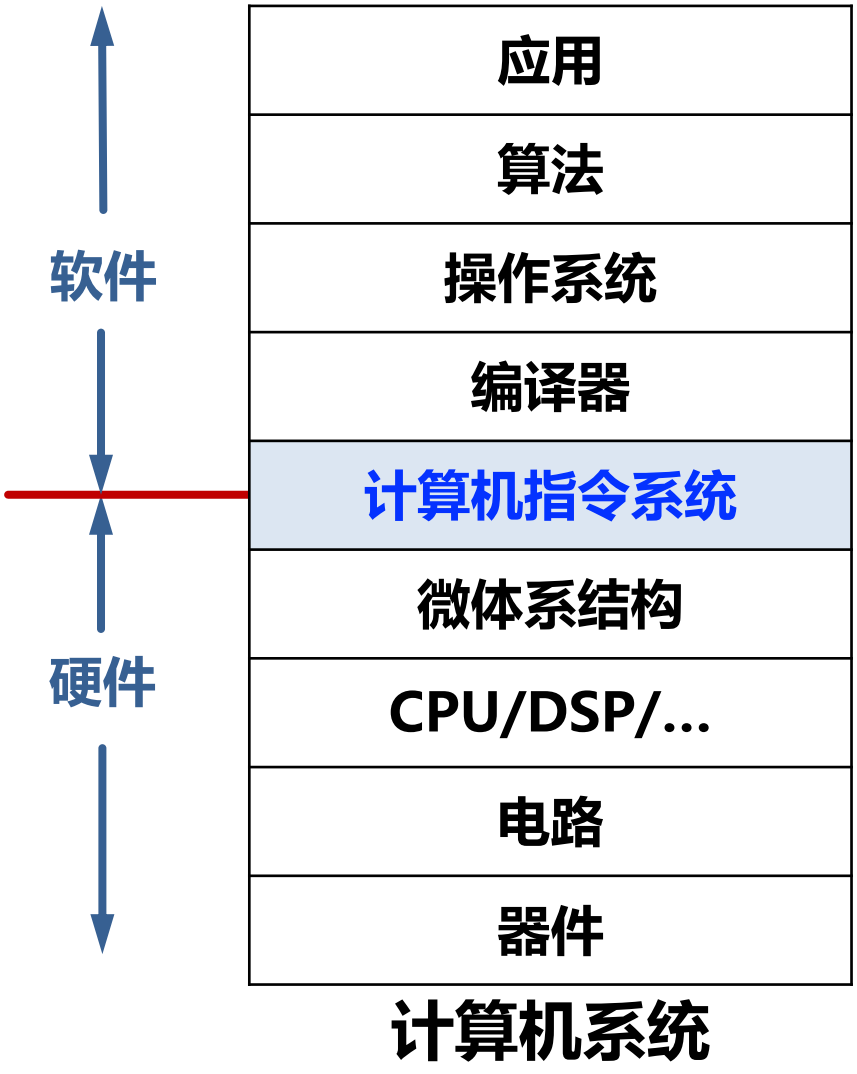
❖ 作业

- ✓ 教材MIPS指令集章节（第V章）习题4、5、8、15、16

第一次大作业：MIPS汇编

- **分两次布置**
  - ✓ 第一部分（4/12）：MIPS汇编基本练习
  - ✓ 第二部分（4/19）：使用汇编语言实现串匹配算法
- **提交时间：三周后**
  - ✓ 第一部分提交： 5/3
  - ✓ 第二部分提交： 5/10
- **习题课：本周日（4/17) 19：00 ~ 21：00**
  - ✓ 内容：第一次大作业讲解 + 期中试卷讲评
  - ✓ 地点：TBD

计算机指令系统



周	日期	主要内容
1	2/22	概论
2	3/1	布尔代数
3	3/8	组合逻辑
4	3/15	时序逻辑
5	3/22	时序逻辑
6	3/29	计算机指令系统
7	4/10	期中考试
8	**4/12**	**计算机指令系统**
9	4/19	微处理器设计
10	4/26	微处理器设计
11	5/7	劳动节放假5/3->5/7
12	5/10	微处理器/流水线
13	5/17	流水线技术
14	5/24	流水线技术
15	5/31	存储器技术
16	6/7	存储器-IO/总线/总结

本节课学习目标

- **知识点：指令集的概念、原理和评价指标**
  - ✓ ISA、CPI、执行时间、RISC/CISC、图灵机、存储结构、 多层次计算机系统、基础指令、MIPS ISA
- **能力**
  - ✓ 掌握分析和设计指令集的能力
  - ✓ 掌握评价计算机性能的方法
- **思想**
  - ✓ 通用同步计算
  - ✓ 系统的跨层次协同设计与优化

目录

回顾：如何支持通用算法

Quick Sort

```
QUICK_SORT(A, p, r)
if p < r
   q = PARTITION(A, p, r)
   QUICK_SORT(A, p, q - 1)
   QUICK_SORT(A, q + 1, r)

PARTITION(A,p,r)
i = p - 1
for j = p to r - 1
   if A[j] < A[r]
      i = i + 1
      exchange A[j] with A[i]
exchange A[i + 1] with A[r]
return i + 1
```

顺序执行，除非跳转

- 赋值
- 算术操作
- 逻辑操作
- 分支跳转
- 直接跳转
- (函数调用)
- 访存操作

DIJKSTRA

```
DIJKSTRA(G, w, s)
S = ∅
Q = G.V
while Q != ∅
   u = EXTRACT-MIN(Q)
   S = S ∪ {u};
   for each v ∈ Q.Adj(u)
      RELAX(u, v, w)

RELAX(u,v,w)
if v.d > u.d + w(u,v)
   v.d = u.d + w(u,v)
   v.π = u
```

算法本质上是几种类型的指令组成的序列

回顾：指令集设计

- 上节课设计的指令

算术指令

`add a0, a1, a2;`	`addi a0, a1, 5;`	`sub a0, a1, a2;`	`subi a0, a1, 5;`
`and a0, a1, a2;`	`andi a0, a1, 5;`	`or a0, a1, a2;`	`ori a0, a1, 5;`
`xor a0, a1, a2;`	`xori a0, a1, 5;`	`sll a0, a1, a2;`	`slli a0, a1, 5;`
`srl a0, a1, a2;`	`srli a0, a1, 5;`	`sra a0, a1, a2;`	`srai a0, a1, 5;`

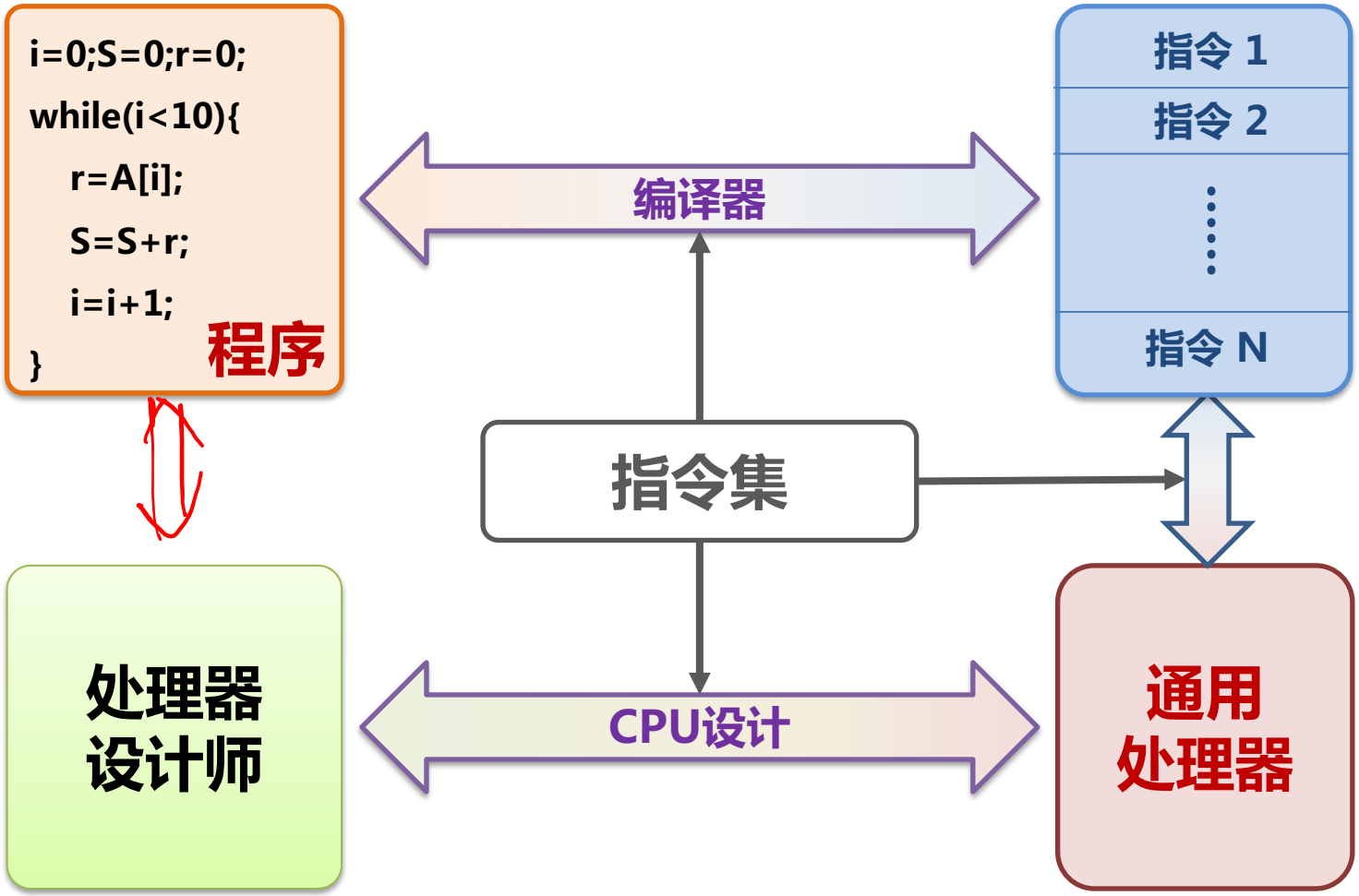
分支跳转指令

`beq a0, a1, label;`	`beqi a0, a1, label;`	`blt a0, a1, label;`	`blti a0, a1, label;`
`bgt a0, a1, label;`	`bgti a0, a1, label;`		

访存指令		**直接跳转指令**	
`lw a0, addr;`	`sw a0, addr;`	`j label;`	`jal label;`

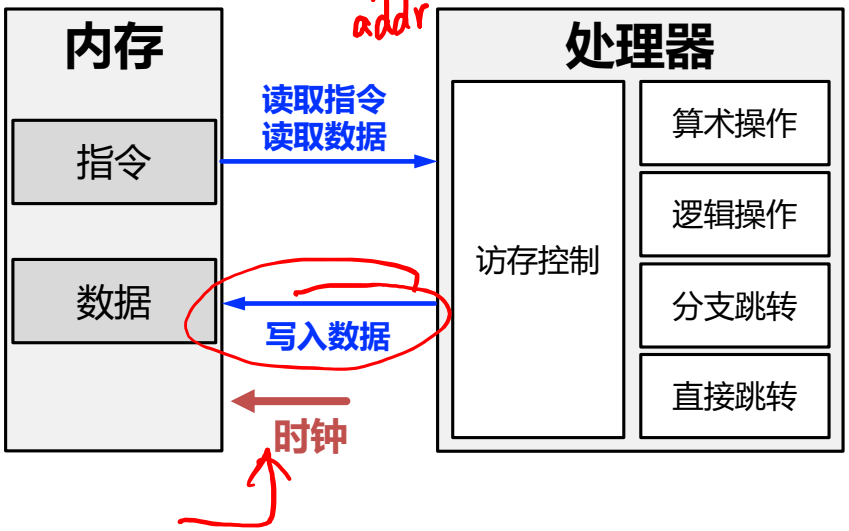
回顾：如何支持通用算法?

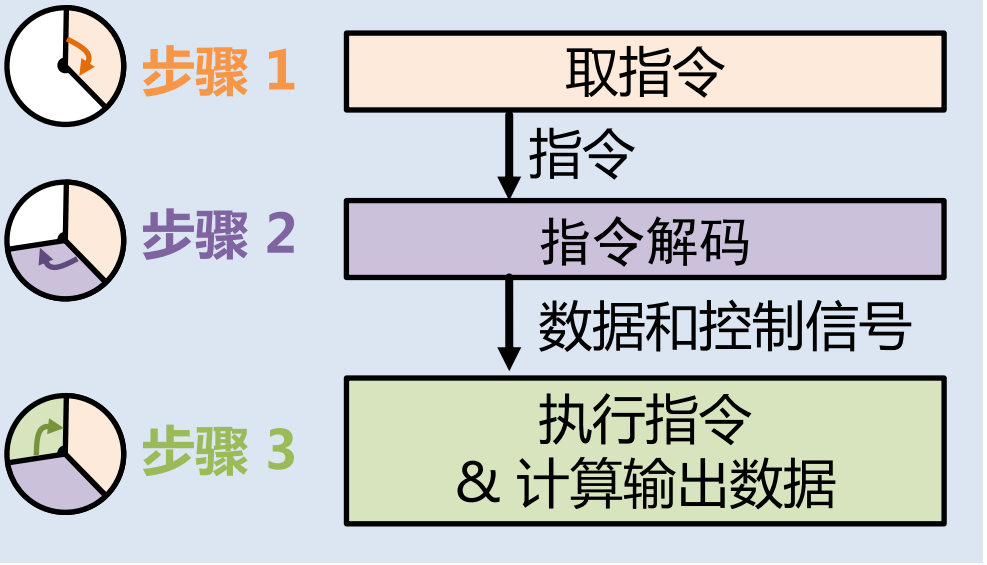
- 指令集的角色
  - ✓ 软件和通用处理器之间的桥梁

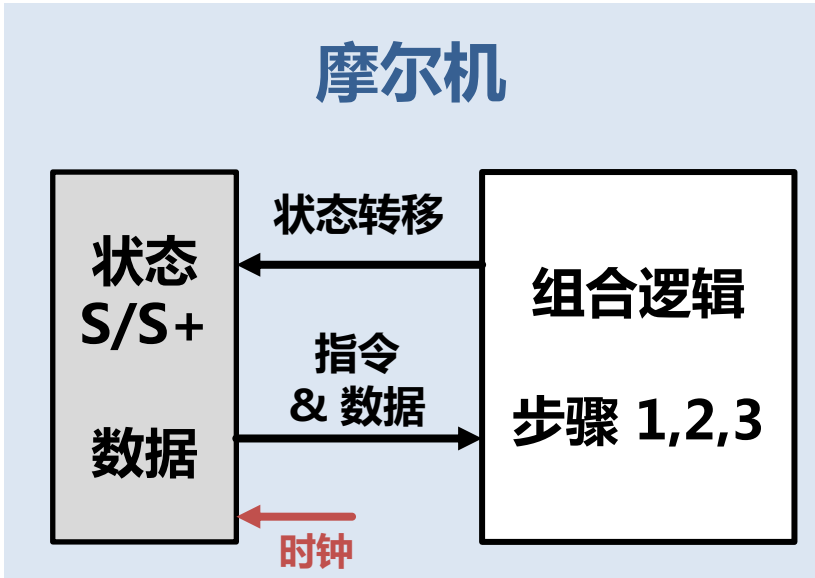


回顾：如何支持通用算法?

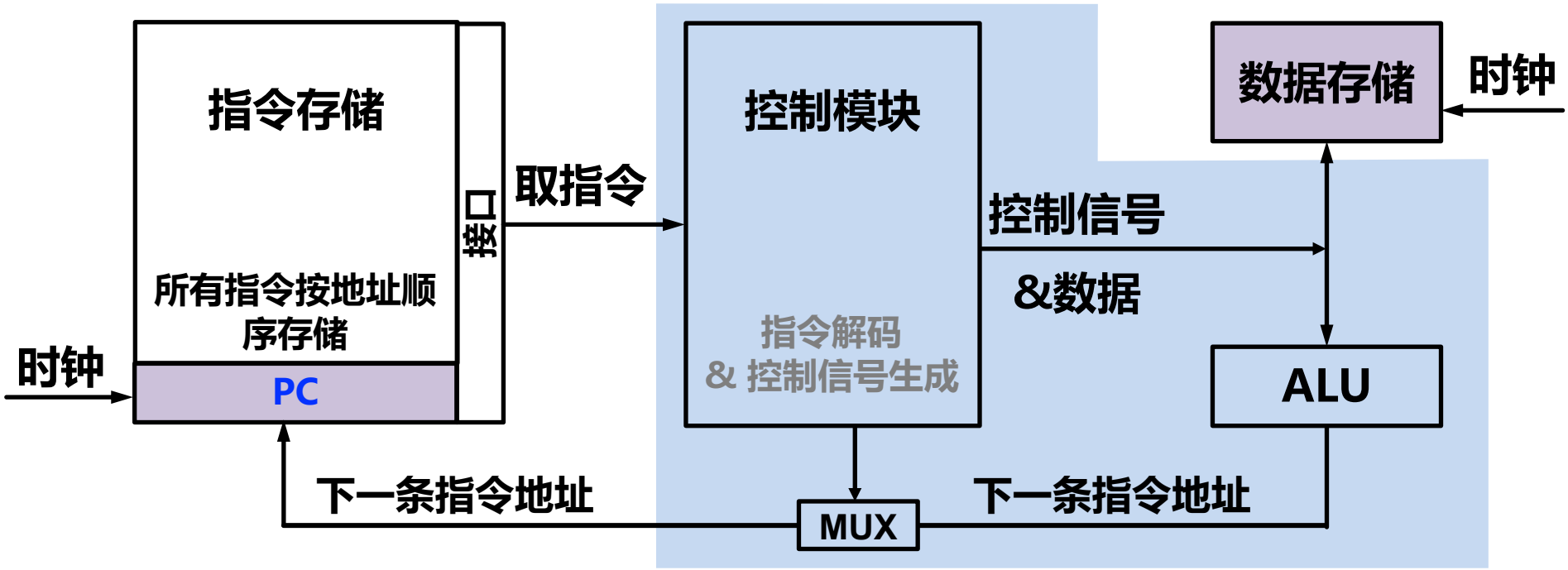
- 如何设计CPU支持ISA?
  - ✓ 明确每条指令的操作步骤
  - ✓ 构造摩尔机
    - 将操作映射到有限状态机





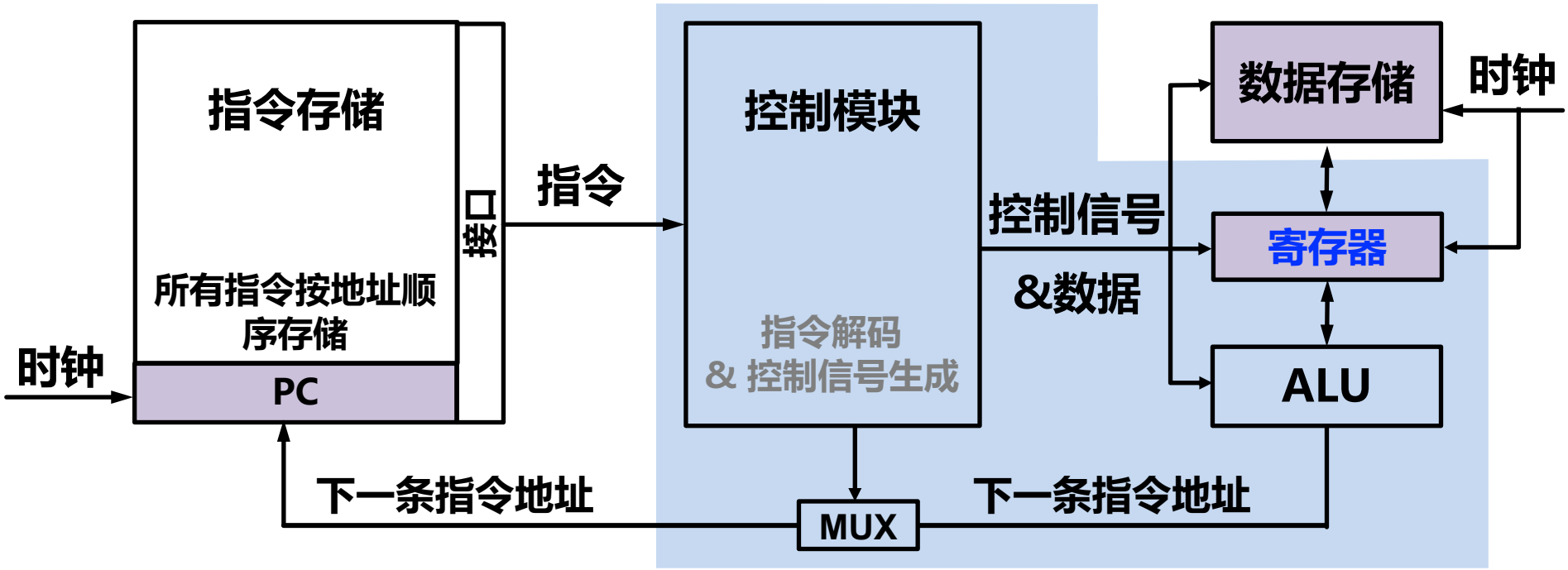


回顾：从专用芯片到处理器



- **PC**
  - ✓ 记录当前执行的指令地址

回顾：从专用芯片到处理器



- **PC**
  - ✓ 记录当前执行的指令地址
- **寄存器**
  - ✓ 将操作数数据存储在CPU中
  - ✓ **只有Load/Store指令可以访问数据存储**

指令集设计：目标

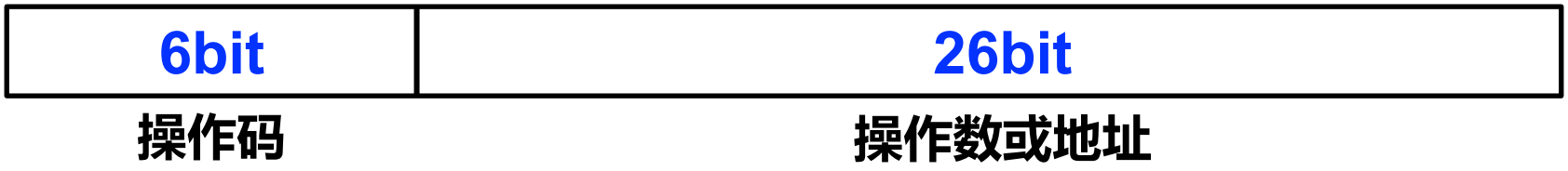
- 目的：更快速、更简单
- 需要包含的信息
  - ✓ 指令的操作
  - ✓ 操作数或其地址
    - 寄存器编号
    - 立即数操作数/地址
      - –算术数值、跳转地址、访存地址
- 预留以后加入新指令需要的空间

操作	操作数或地址

指令集设计：约束

- 最小访存单位（例如8比特）
  - ✓ 单条指令长度是Byte的整数倍（例：32bit）
- 支持一定数量通用寄存器（例如32个）
  - ✓ 例：32个需要用5bit编码
- 简化指令集共26条 ➔ 至少5bit用于表示不同指令
  - ✓ 16条算术指令 + 10条其他指令
  - ✓ 保留一些比特用于未来加入的指令编码

一种可能的编码方案



指令集设计：算术&逻辑指令

指令	功能	指令	功能
add a0, a1, a2;	a0=a1 + a2	addi a0, a1, 5;	a0=a1 + 5
sub a0, a1, a2;	a0=a1 - a2	subi a0, a1, 5;	a0=a1 - 5
and a0, a1, a2;	a0=a1 & a2	andi a0, a1, 5;	a0=a1 & 5
or a0, a1, a2;	a0=a1 | a2	ori a0, a1, 5;	a0=a1 | 5
xor a0, a1, a2;	a0=a1 ^ a2	xori a0, a1, 5;	a0=a1 ^ 5
sll a0, a1, a2;	a0=a1 << a2	slli a0, a1, 5;	a0=a1 << 5
srl a0, a1, a2;	a0=a1 >> a2 （逻辑移位）	srli a0, a1, 5;	a0=a1 >> 5 （逻辑移位）
sra a0, a1, a2;	a0=a1 >> a2 （算术移位）	srai a0, a1, 5;	a0=a1 >> 5 （算术移位）

算术指令编码需要包含的内容：

操作类型、寄存器、立即数

指令集设计：算术&逻辑指令

- `add a0, a1, a2` `# a0 = a1 + a2`
- 需要几位编码寄存器？
  - ✓ 一共32个寄存器 00000 ... 11111
  - ✓ add需要三个寄存器：需要使用15bit

可能的算术&逻辑指令编码方案

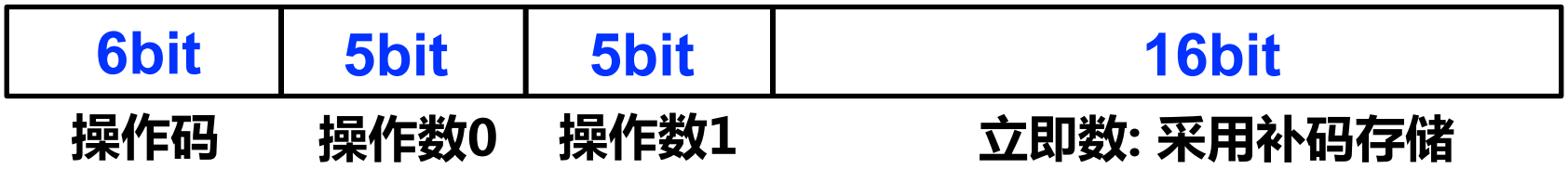
6bit	5bit	5bit	5bit	11bit
操作码	操作数0	操作数1	操作数2	保留

- 问题1：算数&逻辑指令11bit冗余能否利用起来？

指令集设计：算术指令

- `addi a0, a1, 5  # a0 = a1 + 5`
- 如何编码立即数？

可能的算术&逻辑指令编码方案



- 如何处理超过16bit的立即数？
  - ✓ 例：`andi a0, a1, 0x7fffffff`

指令集设计：分支指令

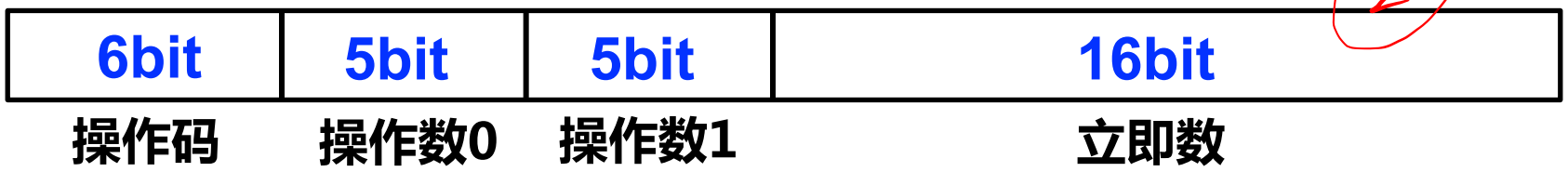
Instruction	Function	Instruction	Function
`beq a0, a1, label;`	Jump to `label` if `a0 == a1`	`beqi a0, 5, label;`	Jump to `label` if `a0 == 5`
`blt a0, a1, label;`	Jump to `label` if `a0 < a1`	`blti a0, 5, label;`	Jump to `label` if `a0 < 5`
`bgt a0, a1, label;`	Jump to `label` if `a0 > a1`	`bgti a0, 5, label;`	Jump to `label` if `a0 > 5`

分支指令编码需要包含的内容：

操作类型、寄存器、地址、立即数

指令集设计：分支指令

❖ *beq a0, a1, label*



❖ 16位能表示几个地址？

- ✓ 常见：指令都占据4个字节（32位），地址按字节编号
- ✓ 指令地址低2位是00，16位编码中可以不包含地址的低2位00

指令集设计：分支指令

- *beq a0, a1, label*
- 问题2：分支可能的地址范围有32位，如何用16bit表示？
- 观察：分支指令大多用于条件、循环等指令
  - ✓ **目标地址接近当前地址**
  - ✓ **目标地址 = 当前地址 + 偏移地址**

2^{16}

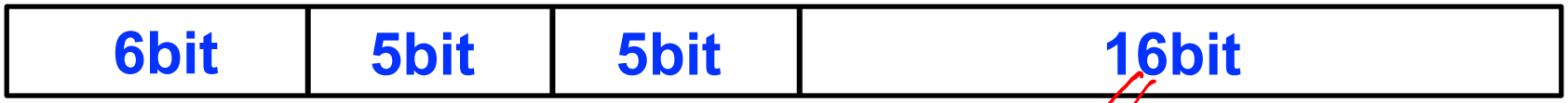
```
QUICK_SORT(A, p, r)
if p < r
  q = PARTITION(A, p, r)
  QUICK_SORT(A, p, q - 1)
  QUICK_SORT(A, q + 1, r)
PARTITION(A,p,r)
i = p - 1
for j = p to r - 1
  if A[j] < A[r]
    i = i + 1
    exchange A[j] with A[i]
exchange A[i + 1] with A[r]
return i + 1
```

```
DIJKSTRA(G, w, s)
S = ∅
Q = G.V
while Q != ∅
  u = EXTRACT-MIN(Q)
  S = S ∪ {u};
  for each v ∈ Q.Adj(u)
    RELAX(u, v, w)
RELAX(u,v,w)
if v.d > u.d + w(u,v)
  v.d = u.d + w(u,v)
  v.π = u
```

指令集设计：分支指令

- *beq a0, a1, label*
- 问题2：分支可能的地址范围有32位，如何用16bit表示？

可能的分支指令编码方案



target = PC_now + {Offset[15:0], 2'b00}

使用当前地址作为基址，实现尽可能大的跳转范围

✓ **问题：如何处理超过16bit的跳转范围？**

指令集设计：分支指令

- `beqi a0, 5, label`
- 问题3：带立即数的分支指令立即数如何编码？
  - ✓ 削减偏移地址位数？

一种可能的编码方案

可表示的范围仍然有限

- ✓ 一条指令不行，改用 *比较 + 分支* 两条指令组合实现？

指令集设计：访存指令

指令	功能	指令	功能
lw a0, addr;	从存储器地址addr取出数据赋值给变量a0	*sw a0, addr;*	将变量a0的数据写入到存储器地址addr

访存指令编码需要包含的内容：

操作类型、寄存器、地址

- 问题4：可能的访存地址有32位，如何用21bit表示？
  - ✓ 基址+偏移地址？
  - ✓ 寄存器存储？

指令集设计：跳转指令

指令	功能
j label	Jump to *label*

跳转指令编码需要包含的内容：
操作类型、地址

- 问题5：跳转可能的地址有32位，如何用26bit表示？

2^{26} 2^{30}

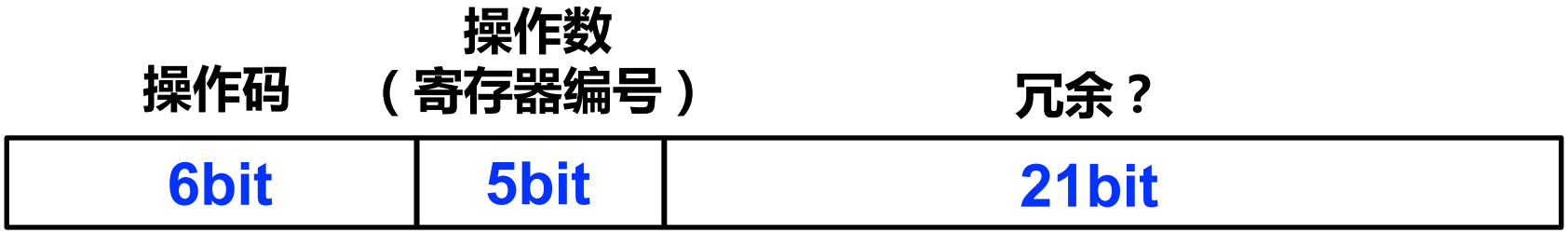
- ✓ 如地址低2位默认是00，可以不需要包含低2位
- ✓ **可能的方案**

指令集设计：跳转指令

- 基址+偏移地址

target = PC_now + {Address[25:0], 2'b00}

- 跳转到寄存器存储的地址（*jr a0*），一条新的指令



target = $a0

指令集设计：跳转指令

- **可能的编码方案**
  - ✓ 当前地址+偏移地址？
  - ✓ 跳转到寄存器中存储的地址？
  - ✓ 其他？
  - ✓ ……

选择哪种方案？

- 需要根据实际情况确定
  - ✓ 选择最适合的一种

指令集设计：总结

- 遇到的问题
  - ✓ 问题1：算数&逻辑指令11bit冗余能否利用起来？
  - ✓ 问题2：分支可能的地址范围有32位，如何用16bit表示？
  - ✓ 问题3：带立即数的分支指令立即数如何编码？
  - ✓ 问题4：可能的访存地址有32位，如何用21bit表示？
  - ✓ 问题5：可能的跳转地址有32位，如何用26bit表示？

编码中的主要矛盾：寻址

如何用有限的比特表示32位地址范围

指令集设计：总结

- 指令集设计是一个不断迭代的过程

软件 ⟷ 指令集 ⟷ 硬件

- 可能的编码方案很多
- 需要结合实际需求、硬件实现确定哪种方案是最优的

目录

- ❖ 指令设计思想与考虑
- ❖ MIPS指令集介绍
  - ✓ 指令存储与数据存储
  - ✓ 指令分类与格式
  - ✓ 寻址方式
  - ✓ 指令系统
  - ✓ 过程调用
- ❖ MIPS指令集程序

一个MIPS指令集架构例子

- 指令长度固定为32-bit
- 32个通用寄存器general-purpose registers
  - ✓ 例：32个寄存器可以用5bit字段表示其序号，但是立即数（常数）往往大于32（即位宽>5）
- MIPS设计折中
  - ✓ 不同指令字段长度不同
  - ✓ 指令前6位为opcode

Op	指令的其他字段
6bit	26bit

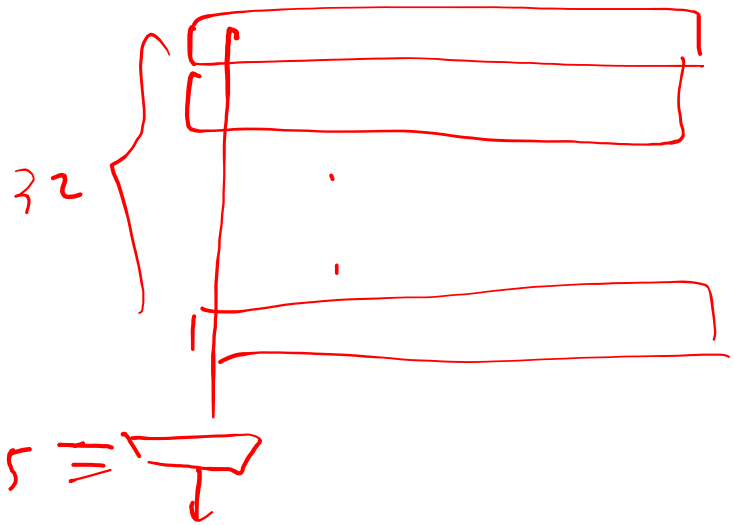
MIPS架构中的数据存储：寄存器

- MIPS架构包含32个32bit的通用寄存器
  - ✓ 使用‘$’+序号字符的方式代表一个寄存器
- 通用寄存器的习惯用法和命名

寄存器编号	助记符	用法
0	*zero*	永远为0
1	*at*	用做汇编器的临时变量
2-3	*v0, v1*	用于过程调用时返回结果
4-7	*a0-a3*	用于过程调用时传递参数
8-15	*t0-t7*	临时寄存器。在过程调用中被调用者不需要保存与恢复
24-25	*t8-t9*	
16-23	*s0-s7*	保存寄存器。在过程调用中被调用者一旦使用这些寄存器时，必须负责保存和恢复这些寄存器的原值
26,27	*k0,k1*	通常被中断或异常处理程序使用，用来保存一些系统参数
28	*gp*	全局指针。一些运行系统维护这个指针来更方便的存取static和extern变量
29	*sp*	堆栈指针
30	*fp*	帧指针
31	*ra*	返回地址

问题

- **MIPS寄存器为什么只有32个？**
  - ✓ 速度：寄存器数量越多，延时越长
  - ✓ 寄存器编号被限制为5bit

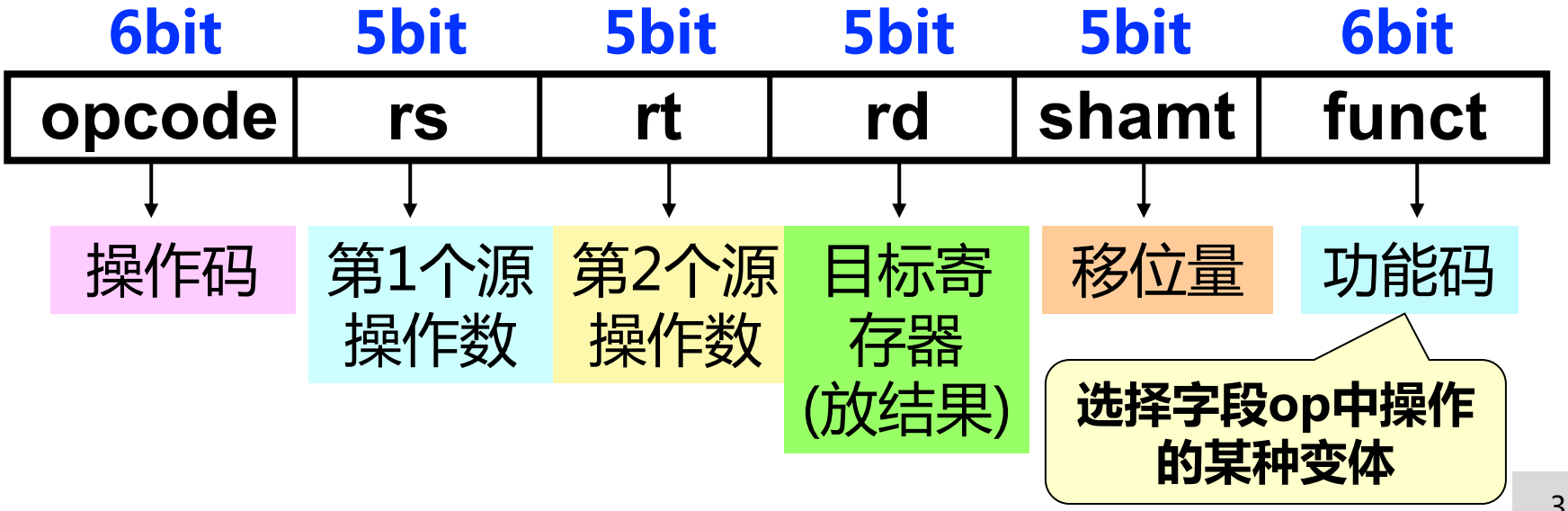


目录

- 指令设计思想与考虑
- MIPS指令集介绍
  - ✓ 指令存储与数据存储
  - ✓ 指令分类与格式
  - ✓ 寻址方式
  - ✓ 指令系统
  - ✓ 过程调用
- MIPS指令集程序

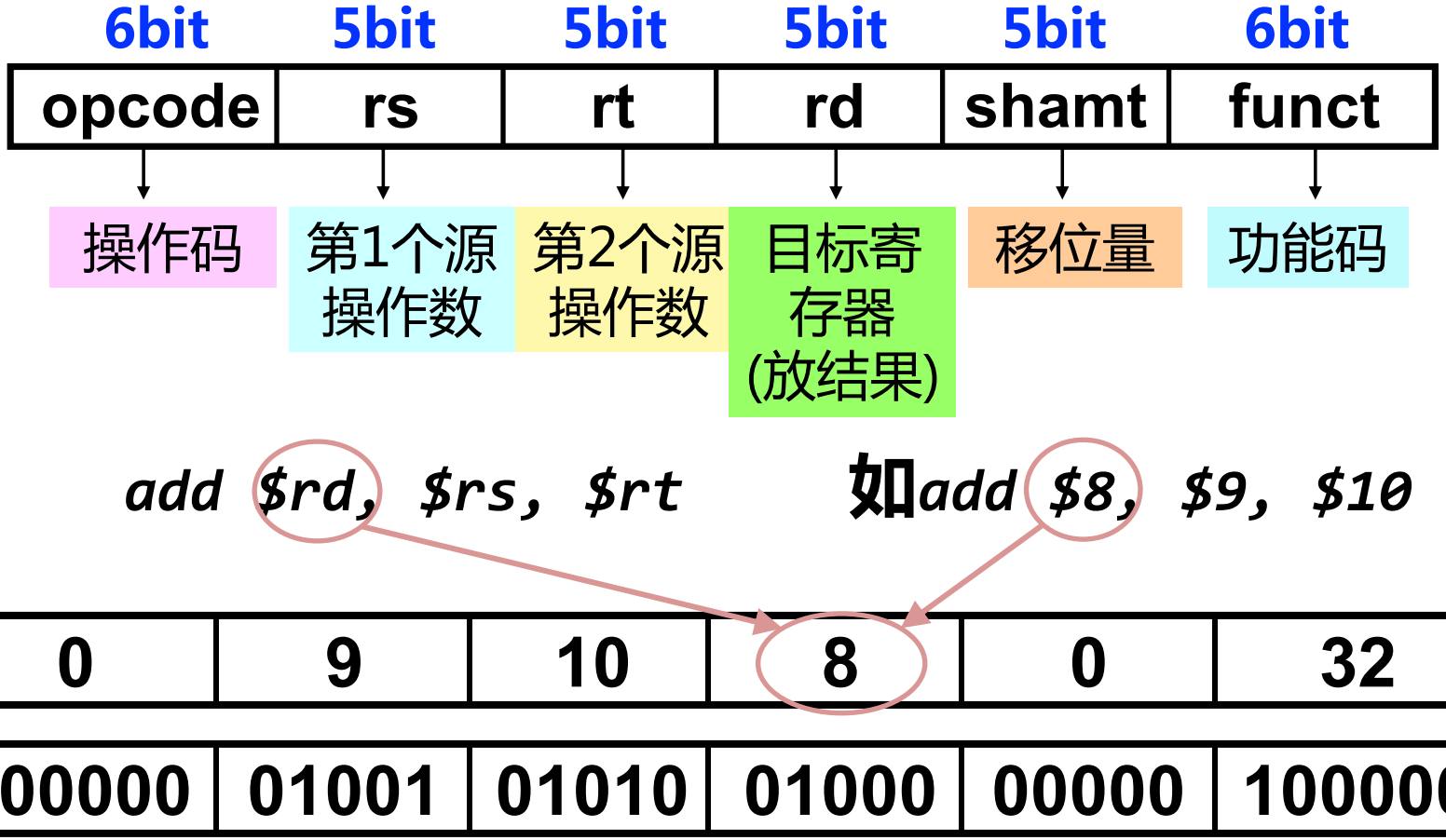
MIPS指令格式：R型指令

- R型指令的字段划分及具体含义如下
  - ✓ 5bit表示寄存器编号
  - ✓ 5bit用于32bit寄存器的移位量（shamt）
  - ✓ 5bit x 3用于rs、rt、rd
    - rs: 1st source register, rt: 2nd register, rd: destination register
  - ✓ opcode与funct共同确定该R型指令的具体功能



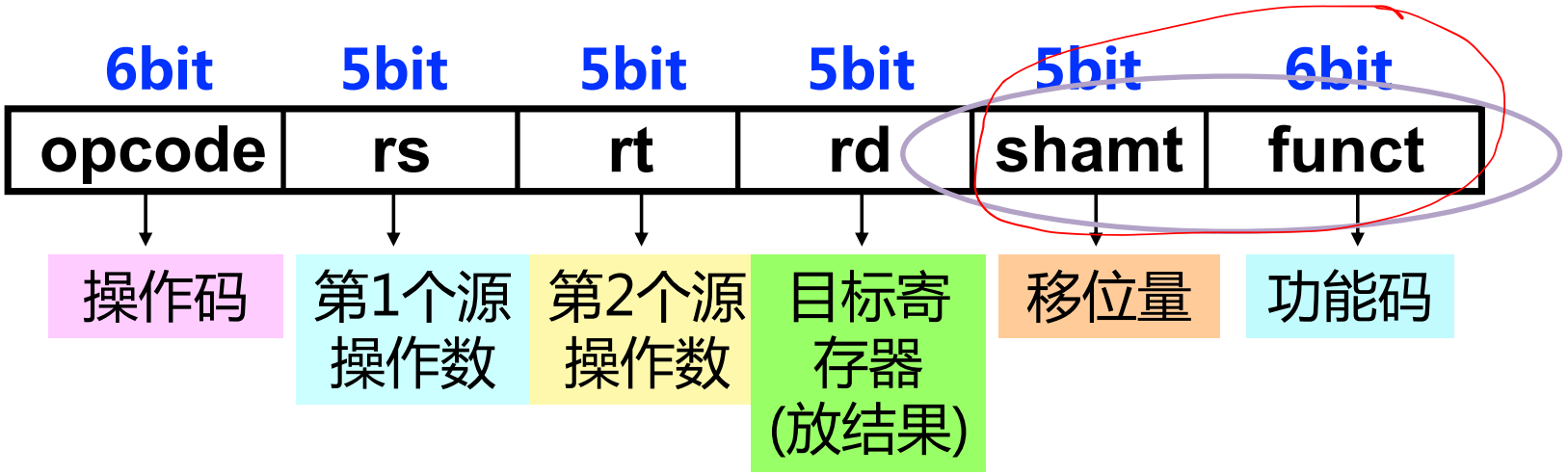
MIPS指令格式：R型指令

- 例子：加法指令



回顾：指令编码

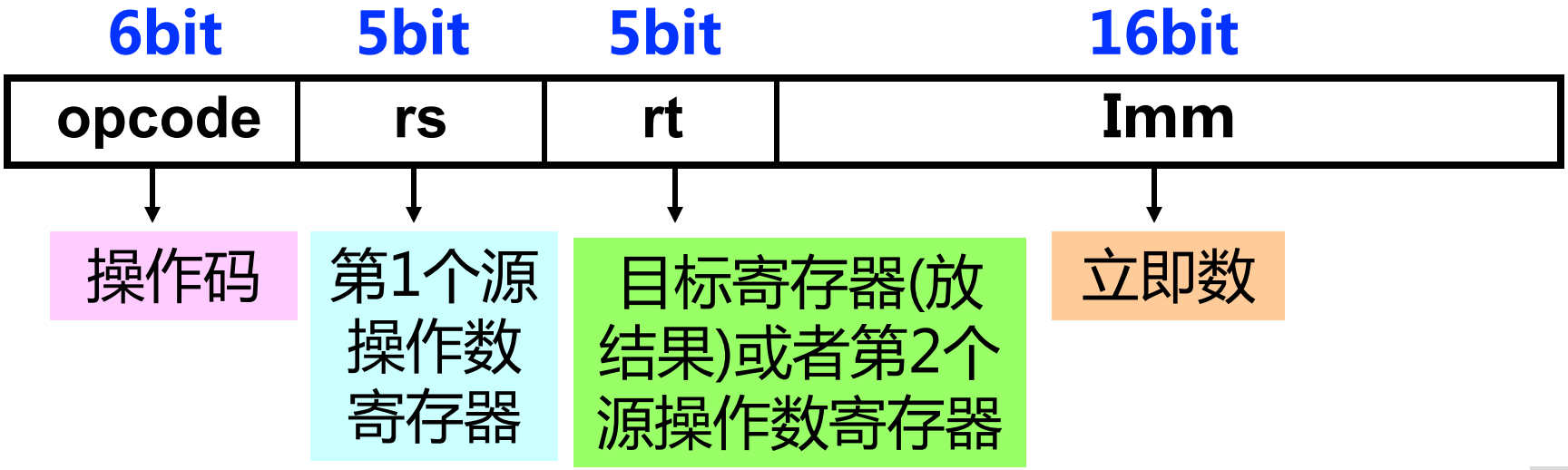
- **问题1：算数&逻辑指令11bit冗余能否利用起来？**



- 额外11bit用于移位量&功能码
  - ✓ 寄存器算术操作只占用一种操作码，指令集可以使用其他操作码支持更多种指令

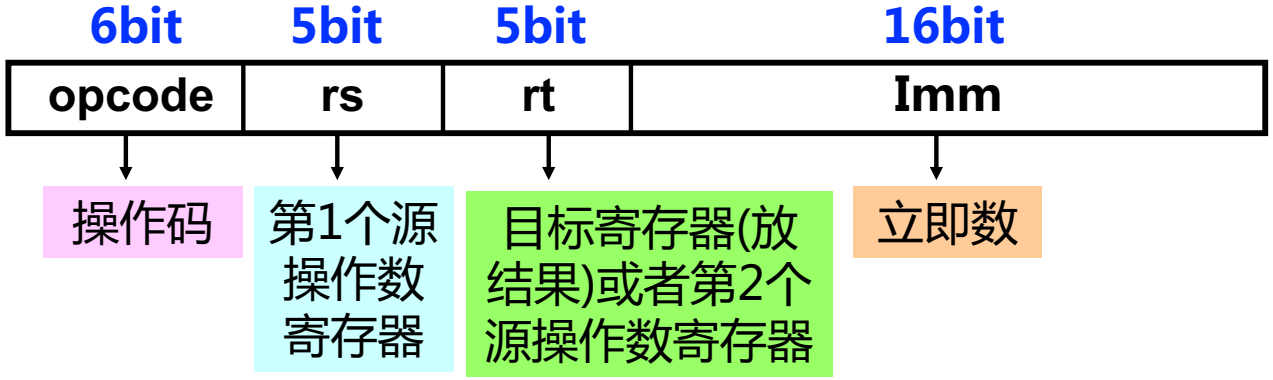
MIPS指令格式：I型指令

- I型指令的字段划分及具体含义如下
  - ✓ 5bit表示寄存器
    - rt作为源操作数或目标寄存器
  - ✓ 立即数
    - 可表示16bit常数，也可表示地址偏移量



MIPS指令格式：I型指令

❖ 例子1：*addi*　A=B+50。



addi $21, $22, 50　　**50需转换为二进制机器码**

源操作数寄存器
第22号寄存器

立即数源操作数50（十进制表示）

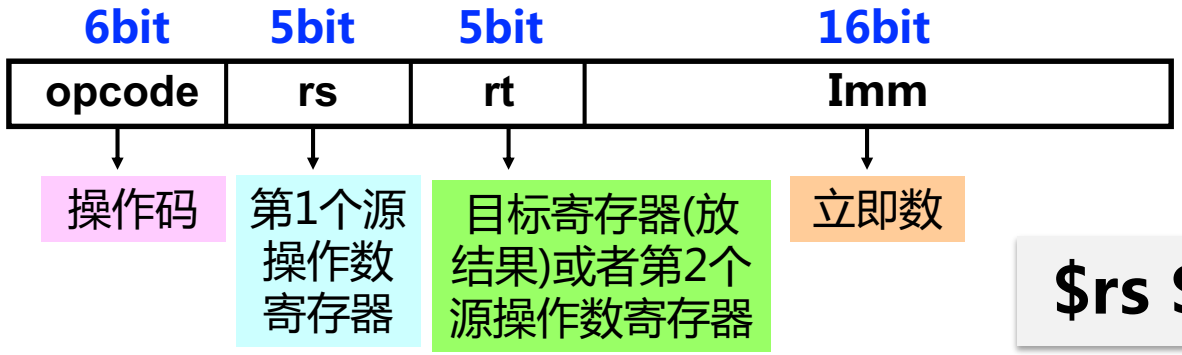
opcode = 8
指定是立即数加操作

目的寄存器
第21号寄存器

只有$rs为源寄存器

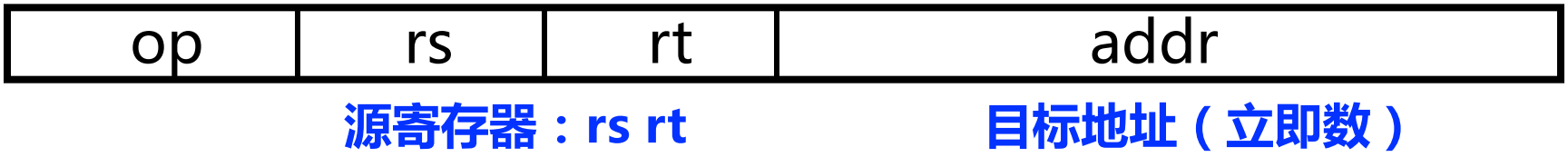
MIPS指令格式：I型指令

- 例子2：分支指令，如 if (A==B) goto addr
  - ✓ 目标地址：PC + 4 + (Addr << 2)
    - Addr << 2（直接地址 ➔ 字节地址）
    - PC+4（每条指令4字节）：不发生分支跳转的下一条指令



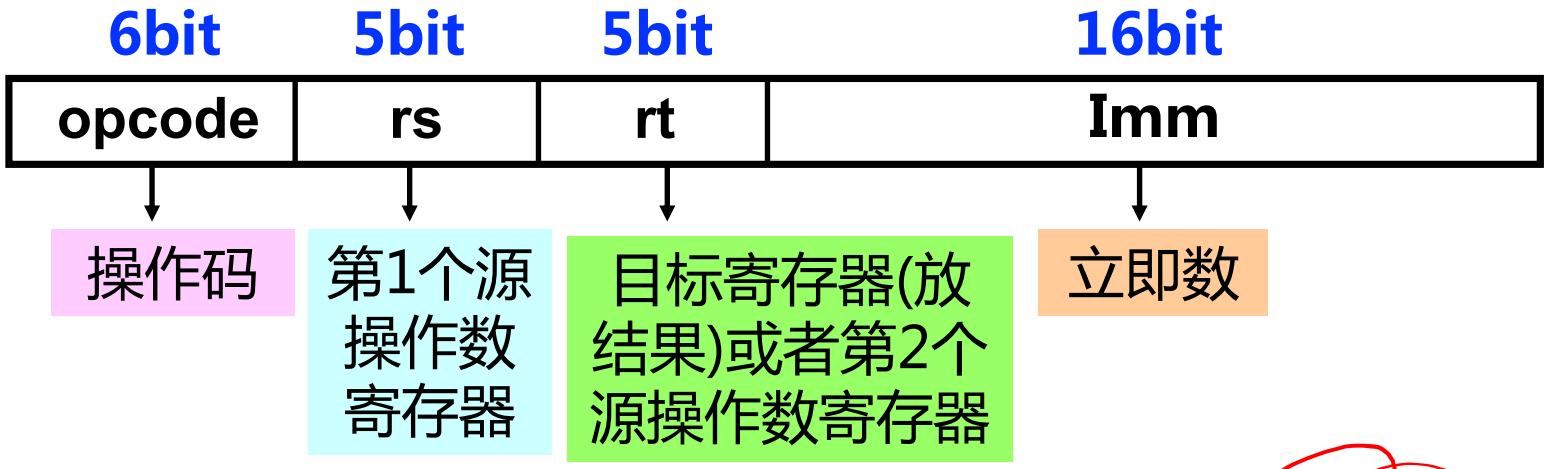
$rs $rt均为源寄存器

beq $rs, $rt, Address



回顾：指令编码

- **问题2：分支可能的地址范围有32位，如何用16bit表示？**



target = PC_now + {Offset[15:0], 2'b00}

- MIPS采用基址+偏移地址的寻址方式
  - ✓ 更大范围采用跳转指令J/Jr实现

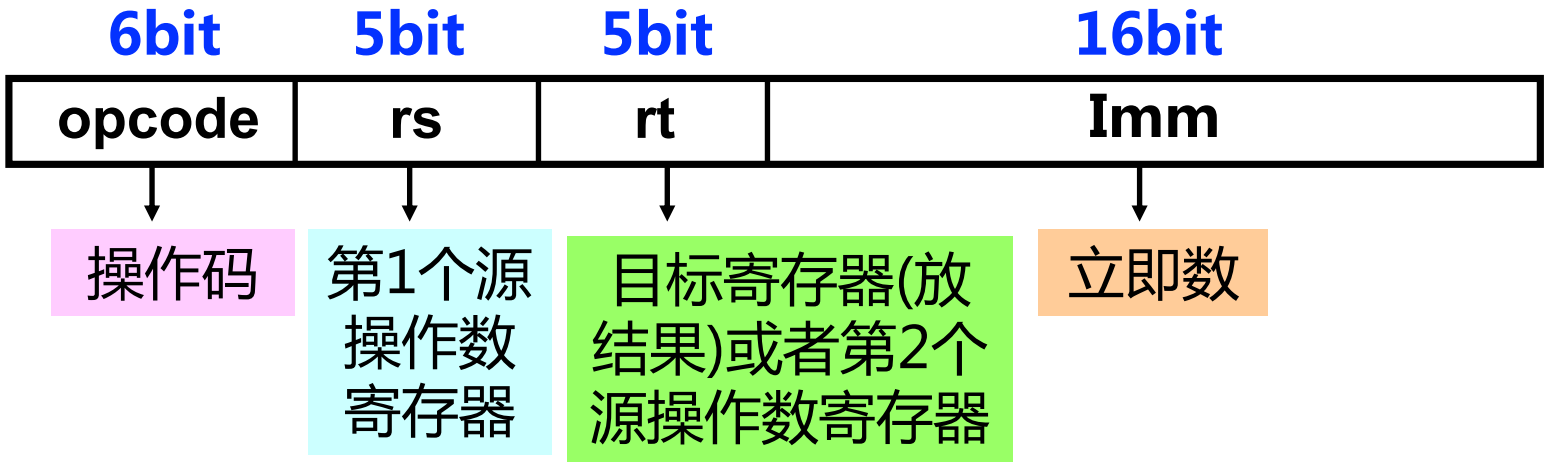
MIPS指令格式：I型指令

- 一些分支指令的例子
  - ✓ beq rs, rt, target　#go to target if rs = rt
  - ✓ bne rs, rt, target　#go to target if rs != rt
  - ✓ beqz rs, target　#go to target if rs = 0
  - ✓ bltz rs, target　#go to target if rs < 0

回顾：指令编码

if($rs==5) jump

❖ **问题3：带立即数的分支指令立即数如何编码？**

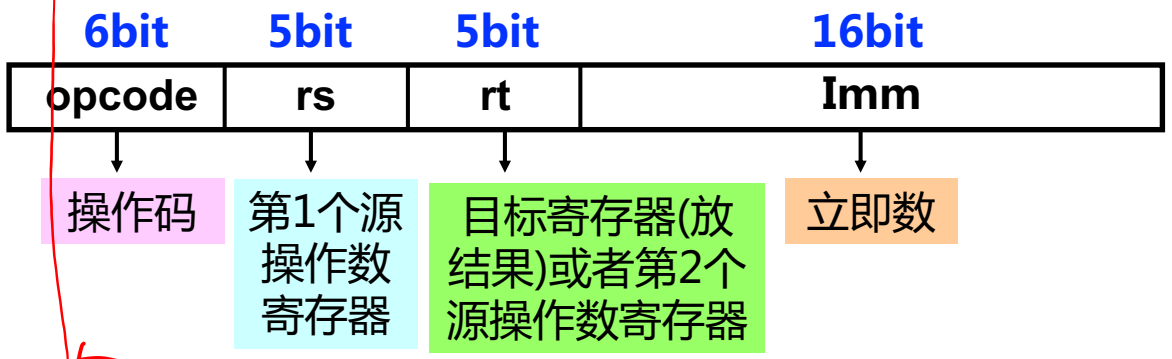


❖ MIPS没有带立即数的分支指令

✓ 使用比较指令(*slti*、*sltiu*等）+ 分支指令组合实现

MIPS指令格式：I型指令

- 例子3：装入/存储指令 load word/store word，如 A=B[25]
  - ✓ sw/lw $rt, immediate($rs)



lw $s1, 100($s2) **二进制码：** 100_{10}

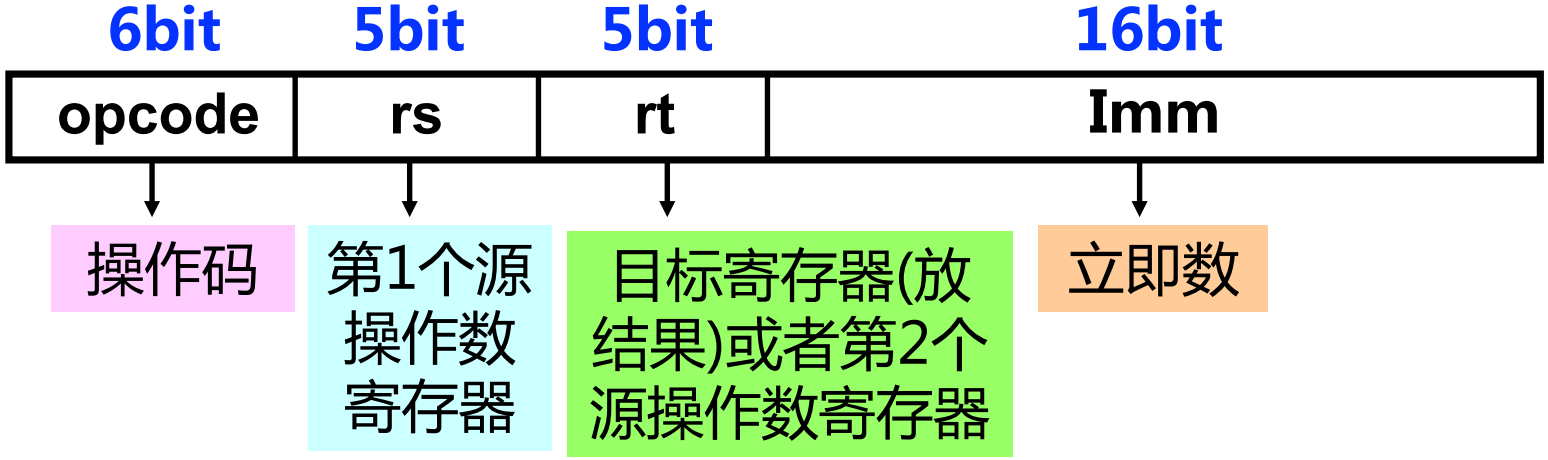
100011	10010	10001	0000 0000 0110 0100

- 数据装入：Rt = Mem[Rs + Address]
- 数据存储：Mem[Rs + Address] = Rt

只有$rs为源寄存器

回顾：指令编码

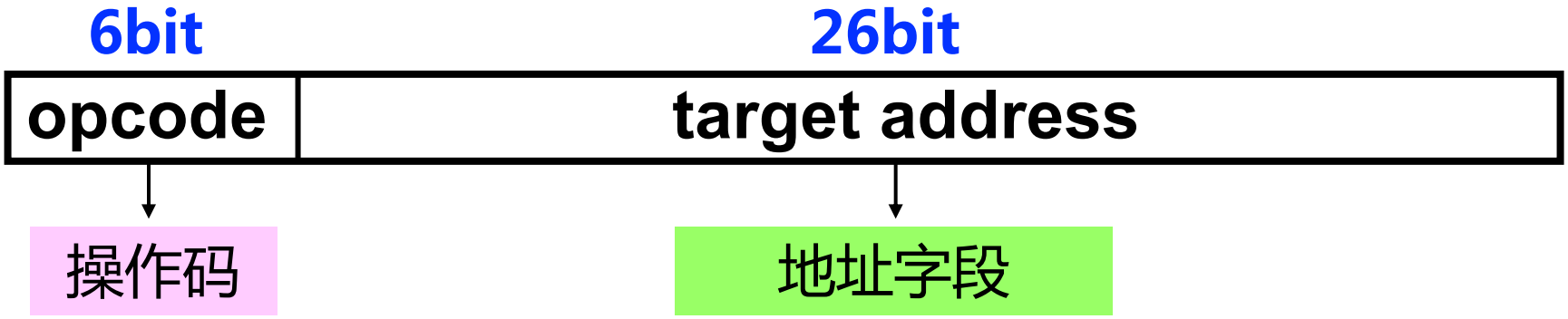
- **问题4：访存可能的地址有32位，如何用21bit表示？**
  - ✓ 数据装入：Rt = Mem[Rs + Address]
  - ✓ 数据存储：Mem[Rs + Address] = Rt



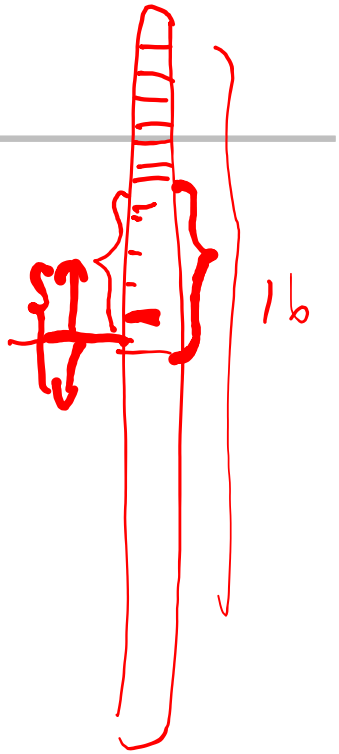
- MIPS采用基址+偏移地址的寻址方式进行访存

MIPS指令格式：J型指令

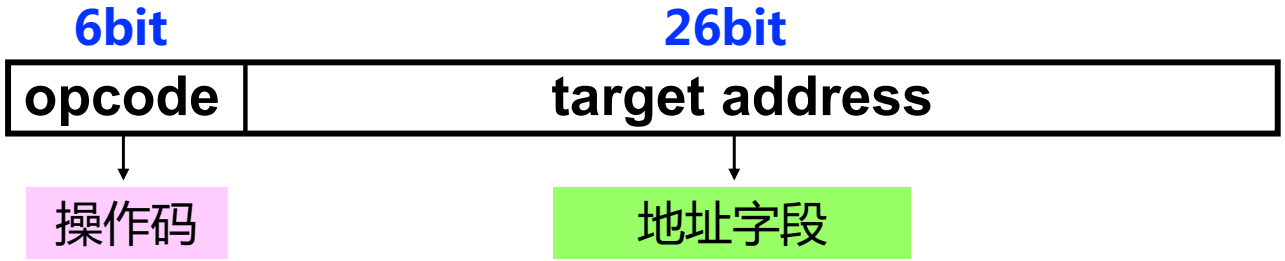
- J型指令的字段划分及具体含义如下
  - ✓ 26-bit目标地址



MIPS指令格式：J型指令



- 例子：跳转指令，如go to location 10000_{10}
  - ✓ 伪直接寻址固定PC的高4位不变（PC[31:28]）
    - PC_new = {PC[31:28], target_addr, 00}, 32bit
  - ✓ {target_addr, 00}作为低28位目标地址
    - {target_addr, 00} = target_addr << 2



j 10000$_{10}$

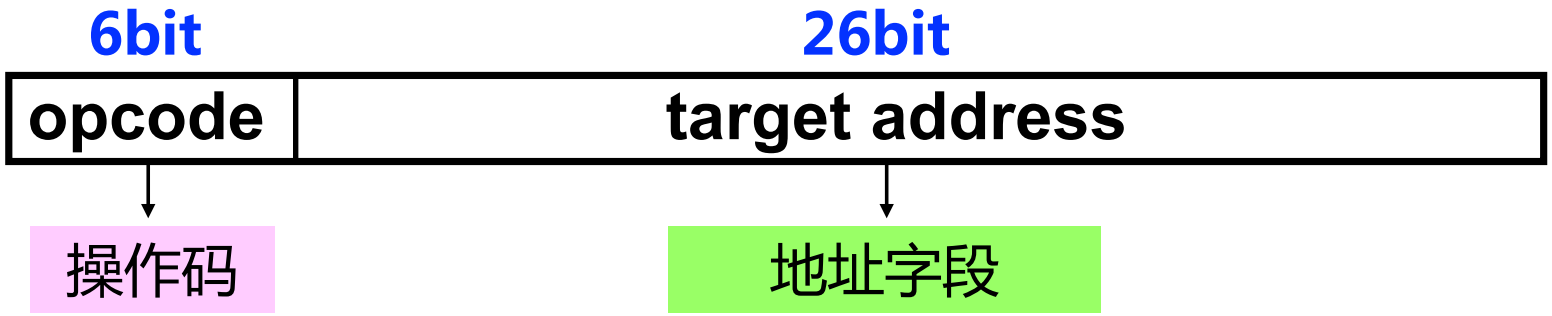
10000（十进制）

000010	00000000000010011100010000

二进制表示（机器码）

回顾：指令编码

- **问题5：跳转可能的地址有32位，如何用26bit表示？**



- MIPS中，jump操作包括 *j* 和 *jr*
  - ✓ 使用 *j* 指令（直接跳转）进行有限范围的跳转
  - ✓ 使用 *jr* 指令（跳转到寄存器存储的地址）进行大范围跳转

指令格式：总结

- opcode/funct共同定义指令的操作
- I型指令（带立即数）：*lw sw beq bne*
  - ✓ *lw/sw*: load word; store word
  - ✓ *beq/bne*: branch on equal; branch on not equal
  - ✓ opcode: 除000000, 00001x, 0100xx外
- J型指令（用于跳转）：*j jal*
  - ✓ j: jump; jal: jump and link
  - ✓ opcode: 000010, 000011
- R型指令（所有其他指令，包括*jr*）
  - ✓ opcode: 000000

opcode: 区分不同指令的关键

Quiz1 判断题

- I型指令不但可以用立即数参与计算，还可以实现指令跳转（是/否）
- I型指令进行跳转时无法跳转到比当前指令地址小的指令（是/否）

Quiz1 判断题

- I型指令不但可以用立即数参与计算，还可以实现指令跳转（是，如beq等条件跳转指令）
- I型指令进行跳转时无法跳转到比当前指令地址小的指令（否，相对寻址是有符号数操作）

目录

- 指令设计思想与考虑
- MIPS指令集介绍
  - ✓ 指令存储与数据存储
  - ✓ 指令分类与格式
  - ✓ 寻址方式
  - ✓ 指令系统
  - ✓ 过程调用
- MIPS指令集程序

寻址方式

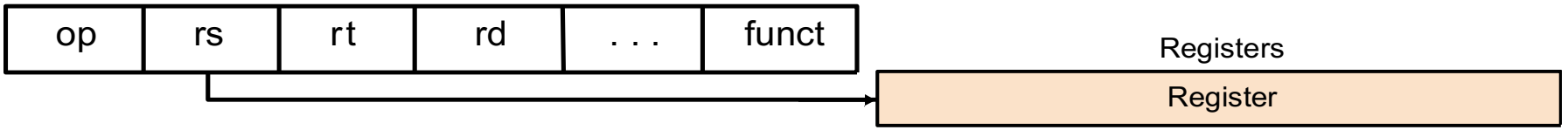
- 如何计算数据和指令的地址？
- MIPS有5种寻址模式
  - ✓ 寄存器寻址
  - ✓ 立即数寻址
  - ✓ 基址（基址-偏移）寻址
  - ✓ PC相对寻址
  - ✓ 伪直接寻址

寄存器寻址

- 找到对应的寄存器，从寄存器中取数/写数
  - ✓ MIPS算术运算指令的操作数可从32个32位寄存器中选取
- 例：计算 (g + h) – (i + j)

```
add $ t0, $s1, $s2    # 寄存器$t0的内容为 g+h
add $ t1, $s3, $s4    # 寄存器$t1的内容为 i+j
sub $ s0, $t0, $t1    # 寄存器$s0的内容为 (g+h)-(i+j)
```

2. Register addressing



立即数寻址

- “访存”指令中的立即数可以被直接使用
- 对于常数操作数，在指令中加入立即数字段，比存储器访问快得多

a=a+4

```
lw $t0, addr($zero)
# 假设 $zero + addr 是常数4的存储地址*
add $ sp, $sp, $t0
# $sp = $sp + $t0 ($t0 == 4)
```

从存储器读出4

慢

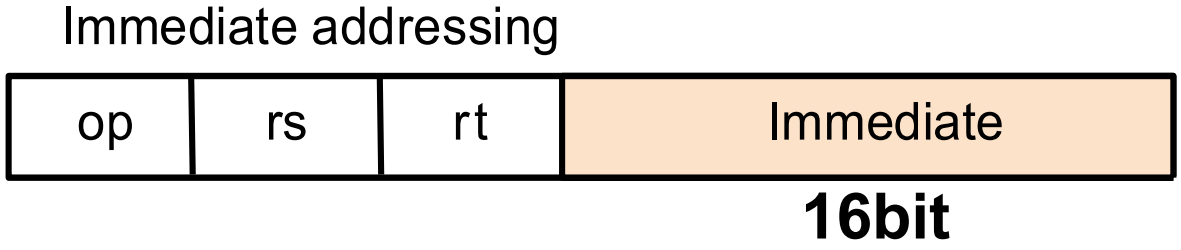
```
addi $ sp, $sp, 4
# $sp = $sp + 4
```

使用I型指令

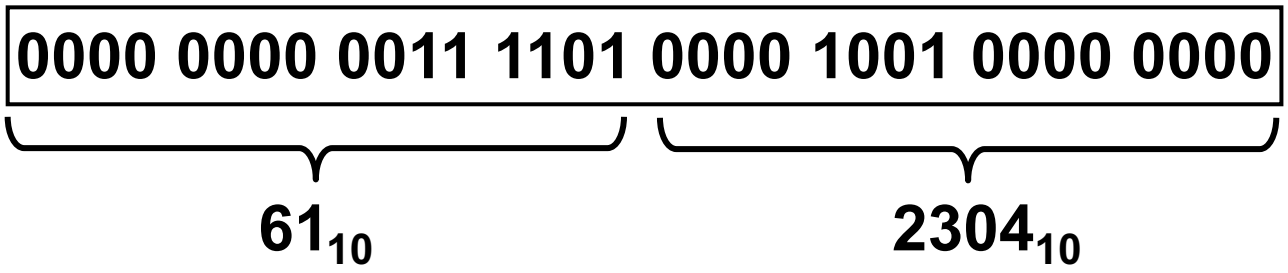
快

立即数寻址

- I型指令中Imm字段只有16bit
  - ✓ 如何将一个32bit的常数装入到寄存器$s0中呢?

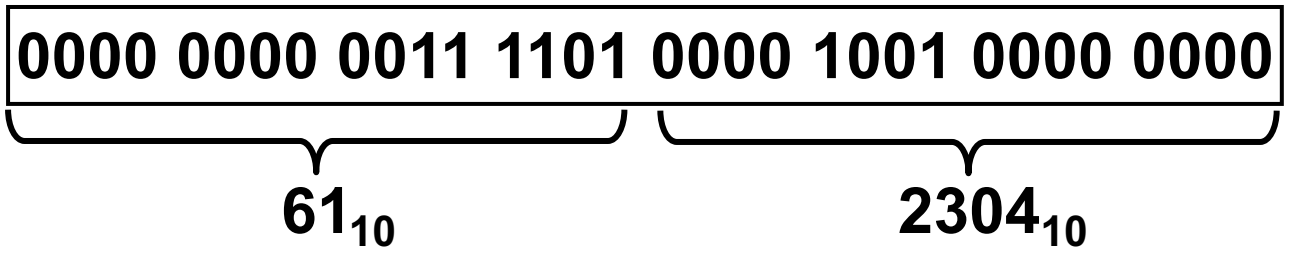


- 例如32bit数：



立即数寻址

- I型指令中Imm字段只有16bit
  - ✓ 如何将一个32bit的常数装入到寄存器$s0中呢?
- 例如32bit数：



Load Upper Immediate 先加载常数的高16bit到寄存器的高16位

```
Lui  $s0, 61      # $ s0= 0000 0000 0011 1101 0000 0000 0000 0000
addi $s0, $s0, 2304  # $ s0= 0000 0000 0011 1101 0000 1001 0000 0000
```

再与低16位相加

或者：

```
ori $s5,$0,0x1000 #or immediate $s5=0x10000000
sll $s5,$s5,16    #shift logic left, 提取高16位后左移
```

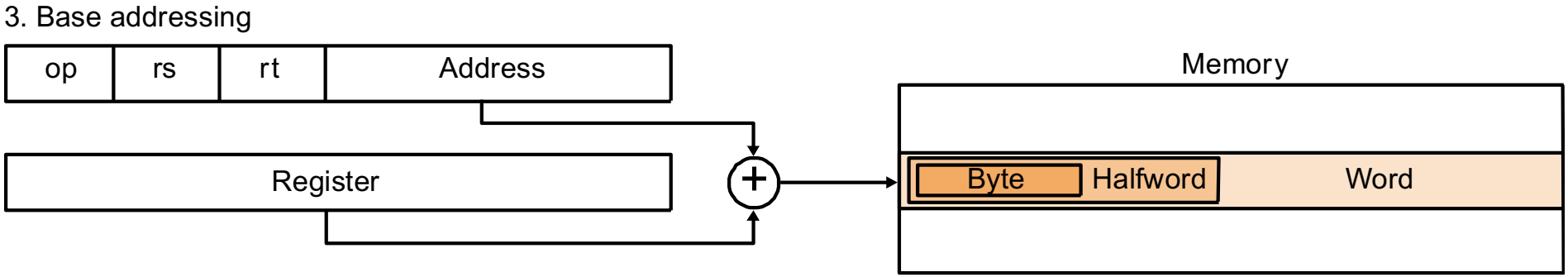
基址-偏移寻址

- 目标地址=基址（存储于寄存器中）+ 立即数

8:偏移量　$s0:基址

```
lw $t0, 8($s0)
```

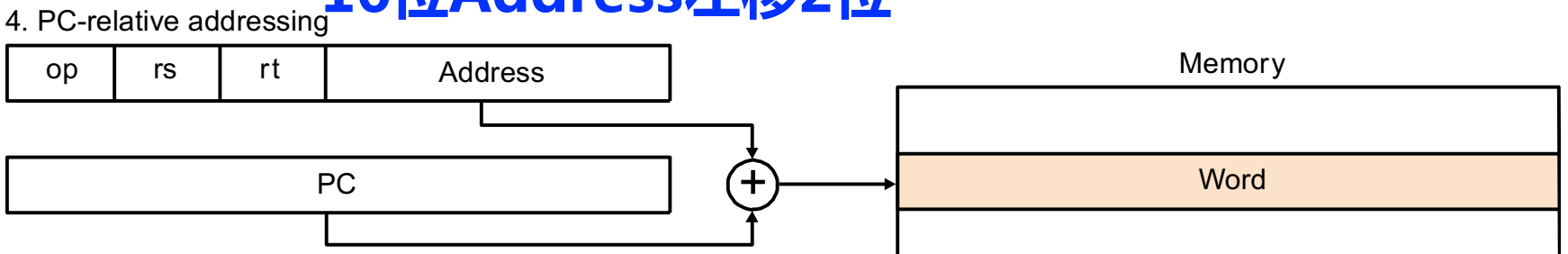
#$s0中装的是存储器中的地址



PC相对寻址

- PC + 立即数
- 通常相对于下一条指令的地址PC+4（单位：**字节**）
- 例如分支指令：16bit立即数字段（单位：**字**），可以访问$\pm 2^{15}$（单位：**字**）的地址范围
- 分支的目标地址：PC_new = PC + 4 + (addr<<2)

16位Address左移2位



PC记录下一条指令的地址（即PC+4）

PC相对寻址

- 为什么选PC寄存器做相对寻址？
  - ✓ 目标地址通常位于当前指令地址附近（接近PC的值）
- 如何处理长距离分支（ > 16bits ）？
  - ✓ 插入一条 *j* 指令跳到分支目标地址

```
bne $s0, $s1, l2
```

If *$s0*!= *$s1*, jump to *l2*

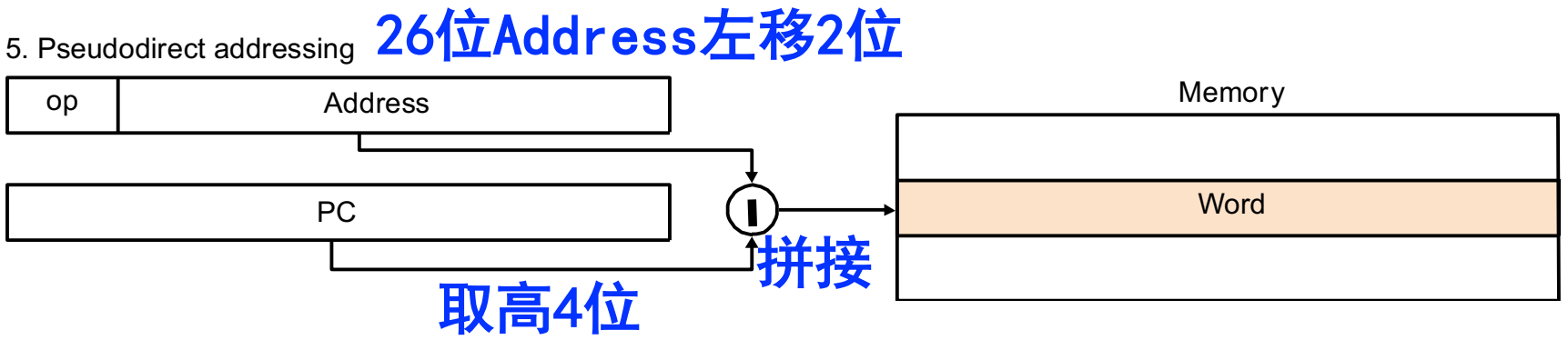
```
  beq $s0, $s1, l1
  j   l2
l1:
        ……
l2:
        ……
```

If *$s0*== *$s1*, jump to *l1* (short distance)

If *$s0*!= *$s1*, *j l2* (long distance)

伪直接寻址

- 伪直接寻址固定PC高4位不变（PC[31:28] or (PC+4)[31:28]）
  - ✓ PC_new = {PC[31:28], target_addr, 00}, 32bit
- {target_addr, 00}作为低28位目标地址
  - ✓ {target_addr, 00} = target_addr << 2



MIPS寻址方式总结

- 寄存器寻址
  - ✓ 找到对应的寄存器，从寄存器中取数/写数
- 立即数寻址
  - ✓ 访存中的立即数可以被直接使用
  - ✓ 对于常量操作数，使用立即数字段比访存更加快速
- 基址寻址
  - ✓ 目标地址=基址（存储于寄存器中）+ 立即数
- PC相对寻址： PC_new = PC+4 + (Addr<<2)
- 伪直接寻址： PC_new = {PC[31:28], target_addr, 00}

Quiz2 单选题

❖ 下列指令的寻址方式是：

1、*add*　2、*addi*　3、*lw*和*sw*

4、*beq*　5、*j*　6、*jr*

a. 寄存器寻址　b. 立即数寻址 c. 基址或偏移寻址

d. PC相对寻址　e. 伪直接寻址

A. 1.a 2.ac 3.ab 4.ad 5.e 6.a

B. 1.a 2.ac 3.ac 4.ad 5.a 6.e

C. 1.a 2.ab 3.ac 4.ad 5.e 6.a

D. 1.a 2.ab 3.ab 4.ac 5.a 6.e

Quiz2 单选题

❖ 下列指令的寻址方式是：

1、*add*　2、*addi*　3、*lw*和*sw*

4、*beq*　5、*j*　6、*jr*

a. 寄存器寻址　b. 立即数寻址 c. 基址或偏移寻址

d. PC相对寻址　e. 伪直接寻址

A. 1.a 2.ac 3.ab 4.ad 5.e 6.a

B. 1.a 2.ac 3.ac 4.ad 5.a 6.e

C. 1.a 2.ab 3.ac 4.ad 5.e 6.a

D. 1.a 2.ab 3.ab 4.ac 5.a 6.e

目录

- 指令设计思想与考虑
- MIPS指令集介绍
  - ✓ 指令存储与数据存储
  - ✓ 指令分类与格式
  - ✓ 寻址方式
  - ✓ 指令系统
  - ✓ 过程调用
- MIPS指令集程序

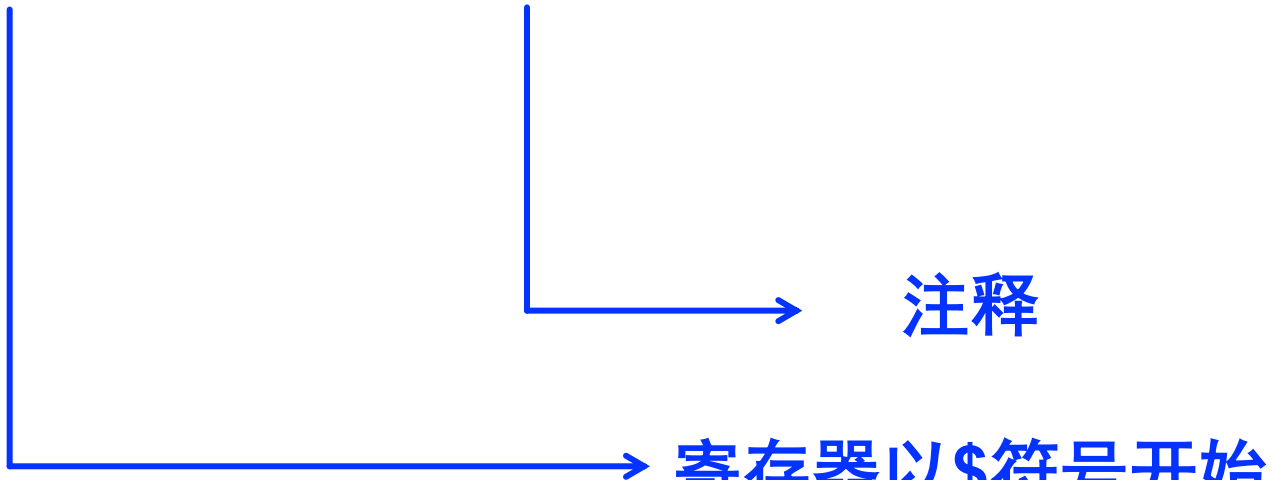
算术运算

- 加法

```
add $t0,$t1,$t2     # $t0 = $t1 + $t2
```

- 减法

```
sub $t2,$t3,$t4     # $t2 = $t3 - $t4
```



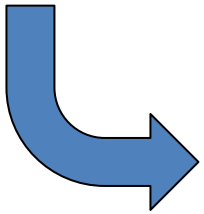
算术运算

- 编译C语言的赋值语句

$$f = (g + h) - (i + j)$$

```
add t1, g, h
add t2, i, j
sub f, t1, t2
```

假设变量 f、g、h、i、j 分别分配给寄存器 $s0~$s4 临时变量 t1、t2 分别分配给寄存器 $t1、$t2



```
add $t1, $s1, $s2
add $t2, $s3, $s4
sub $s0, $t1, $t2
```

算术运算

- 加16位有符号立即数

```
addi $t2, $t3, 5   # $t2 = $t3 + 5
```

- MIPS没有减立即数的指令
  - ✓ 如何实现？

使用2-补码实现立即数减！

逻辑运算

```
and   $t0,$t1,$t2          # $t0 = $t1 & $t2
or    $t0,$t1,$t2          # $t0 = $t1 | $t2
xor   $t0,$t1,$t2          # $t0 = $t1 ⊕ $t2
nor   $t0,$t1,$t2          # $t0 = ~($t1 | $t2)
```

✓如何实现NOT运算？

使用NOR指令，其中一个寄存器值为0

逻辑运算

- 带立即数的逻辑运算

```
andi $t0,$t1,10
ori $t0,$t1,10
xori $t0,$t1,10
```

移位操作

- 移位量为立即数

  逻辑左移 —— `sll $t0, $t1, 10`

  `# $t0 = $t1 << 10 , shift left logical`

  逻辑右移 —— `srl $t0, $t1, 10`

  `# $t0 = $t1 >> 10 , shift right logical`

  算术右移 —— `sra $t0, $t1, 10`

  `# $t0 = $t1 >> 10 , shift right arithmetic`

- 移位量在某个寄存器中

```
sllv $t0, $t1, $t3    # $t0 = $t1 << ($t3%32)
srlv $t0, $t1, $t3    # $t0 = $t1 >> ($t3%32)
srav $t0, $t1, $t3    # $t0 = $t1 >> ($t3%32)
```

比较指令

- 比较两个寄存器的内容，并根据比较的结果设置第三个寄存器

```
slt $t1,$t2,$t3
# if ($t2 < $t3) $t1=1;
# else $t1=0

sltu $t1,$t2,$t3     # 无符号比较
```

- 寄存器与立即数比较

```
slti $t1, $t2, 10    # 与立即数比较
sltui $t1, $t2, 10   # 与无符号立即数比较
```

关于xxx/xxxi/xxxu/xxxiu

- xxx表示原指令
- xxxi表示立即数指令
- xxxu表示无符号指
- xxxiu表示无符号立即数指令

- **针对不同类型的指令，对应的细节不同**
- **详见课程教材中的描述**

注意处理方式的差别！

装入/存储指令

❖ 实现内存与寄存器之间的数据传送

```
lw $t1, 30($t2)          # Load word
sw $t3, 500($t4)         # Store word
```

- ✓ 基址(变址)+偏移量的内存寻址方式
- ✓ 存放基址的寄存器称为基址寄存器
- ✓ 指令中的常量称为偏移量

分支(branch)指令

```
beq $t0, $t1, target    #如果 $t0=$t1，则分支执
                        #行标号为target的指令

bne $t0, $t1, target    #如果 $t0!=$t1，则分支执
                        #行标号为target的指令
```

- 分支指令与比较指令相结合，可以实现各种条件分支：相等、不等、小于、小于或等于、大于、大于或等于
  - ✓ 课后思考：写出这些指令

跳转(jump)指令

- 无条件分支

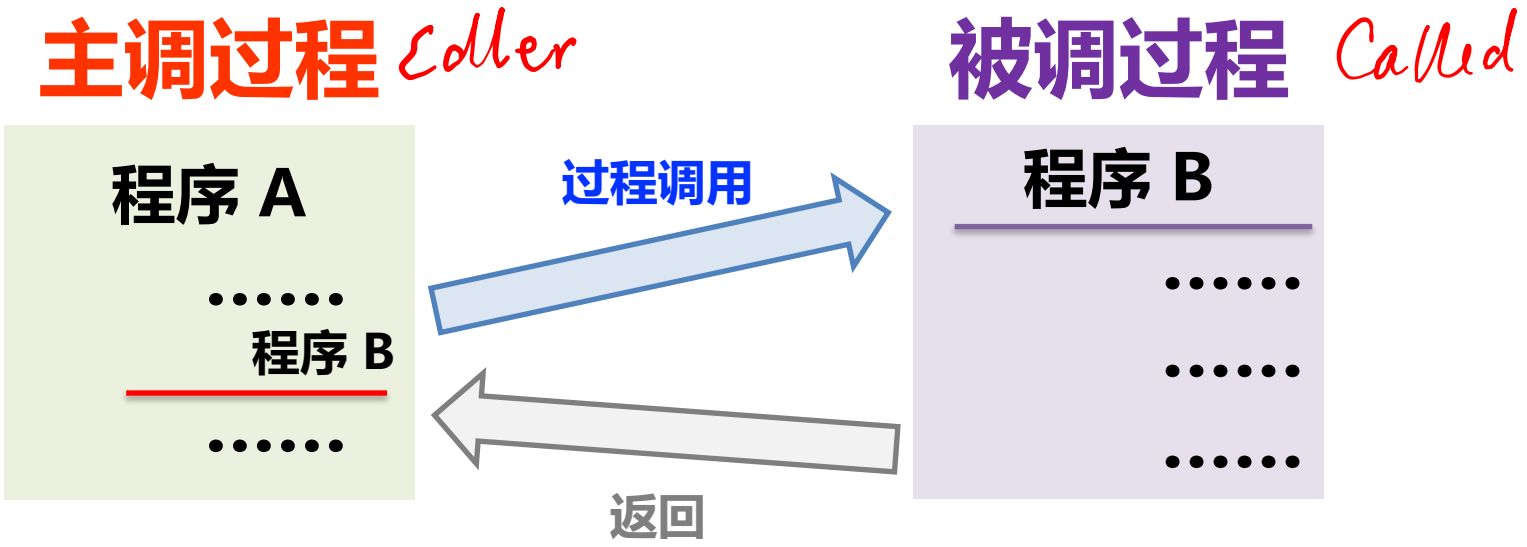
```
j label      #无条件跳转到标号label处
```

目录

- 指令设计思想与考虑
- MIPS指令集介绍
  - ✓ 指令存储与数据存储
  - ✓ 指令分类与格式
  - ✓ 寻址方式
  - ✓ 指令系统
  - ✓ 过程调用
- MIPS指令集程序

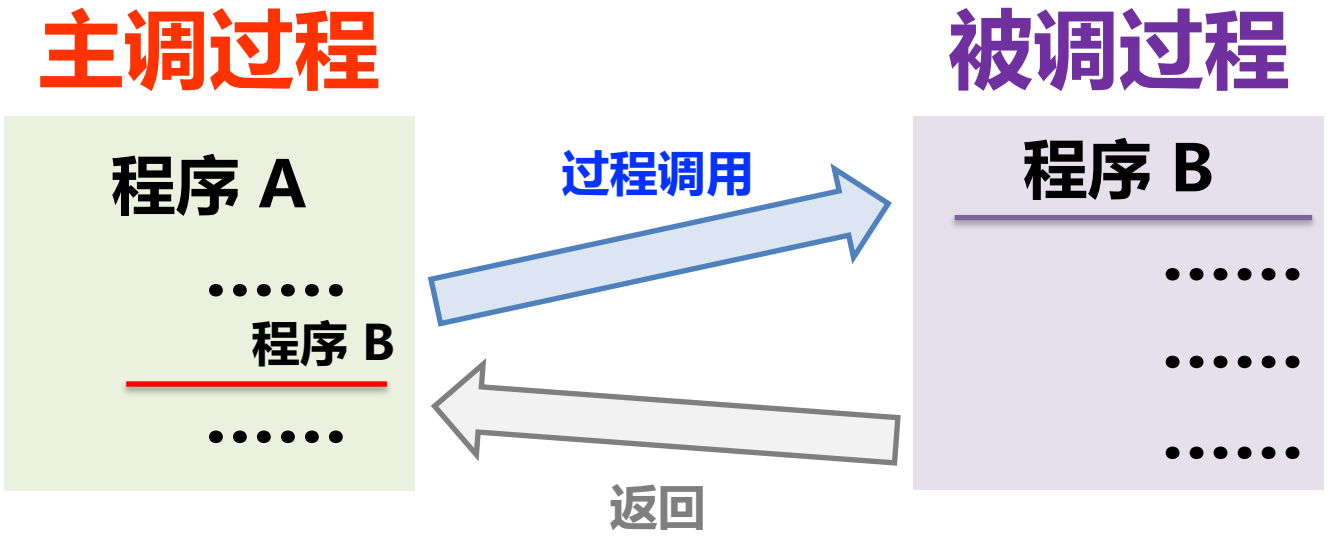
过程调用

- 一个过程包括**入口**，**过程体**和**出口**。
- 准备**过程参数（如果有）**
  - ✓ 并**跳转到过程入口**
  - ✓ 在执行完过程体中的代码
  - ✓ 从出口离开并回到主调过程同时获得该过程的**返回值（如果有）**



过程调用

- **MIPS使用寄存器存储参数和返回的数据/地址：**
  - ✓ 参数寄存器：**$a0 ~ $a3**
  - ✓ 返回值寄存器：**$v0 ~ $v1**
  - ✓ 返回地址寄存器：**$ra**



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(PC+4)保存在$ra寄存器中
                # 程序跳转到过程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
```

问题：为什么不使用跳转指令 *j* ？

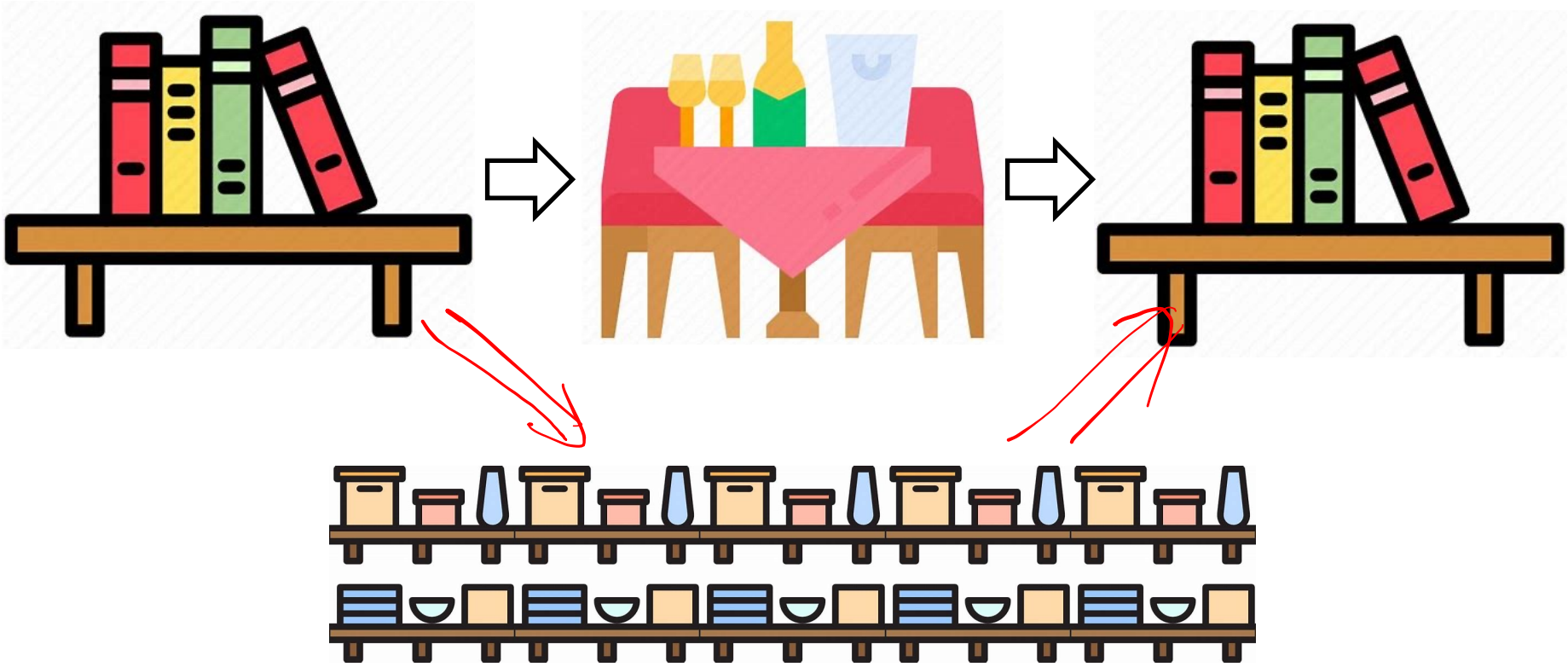
- ***j* 不保存返回地址；**
- ***j* 跳转范围不如 *jr*，且每次过程调用的跳转距离可能不同，返回地址也可能不同**

j、jr、jal指令（细节）

- *jal* 和 *j*：
  - ✓ *j*只跳转，不保存目标地址
  - ✓ 两者都属于J型指令

- *jr*属于R型指令，跳转到**寄存器中保存**的地址。j属于J型指令，跳转到一定范围内的目标地址

保存其他寄存器数据？



我们需要保存未完成的任务进度！

寄存器中保存什么数据？

- **不同的过程维护不同的数据类型**
  - ✓ 临时数据寄存器只被临时使用
  - ✓ 保存数据寄存器直到释放前一直被占用

Name	Register number	Usage	Preserved on call?
$zero	0	the constant value 0	n.a.
$v0–$v1	2–3	values for results and expression evaluation	no
$a0–$a3	4–7	arguments	no
$t0–$t7	8–15	temporaries	no
$s0–$s7	16–23	saved	yes
$t8–$t9	24–25	more temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

FIGURE 2.18 MIPS register conventions. Register 1, called $at, is reserved for the assembler (see Section 2.10), and registers 26–27, called $k0–$k1, are reserved for the operating system.

谁来保存寄存器数据?

- 过程调用保持情况（从caller的视角）
  - ✓ 主程序和子程序都需要维护一些数据

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

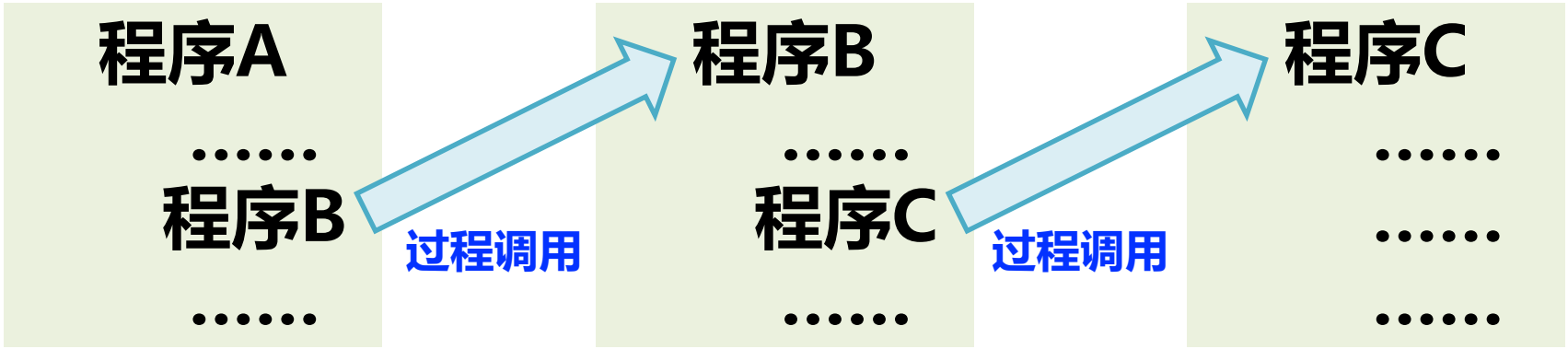
- **子程序不改变这些寄存器数据**
- **如果子程序要用，需要子程序维护好**

- **子程序可以改变这些寄存器数据**
- **如果主程序需要调用结束后继续使用，需主程序提前备份好**

嵌套过程调用

问题: 我们现在知道谁来维护寄存器数据

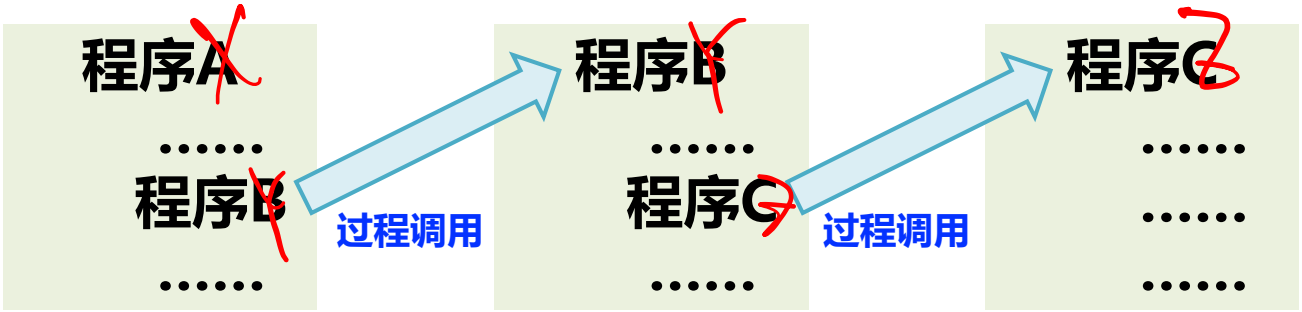
如何维护这些数据?

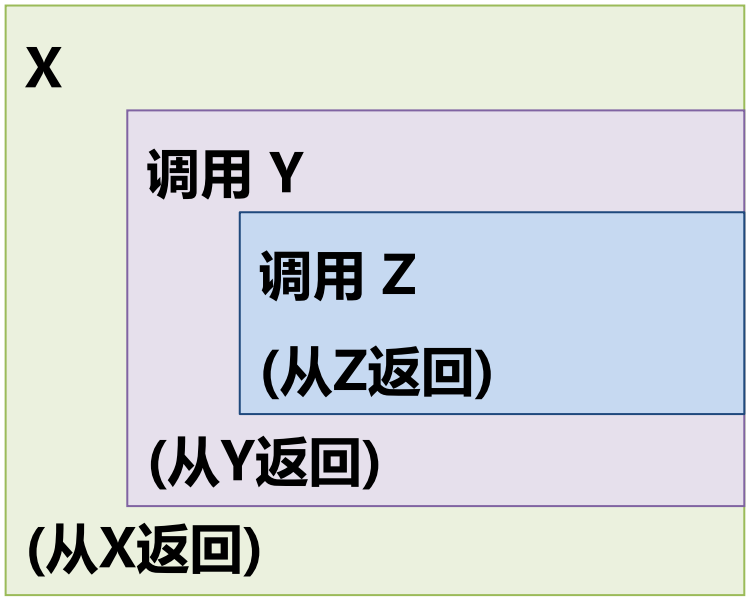


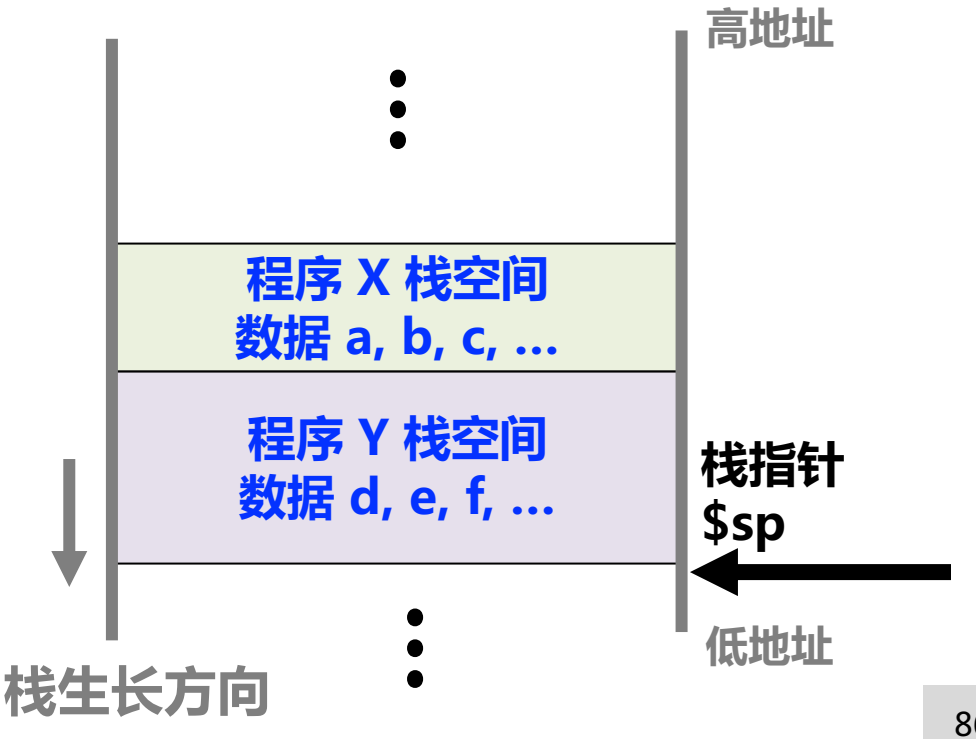
使用栈存储！ Stack

嵌套过程调用

❖ 使用主存储栈来保存数据!





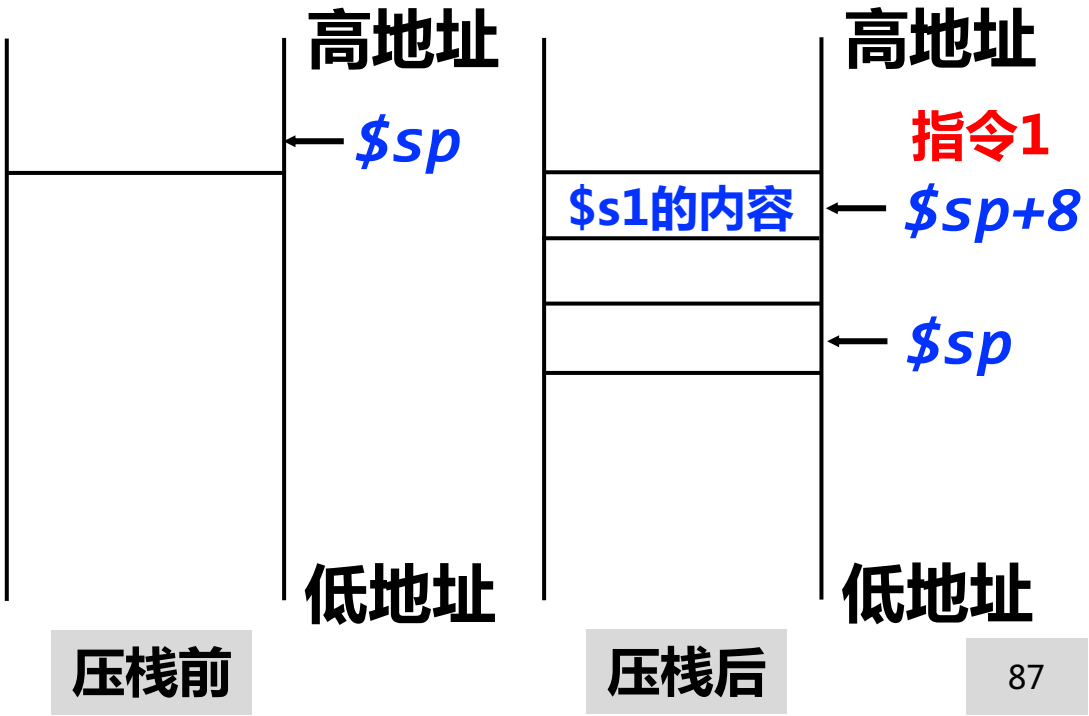


指令系统：栈操作

❖ 例：将 *$s1、$s2、$s3* 寄存器的内容压入栈

```
0. addi $sp, $sp, -12
1. sw   $s1, 8($sp)
2. sw   $s2, 4($sp)
3. sw   $s3, 0($sp)
```

习惯上，栈按照从高到到低的地址顺序增长

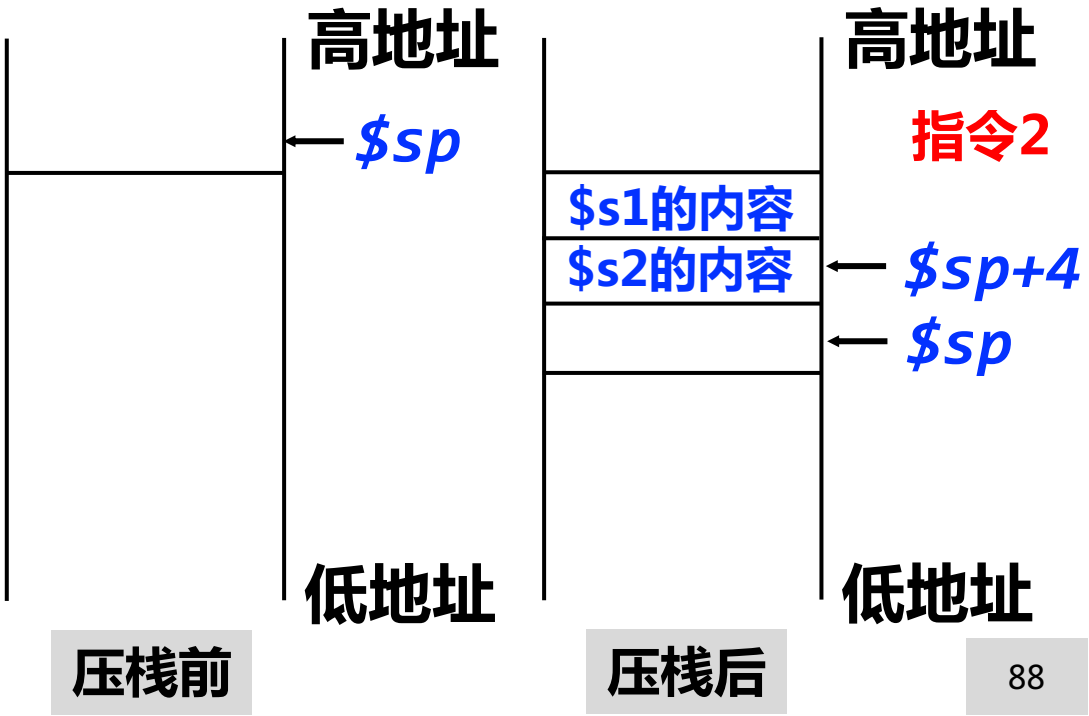


指令系统：栈操作

❖ 例：将 *$s1*、*$s2*、*$s3* 寄存器的内容压入栈

```
0. addi $sp, $sp, -12
1. sw   $s1, 8($sp)
2. sw   $s2, 4($sp)
3. sw   $s3, 0($sp)
```

习惯上，栈按照从高到到低的地址顺序增长

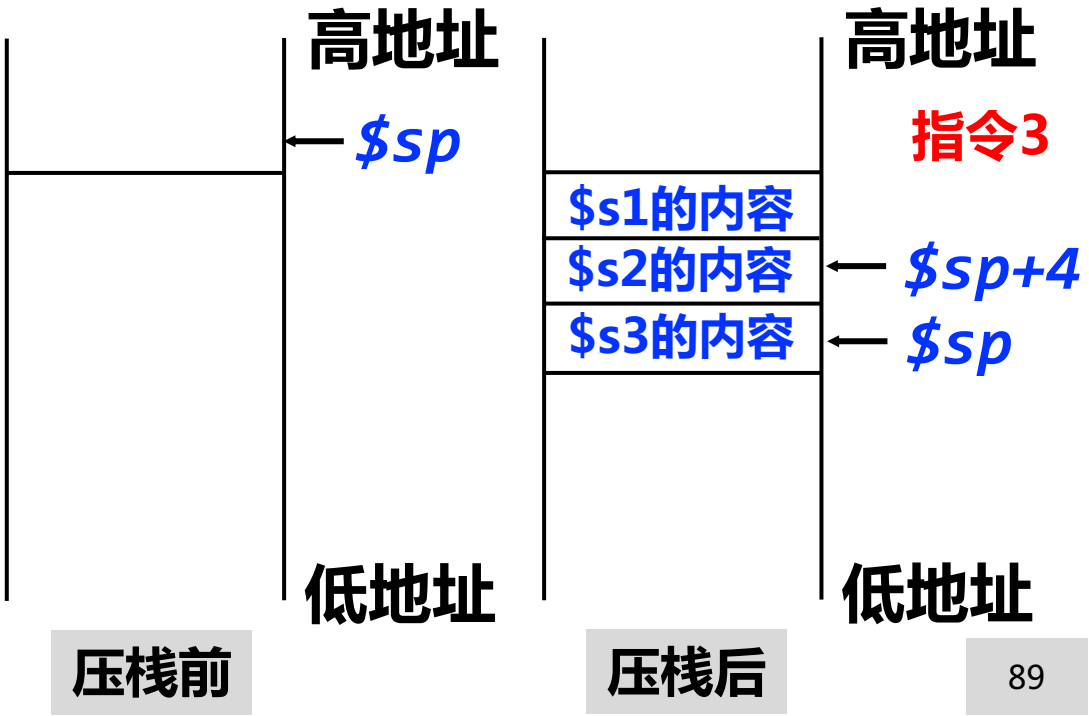


指令系统：栈操作

❖ 例：将 *$s1、$s2、$s3* 寄存器的内容压入栈

```
0. addi $sp, $sp, -12
1. sw   $s1, 8($sp)
2. sw   $s2, 4($sp)
3. sw   $s3, 0($sp)
```

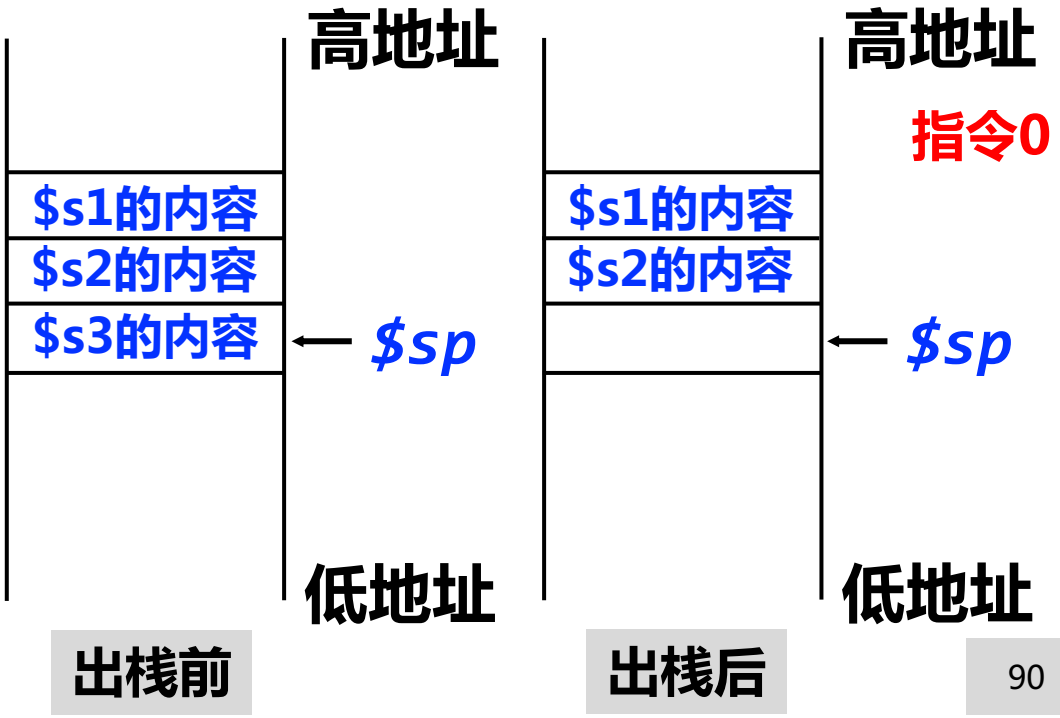
习惯上，栈按照从高到到低的地址顺序增长



指令系统：栈操作

❖ 出栈操作

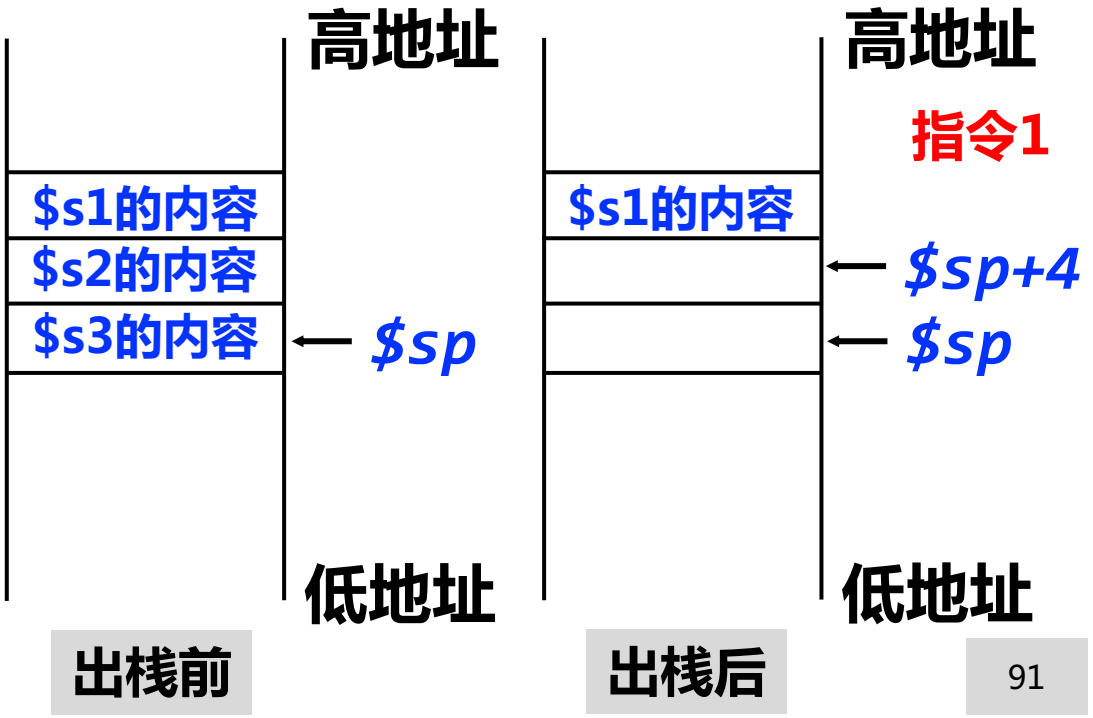
```
0. lw   $s3, 0($sp)
1. lw   $s2, 4($sp)
2. lw   $s1, 8($sp)
3. addi $sp, $sp, 12
```



指令系统：栈操作

❖ 出栈操作

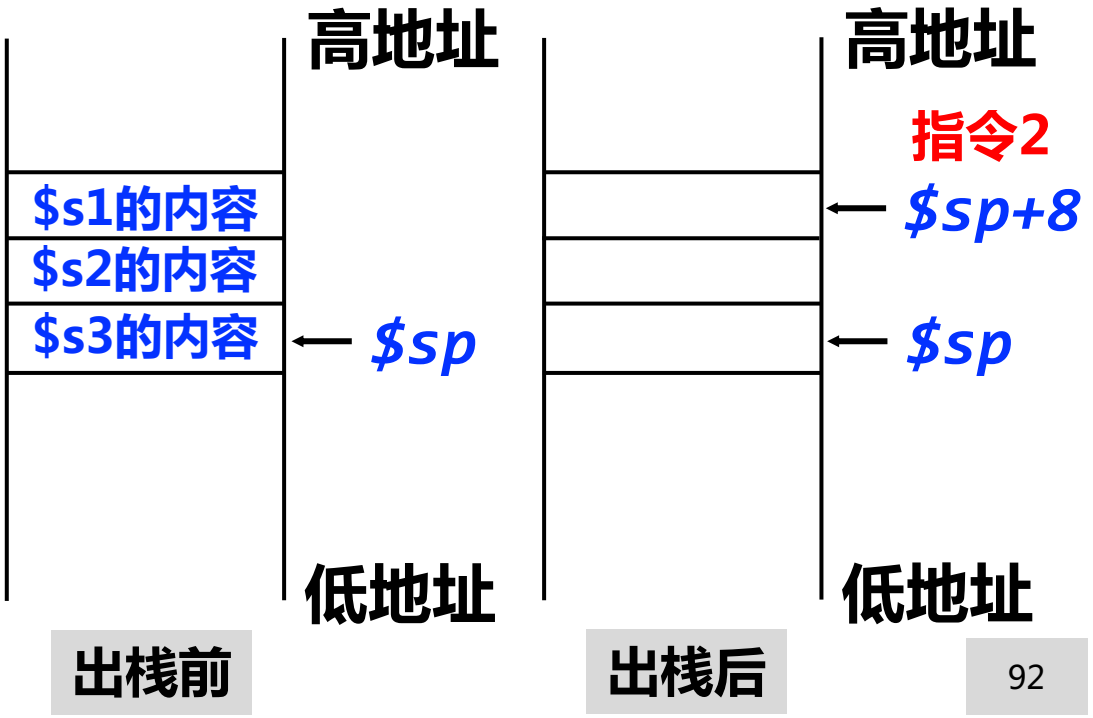
```
0. lw   $s3, 0($sp)
1. lw   $s2, 4($sp)
2. lw   $s1, 8($sp)
3. addi $sp, $sp, 12
```



指令系统：栈操作

❖ 出栈操作

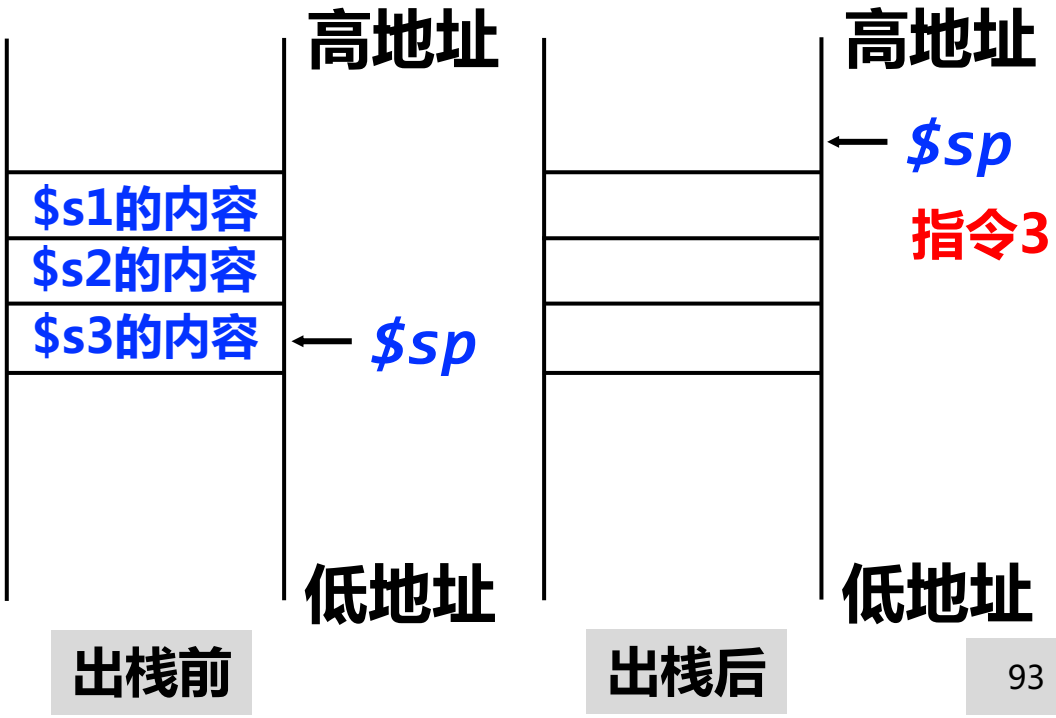
```
0. lw   $s3, 0($sp)
1. lw   $s2, 4($sp)
2. lw   $s1, 8($sp)
3. addi $sp, $sp, 12
```



指令系统：栈操作

❖ 出栈操作

```
0. lw   $s3, 0($sp)
1. lw   $s2, 4($sp)
2. lw   $s1, 8($sp)
3. addi $sp, $sp, 12
```



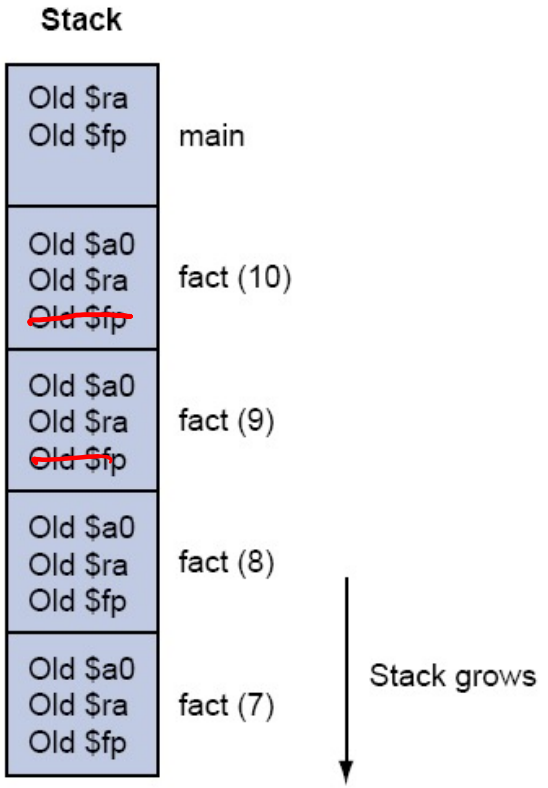
嵌套过程调用（课后）

- 计算n!

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

思考：左边的汇编代码与例程如何对应的？

```
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
```



嵌套过程调用（课后）

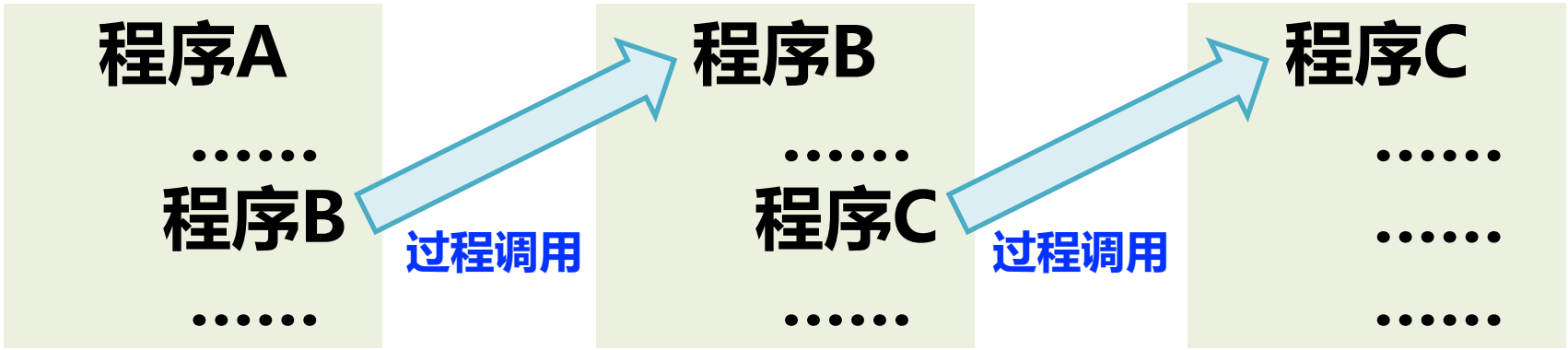
- 计算n!

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

```
fact:
        addi    $sp, $sp, -8        # 移动栈指针
        sw      $ra, 4($sp)
        sw      $a0, 0($sp)         # ra和n入栈
        slti    $t0, $a0, 1         # n和1比较
        beq     $t0, $zero, L1      # 如果a0>=1，跳转到L1
        addi    $v0, $zero, 1       # 如果a0<1，return 1
        addi    $sp, $sp, 8         # 恢复栈指针位置
        jr      $ra                 # 跳转回上一层函数
L1:                                  # 对应else
        addi    $a0, $a0, -1        # 计算本层函数的n=n-1
        jal     fact                # 进入fact(n-1)
        lw      $a0, 0($sp)
        lw      $ra, 4($sp)         # 恢复现场，取出上一层函数中的n和ra
        addi    $sp, $sp, 8         # 恢复栈指针
        mul     $v0, $a0, $v0       # return n*fact(n-1)
        jr      $ra                 # 返回上一层函数
```

叶过程

- 不调用其他过程的过程



叶过程

- 不调用其他过程的过程　　g, h, i, j用寄存器 $a0-$a3 进行传输

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

问题：叶过程中是否需要保存和恢复寄存器$t0、$t1？

```
leaf_example:
    addi    $sp,  $sp,  -12
    sw      $t1, 8($sp)
    sw      $t0, 4($sp)
    sw      $s0, 0($sp)
    add     $t0, $a0, $a1 //$t0=g+h
    add     $t1, $a2, $a3 //$t1=i+j
    sub     $s0, $t0, $t1 //$s0=$t0-$t1
    add     $v0, $s0, $zero
    lw      $s0, 0($sp)
    lw      $t0, 4($sp)
    lw      $t1, 8($sp)
    addi    $sp, $sp, 12
    jr      $ra
```

叶过程

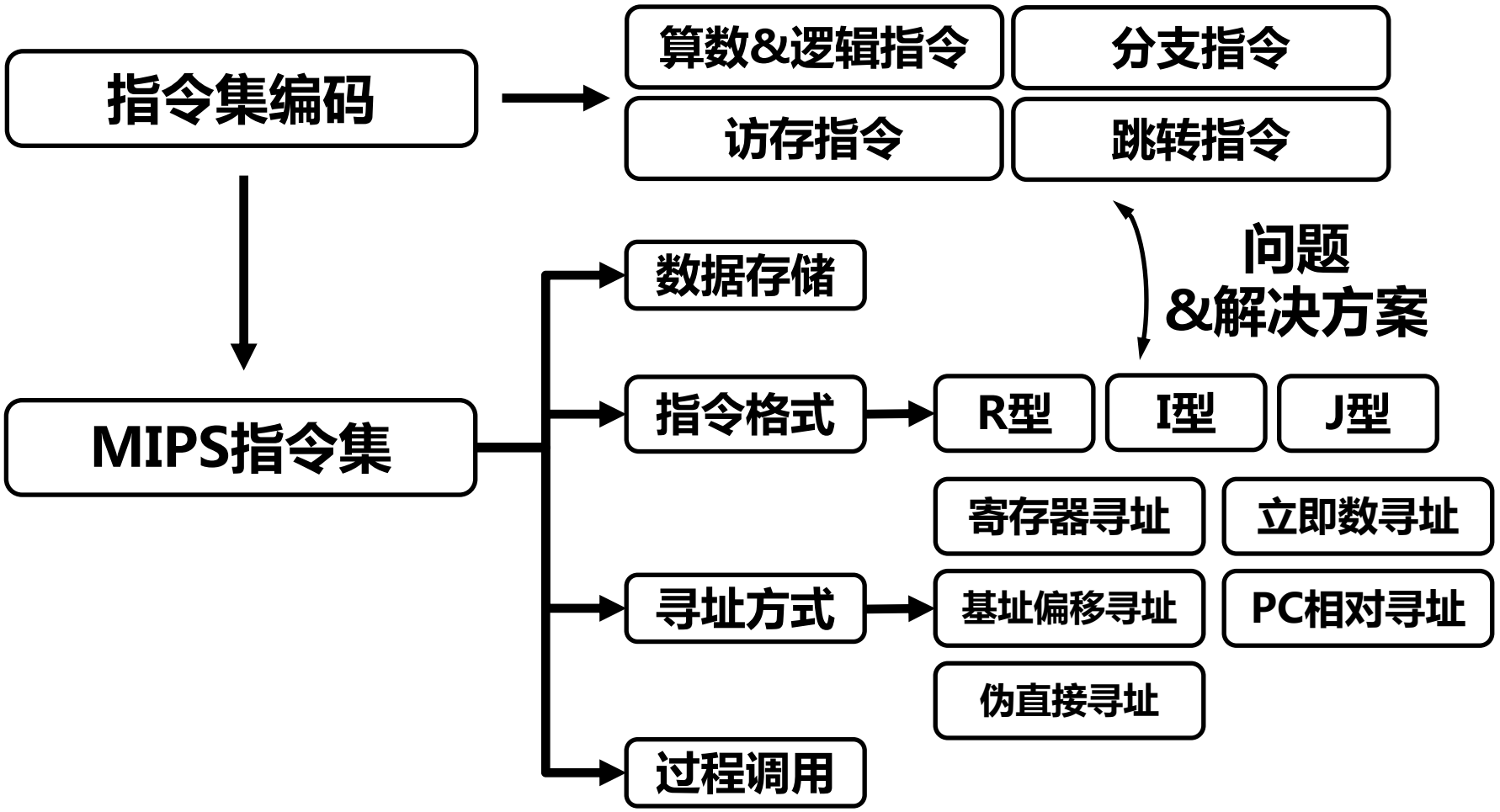
```
int  leaf_example (int g,
int h, int i, int j)
{
    int f ;
    f = (g + h) - (i + j);
    return f;
}
```

对于**叶过程**，由于返回上层程序后，临时寄存器的数值不需要保存，因此不需要压栈！

保存寄存器中的值在返回上层程序之后会被调用，必须保存！

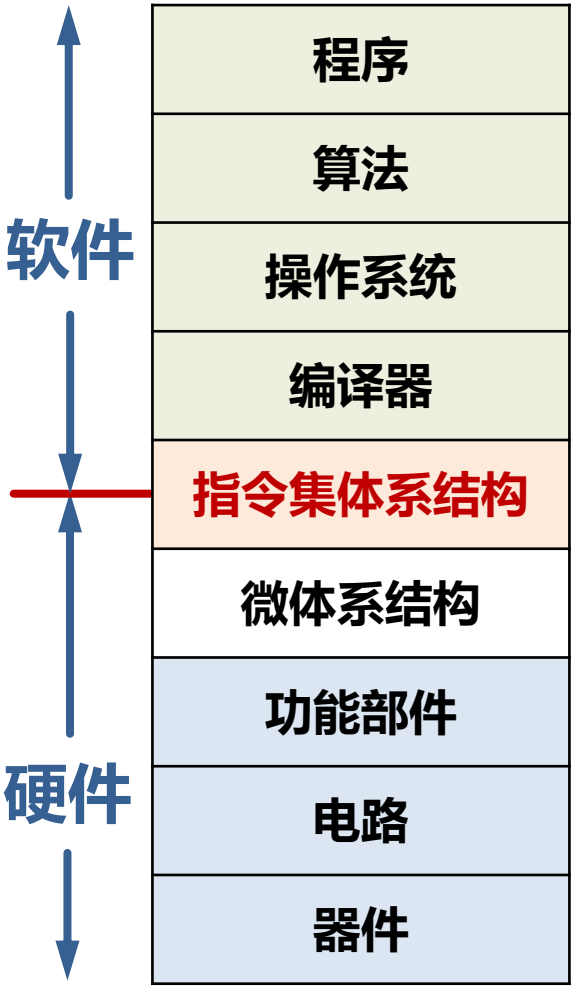
```
leaf_example:
    addi    $sp,  $sp,  -4
    sw      $t1, 8($sp)
    sw      $t0, 4($sp)
    sw      $s0, 0($sp)
    add     $t0, $a0, $a1
    add     $t1, $a2, $a3
    sub     $s0, $t0, $t1
    add     $v0, $s0, $zero
    lw      $s0, 0($sp)
    lw      $t0, 4($sp)
    lw      $t1, 8($sp)
    addi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

总结


指令集编码
算数&逻辑指令
分支指令
访存指令
跳转指令
问题
&解决方案
MIPS指令集
数据存储
指令格式
R型
I型
J型
寄存器寻址
立即数寻址
寻址方式
基址偏移寻址
PC相对寻址
伪直接寻址
过程调用

总结

- 指令集架构：桥梁，需要软硬件协同设计



计算机系统

后续学习建议

- 课程
  - ✓ 《现代计算机体系架构》——理论课
    - 进一步了解现代指令集的发展
  - ✓ 《基于DSP的系统设计》——实验课
    - 学习Blackfin DSP专用指令集的结构和编程
    - 深入理解硬件/软件之间的折衷和协同优化
- 阅读
  - ✓ 《计算机组成与设计硬件/软件接口》第二章
  - ✓ 《计算机系统结构：量化研究方法》附录A

附录

- 超级精简指令计算机URISC
- 关于xxx/xxxi/xxxu/xxxiu的使用细节
- MIPS编程指南（大作业参考资料）

超级精简指令计算机URISC

- 超级RISC结构是 Mavaddat 和 Parham 提出的一种RISC结构，它只有一条指令，指令集不能再精简，所以称为超级精简指令计算机(URISC)
- 尽管URISC只有一条指令，它也是图灵完备的计算机
  - ✓ 所有复杂操作都可以用这一种指令完成

URISC指令的执行过程

- 从第二个操作数中减去第一个操作数，并把运算结果存贮在第二个操作数的地址中
- 如果减法运算得到的结果为负数，则转移到指定的地址继续执行，否则顺序执行下一条指令
- 如果转移到地址0，则停止RISC的运行。

```
sbn a, b, label;     # Mem[b] = Mem[b] - Mem[a]
                     # if (Mem[b] < 0) goto label
```

指令及形式

- URISC指令
  - ✓ 做减运算且在结果为负值时转移
  - ✓ 只有一条指令，不需要对指令定义操作码
  - ✓ 指令中只需指出两个操作数且指出运算结果为负数时的转移地址

各种xxx/xxxi/xxxu/xxxiu

算术运算中：加法四种均有

xxx表示要考虑溢出，xxxu代表不输出溢出信号（overflow强制为0）

xxxi代表立即数计算，均为符号扩展，即使xxxiu也是对立即数符号扩展。

add：

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp
endif
```

addu：

Operation:

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

addi：

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + sign_extend(immediate)
if temp32 ≠ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rt] ← temp
endif
```

addiu：

Operation:

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← temp
```

各种xxx/xxxi/xxxu/xxxiu

算术运算中：减法没有立即数运算，实际是加相反数
xxx表示要考虑溢出，xxxu代表不输出溢出信号（overflow强制为0）
xxxi代表立即数计算，均为符号扩展，即使xxxiu也是对立即数符号扩展。

sub：

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) - (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp31..0
endif
```

subu：

Operation:

```
temp   ← GPR[rs] - GPR[rt]
GPR[rd]← temp
```

subi：

为伪指令，编译时直接转化加相反数并检测溢出

subiu：

为伪指令，编译时直接转化为加相反数不检测溢出

各种xxx/xxxi/xxxu/xxxiu

slt不会输出溢出信号，u指的是将32位数看成无符号数进行比较（等价于在最高位之前再补一个0）。这里立即数扩展依然为有符号扩展。

slt：

Operation:

```
if GPR[rs] < GPR[rt] then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

sltu：

Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

slti：

Operation:

```
if GPR[rs] < sign_extend(immediate) then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

sltiu：

Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

各种xxx/xxxi/xxxu/xxxiu

逻辑运算中，xxxi代表立即数计算，均为无符号扩展（注意与前几种xxxi区分）

and：

Operation:

```
GPR[rd] ← GPR[rs] and GPR[rt]
```

andi：

Operation:

```
GPR[rt] ← GPR[rs] and zero_extend(immediate)
```

or：

Operation:

```
GPR[rd] ← GPR[rs] or GPR[rt]
```

ori：

Operation:

```
GPR[rt] ← GPR[rs] or zero_extend(immediate)
```

xor：

Operation:

```
GPR[rd] ← GPR[rs] xor GPR[rt]
```

xori：

Operation:

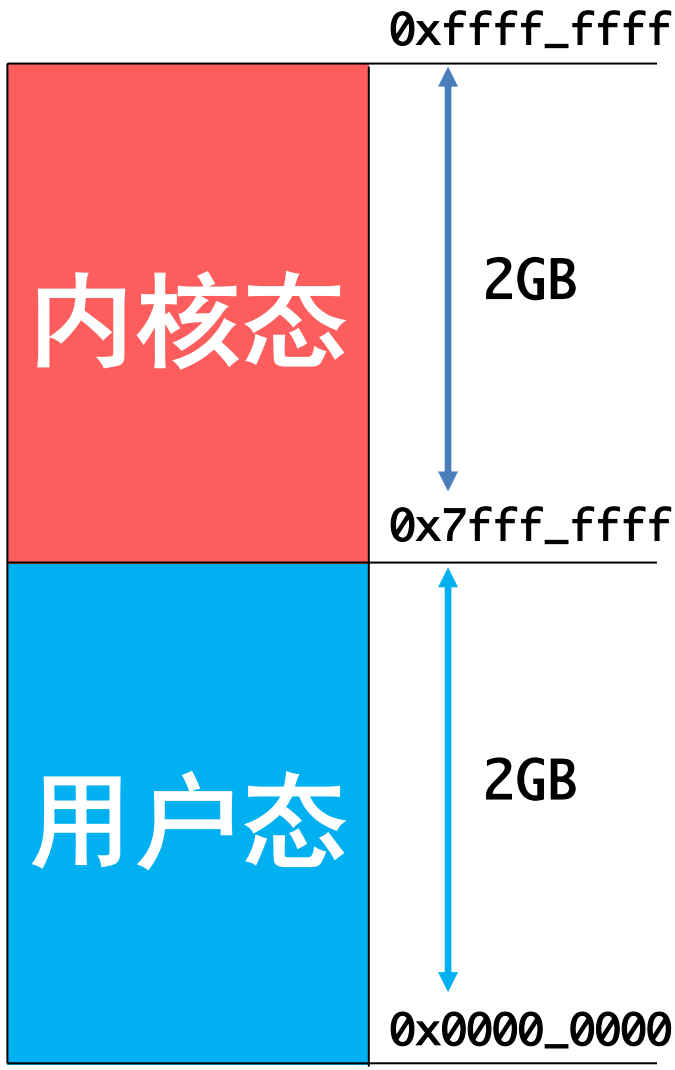
```
GPR[rt] ← GPR[rs] xor zero_extend(immediate)
```

MIPS编程指南

- 32位MIPS的内存分配
- MIPS模拟器
- 汇编程序设计基础

32位MIPS的内存分配

- 内存空间分配
  - ✓ 32位的地址决定了能管理的内存最大为2^{32}=4GB的大小。
  - ✓ 一般我们将低2GB规定为用户态空间，即一般的应用程序可以控制的空间。
  - ✓ 高2GB的空间属于内核态空间，这部分空间是由操作系统控制的，用户态程序不能控制。



32位MIPS的内存分配

- 应用程序中常见的数据

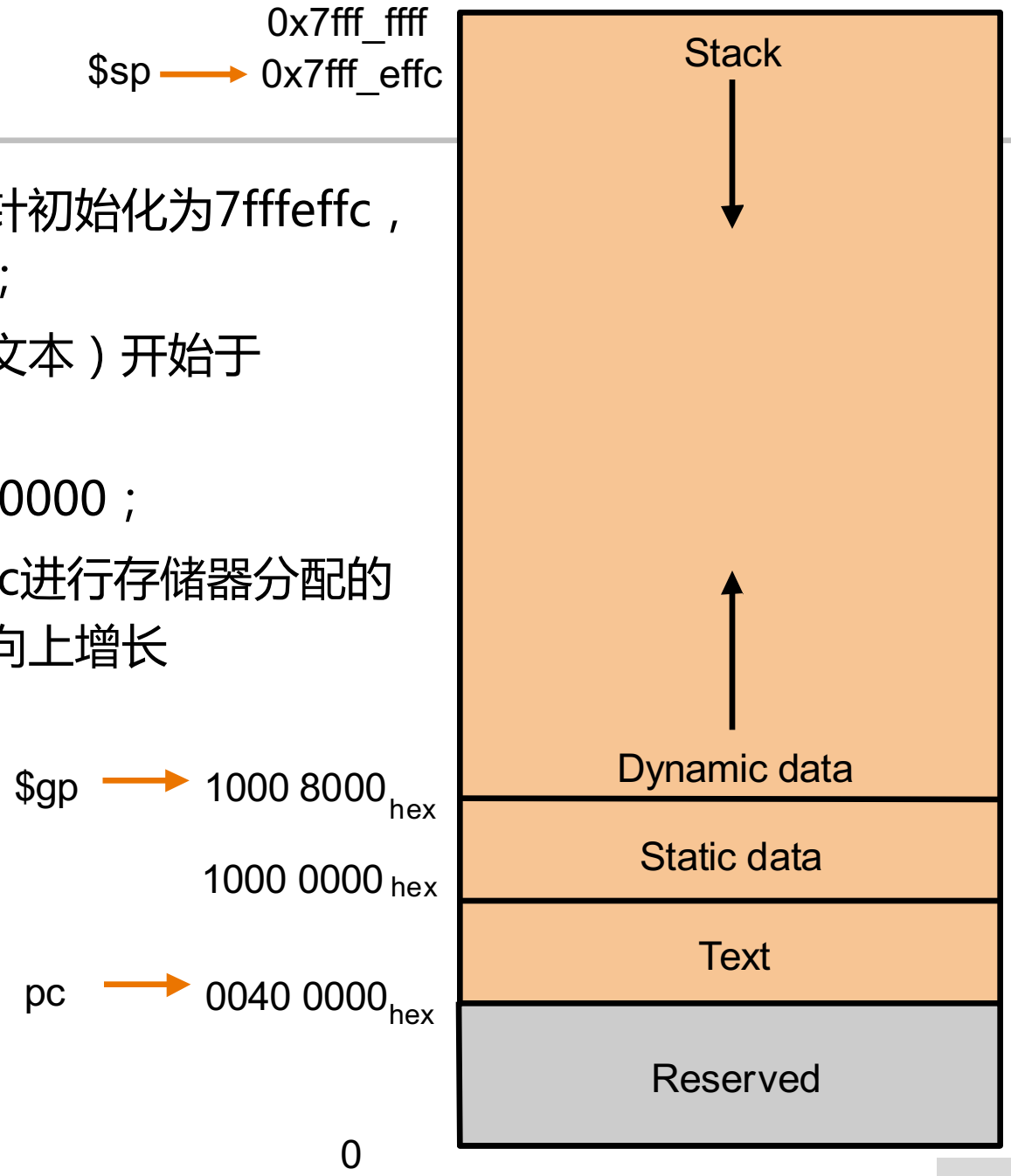
```
int global;

int main()
{
    int local;
    char array0[10];
    int * array1;
    array1=(int *)malloc(10 * sizeof(int));
    free(array1);
}
```

程序运行时这些数据都被放在内存空间的什么地方呢？

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7fffeffc，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



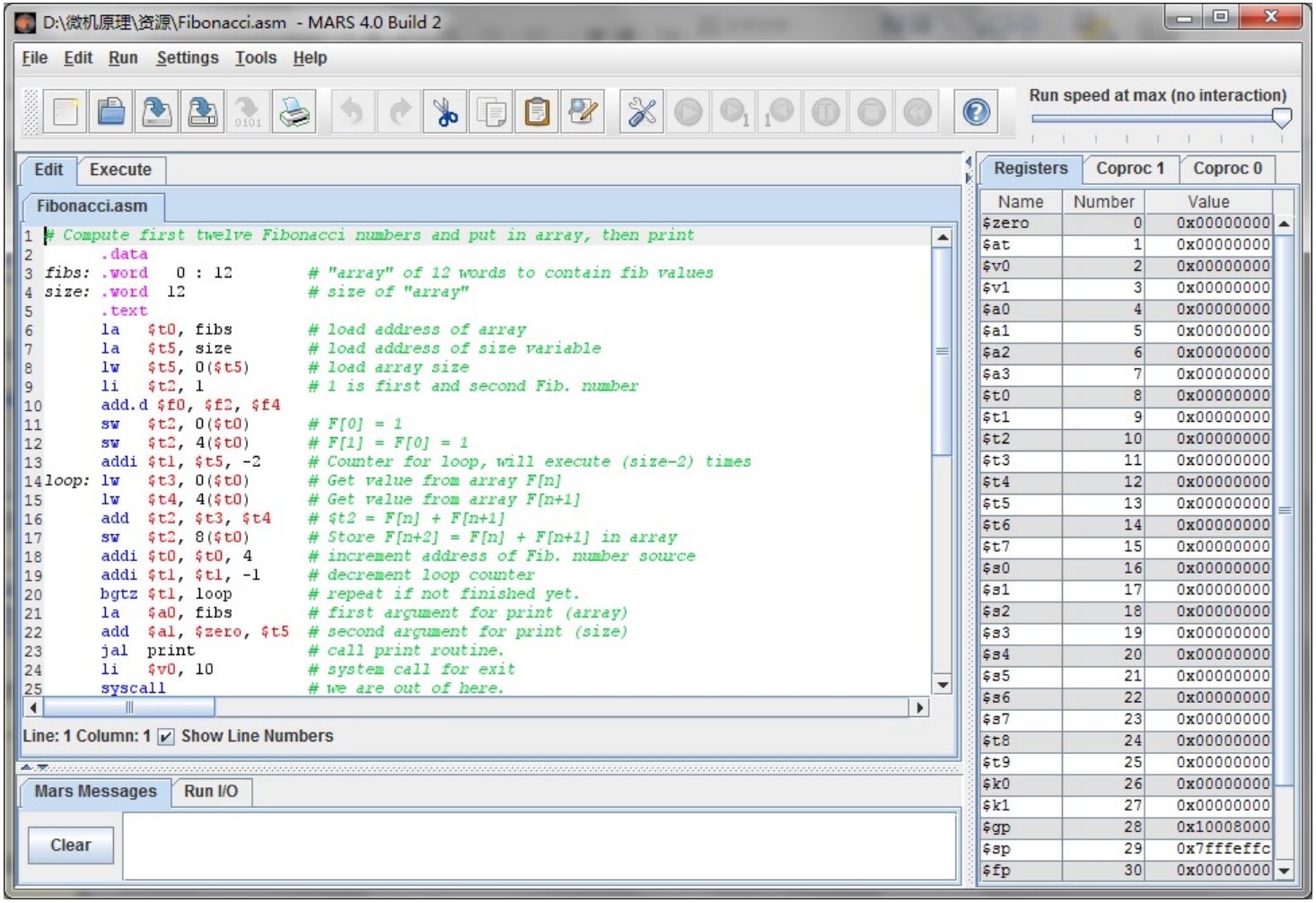
全局指针被设定为易于访问数据的地址，以便使用相对于$gp的±16位偏移量

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

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

汇编程序基础

- 汇编程序设计基础
  - ✓ 语法
  - ✓ 变量
  - ✓ 分支
  - ✓ 数组
  - ✓ 过程调用

编程指南：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
```

编程指南：语法

- 指令与伪指令语句

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

  例如 AddFunc: add $a1 $a2 $a3 # a1=a2+a3

- 汇编命令(directive)语句

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

  例如 .word 0xa3

编程指南：语法

- 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

.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 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

编程指南：语法

- 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
  - ✓规定汇编程序在地址0x00400000开始

```
0x00400020          move $s5, $31
0x00400024          jal 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          addi   $s1,$s1,-1
0x00400110          bnez   $s1,loop
0x00400114          jr     $31
    …         .data 0x10000200
0x10000200  n:      .word 4
0x10000204  f:      .word 0
```

该程序功能是求阶乘，结果是f=n！通过跳转到fact子程序，通过一个循环将n到1都乘在$s0上来求n!。
注意！n,f超出了16位的偏移，而sw和lw是I型指令，所以这样的指令是写不出来的，该程序实际无法运行。

编程指南：语法

- 指令与伪指令
  - ✓ 有一些MIPS指令是和机器码一一对应，可以直接翻译成机器码。
    - add $a,$b,$c　　lw $t1 4($s1)
  - ✓ 还有一些没有对应的机器码，不能直接翻译成机器码，需要先翻译成真的指令。
    - li $s1 0x7f　<====> addi $s1 $zero 0x7f
  - ✓ 编写程序时使用伪指令有利于提高效率并增加可读性。

编程指南：语法

```
    .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) #跳出循环时 $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
```

两个.text实现程序调用

为了让程序可以运行，我们把n和f拆成高低16位，分别装入，再寻址。main程序使用了beq $0,$0,fact进行了跳转。这是一个可运行的版本，该程序用了全局变量（n，f）进行计算的

能运行的版本（2）

这是另一个可执行的版本，这里fact是接在main后面的，没有用跳转指令直接运行。

```
        .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:
        ori         $s6,$0,0x1000       #  获得数据起始地址
        sll          $s6,$s6,16         #  $s6=0x10000000
        addiu  $s5,$s6,0x004            #  $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             #   对应伪指令 subu $s1,$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：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
  - ✓而且事实上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：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
        jr            $ra         #return result in s0
```

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，本程序中没有用可删除以节省寄存器和指令数。

```
lui $s6, 0x1000
```

编程指南：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,$0,0x??**
  means PC ← 0x00400100 ?

需要计算出偏移量

```
        .text
main:   addu  $s3,$ra,$0
        ori     $s6,$0,0x1000
        sll      $s6,$s6,16
        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：分支

- 分支
  - ✓如果和 0比较, 则直接使用blez,bgez,bltz,bgtz, bnez (指令)
    - 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: ***test2***

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

编程指南：分支

```
        .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         #$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： 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的$A和$B

```
lw    $temp, 0($A)        temp =A[0];
sw    $temp, 8($B)        B[2]= temp;
```

  - ✓将常数0，8作为地址偏移量
  - ✓将寄存器**$A和$B**作为数组中的开始地址(A[]，B[])

注意这里$A, $B存的是地址，$temp存的是值

编程指南： 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
        addiu $s6,$s5,0x400
        addiu $s0,$0,0x11
loop:   subu  $s0,$s0,1
        addiu $s1,$0,4
        mul     $s2,$s1,$s0
        addu  $s3,$s2,$s5
        lw        $s4,0($s3)
        addu  $s3,$s2,$s6
        sw       $s4,0($s3)
        bnez    $s0,loop
        addu  $ra,$0,$s7
        jr $ra
```

每个数4Bytes/1word

功能：将数组A的值依次拷贝到数组B中

```
#$s5=A[ ]=0x10000000，
#$s6=B[ ]=0x10000400
#Size(A)=Size(B)=0x11
# 计数

# 换算地址
#计算A[]偏移量,送到$s3
#读出A[]中的值
#计算B[]偏移量,送到$s3
#写到B[]中去

#返回调用程序
```

编程指南： 数组array

- **使用移位操作代替mul和div**（mul 和 div 一般都比sll和srl慢）
  - ✓ sllv **by** k **等价于** mul **by** 2^k
  - ✓ srlv **by** k **等价于** div **by** 2^k

  } **只对无符号数成立，且没有超出数据表示范围**

- 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： 数组array

```
    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: …
```

用移位代替乘法

编程指南：过程调用

- 过程调用的定义以及使用栈的作用
- 过程的栈如何构成的（过程调用帧的组成部分以及顺序）
- 简单参数传递和返回值传递规则
- 例程分析。具体的过程调用中，变量是如何进栈出栈的

重点是过程调用中的常用约定和编程方法

编程指南：过程调用

- 参数寄存器和者临时寄存器入栈：调用后还需使用的参数寄存器$a0～$a3和临时寄存器$t0～$t9压栈。这些寄存器可能会被子程序修改（按约定，子程序不负责维护），所以返回后如仍需使用就需要主程序来备份。

对于叶过程，参数寄存器和临时寄存器的数值不需要入栈保存！

编程指南：过程调用

- 返回地址入栈：存储返回地址寄存器$ra的值（从子程序返回后，主程序要执行的下一条指令的地址）。此值在当前过程执行开始时入栈，并在当前过程返回之前复制回$ra寄存器，以保存和恢复该过程的返回地址

编程指南：过程调用

- 保存寄存器（$s0-$s7）的值入栈：存储当前过程想要使用的保存寄存器($s0到$s7)的值。主程序默认子程序会维护，所以子程序如要使用/修改其值，需要先备份再恢复：在进入当前过程时，将保存寄存器的值($s0到$s7)复制到这部分空间中；此后，当前过程可以随意更改保存寄存器的值。但是当该过程调用结束、返回之前，将这些值从栈中复制回原始的保存寄存器，使其值变成和调用该过程前相同。

编程指南：过程调用

- 参数传递规则

  简单参数传递规则：前四个参数通过寄存器（$a0-$a3）传递，更多的参数通过栈传递。复杂参数（结构体、联合体、可变参数传递见参考资料）

- 返回值的传递规则

  在返回基本数据类型的情况下，与常规参数传递一样，有专用的寄存器约定为传递返回值，整型在通用寄存器$v0中返回，浮点在浮点寄存器$f0中返回。

编程指南： 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

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

- 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“过程调用帧”

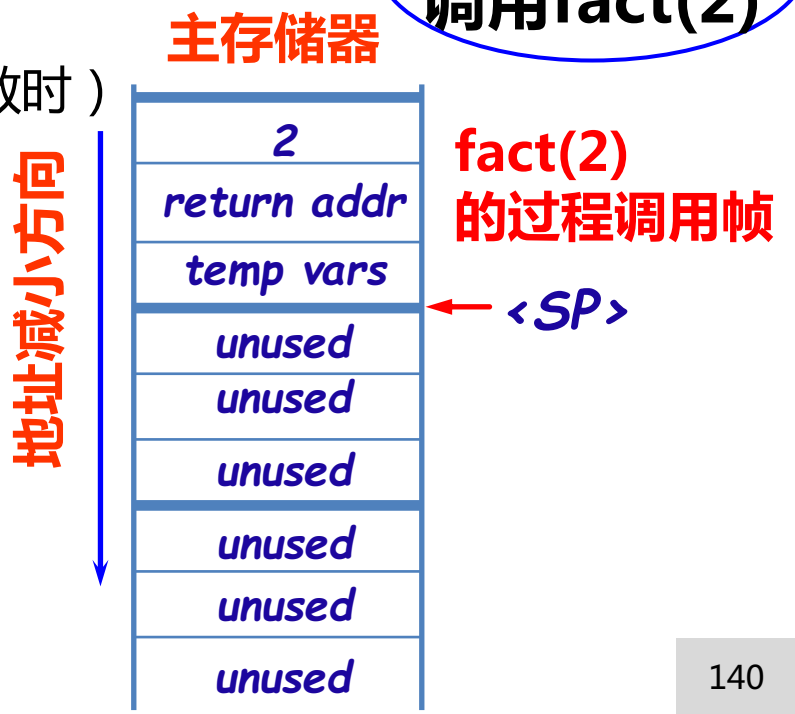
编程指南： 过程调用

- 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

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

例如
调用fact(2)

- 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“过程调用帧”



编程指南： 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

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

- 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“过程调用帧”

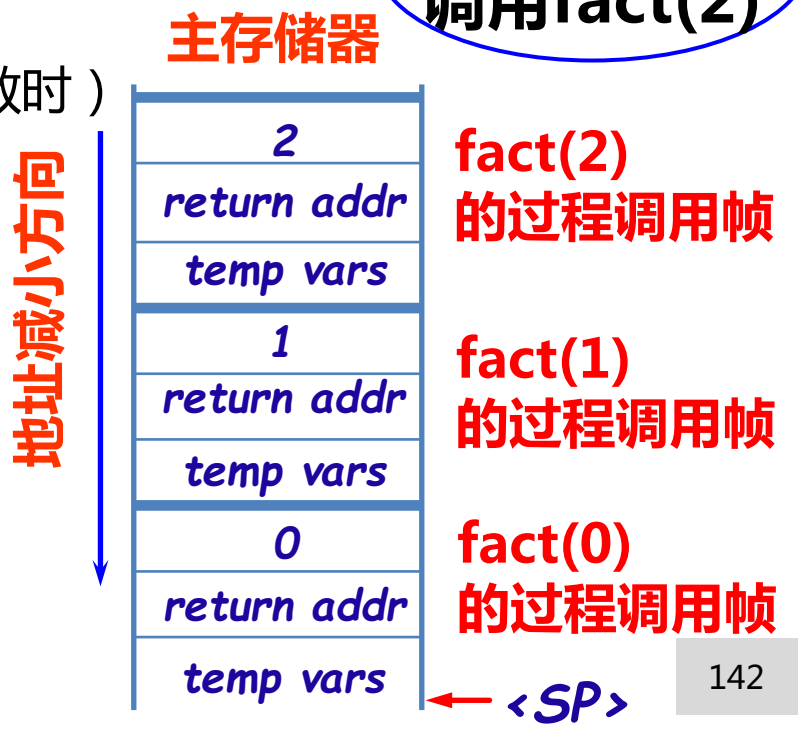
编程指南：过程调用

- 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

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

- 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“过程调用帧”

例如
调用fact(2)



编程指南： 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

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

- 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“过程调用帧”

编程指南：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

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

- 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“过程调用帧”

编程指南： 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

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

- 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“过程调用帧”

编程指南：过程调用

- 一般来说，过程调用帧可能包含：

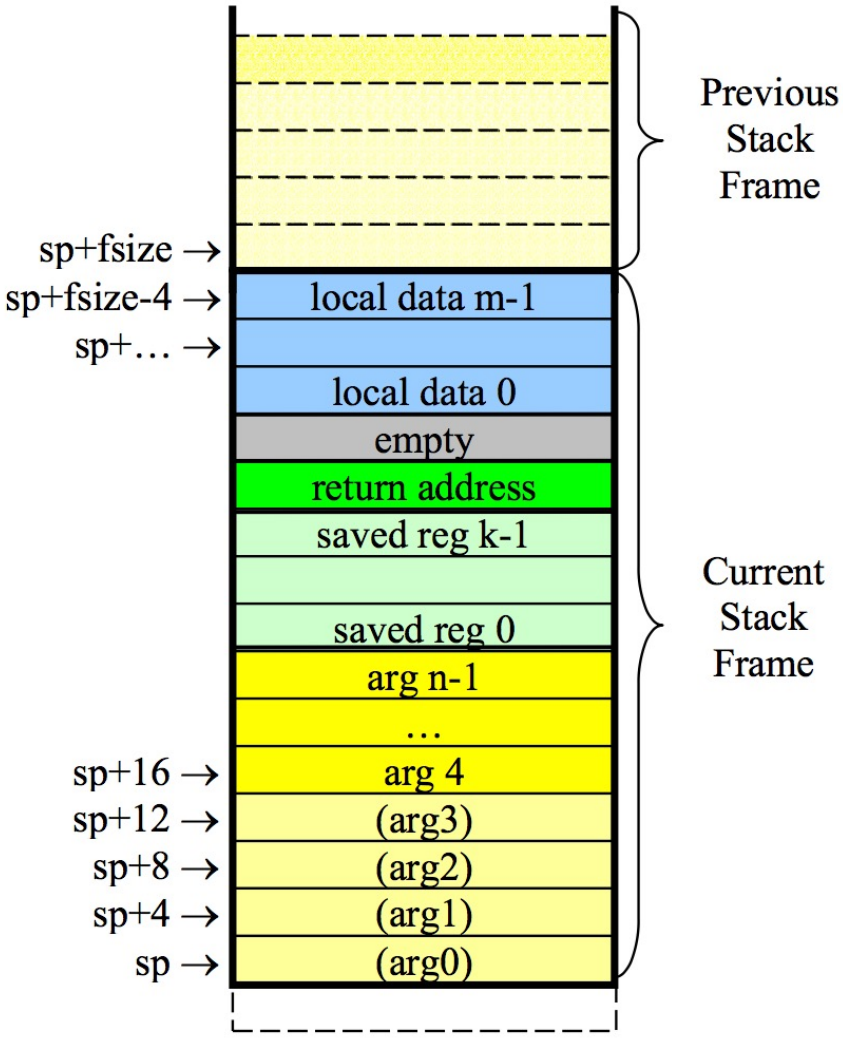
（1）传递给该过程调用的子过程的参数

（2）保存寄存器($s0到$s7)值

（3）子过程返回地址 ($ra)

（4）本地数据

编程指南：过程调用

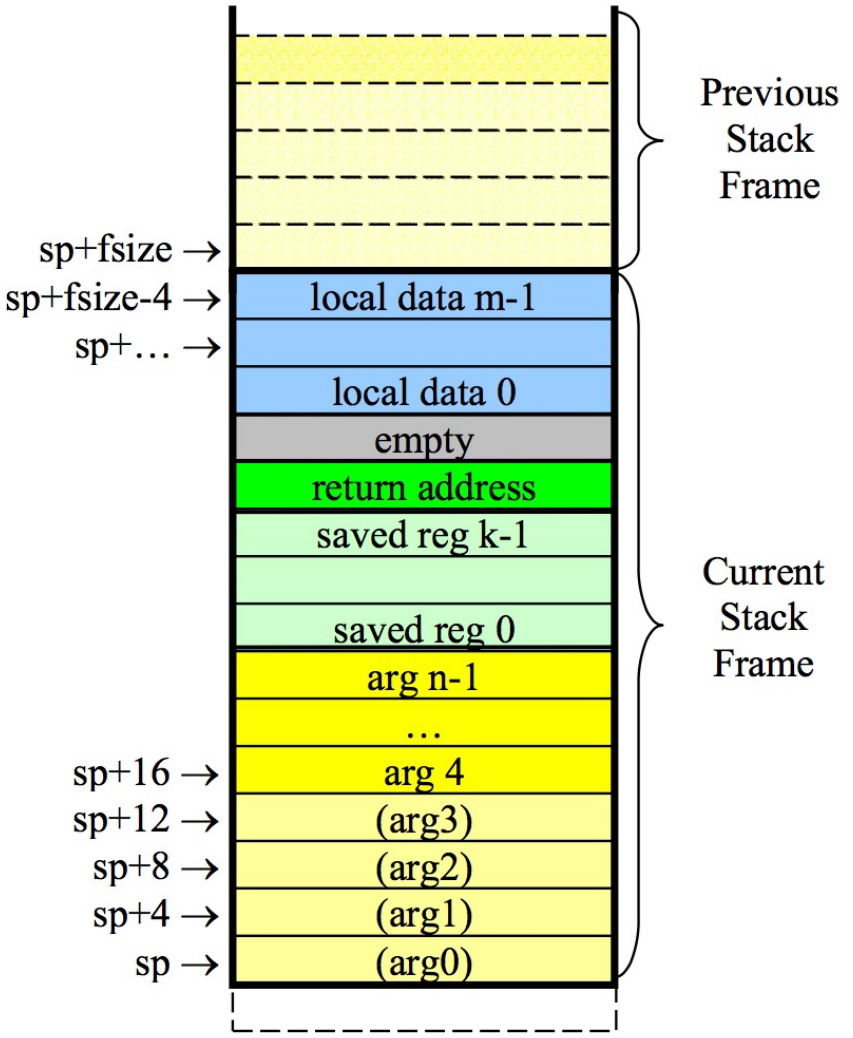


Note that in this figure the stack is growing in a *downward* direction. I.e., the *top* of the stack is at the *bottom* of the picture.

把一个过程调用帧分为5部分：

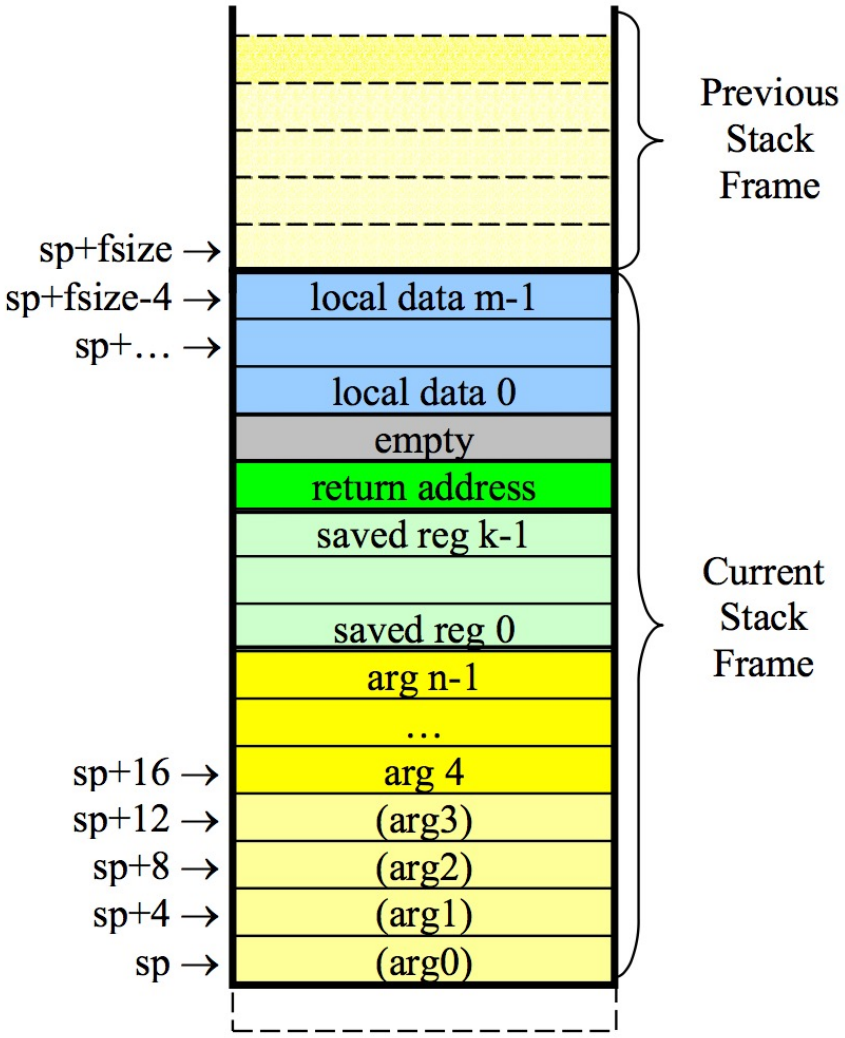
1. 参数部分：栈顶用来保存该过程传递给子过程的参数。前四个位置是不会被当前过程使用的，实际的参数通过对应的参数寄存器（$a0 to $a3 ），传递给该过程的子过程。如果传递的参数超过4个，多出的参数会存储在sp+16, sp+20, sp+24, ……

编程指南：过程调用



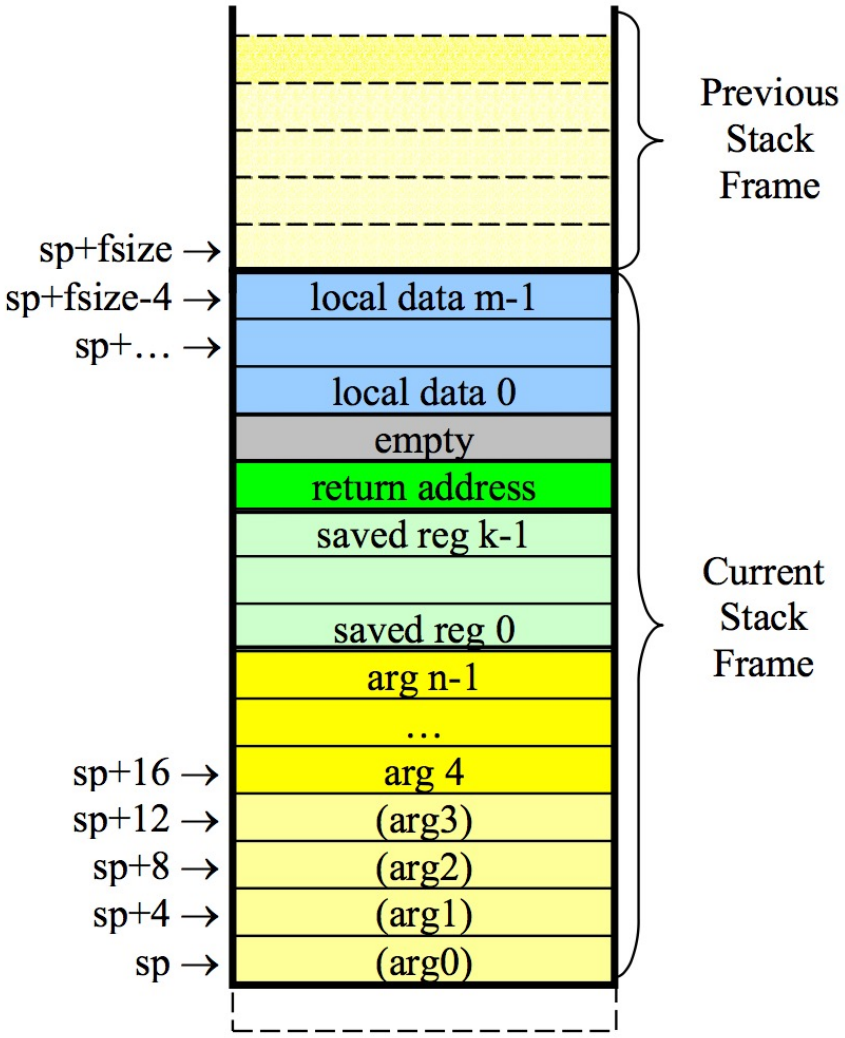
2. **保存寄存器部分**：存储当前过程想要使用的任何保存寄存器($s0到$s7)的值。
在进入当前过程时，将保存寄存器的值($s0到$s7)复制到这部分空间中，在执行该过程的过程中可能会更改这些寄存器的值。就在过程返回之前，它将这些值从栈中复制回原始的保存寄存器。在这期间，当前过程可以随意更改保存寄存器的值。但是当该过程调用结束，保存寄存器的值将变成和调用该过程前相同。
作用是保证保存寄存器的值不被过程修改

编程指南：过程调用



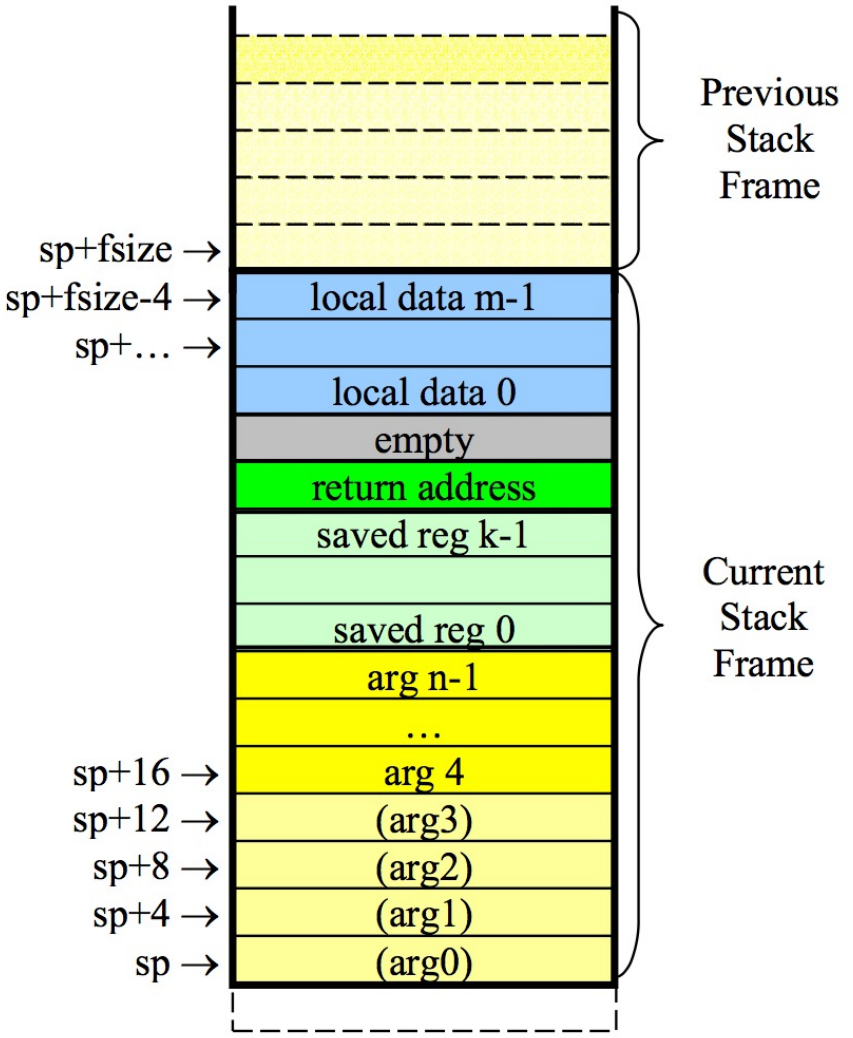
3. 返回地址部分：用于存储返回地址寄存器$ra的值。
此值在当前过程执行开始时复制到栈中，并在当前过程返回之前复制回$ra寄存器。
作用是保存该过程的返回地址

编程指南：过程调用



4. 空白部分： 在栈中插入空白，以确保栈的大小总是8的倍数。它被插入到这里，以确保本地数据存储区域从双字节边界开始。

编程指南：过程调用



5. 本地数据部分：用于本地变量存储。当前过程必须在此区域为其所有本地数据保留足够的存储字，包括存储任何临时寄存器值的空间($t0到$t9)，它需要在该过程调用期间保留这些值。还必须填充本地数据存储区域，使其大小总是8字节的倍数。

编程指南：过程调用

- 额外的规则：
- 栈指针的值必须在任何时候都是8的倍数。这确保可以将64位数据对象入栈，而不会在运行时产生地址对齐错误。这意味着每个栈帧的大小必须是8的倍数。
- 参数寄存器$a0-$a3的值在过程调用中不需要保留。因此，允许过程更改任何参数寄存器的值，而不保存/恢复它们。
- 栈帧的参数部分的前四个字节被称为参数槽——用来存储四个参数$a0-$a3。值得注意的是，一个主调过程不会在前四个参数槽中存储任何内容，实际的参数以$a0到$a3的形式传递。但是，被调用的子过程如果想保存参数寄存器的值，可以选择将$a0到$a3的值复制到参数槽中。

编程指南：过程调用

- 参数槽的一点说明：
  - ✓ 参数槽由调用者分配，但由被调用者使用!
  - ✓ 所有四个参数槽都是必需的，即使调用者知道它传递的参数少于四个。因此，在进入该过程时，如果需要，过程可以合法地将所有的参数寄存器存储到参数槽中。
  - ✓ 调用方必须在过程调用帧中为其调用的任何子过程分配最大参数数的空间(或者如果其调用的所有子过程的参数数都少于4个，为所有子过程都分配4个参数数)。

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