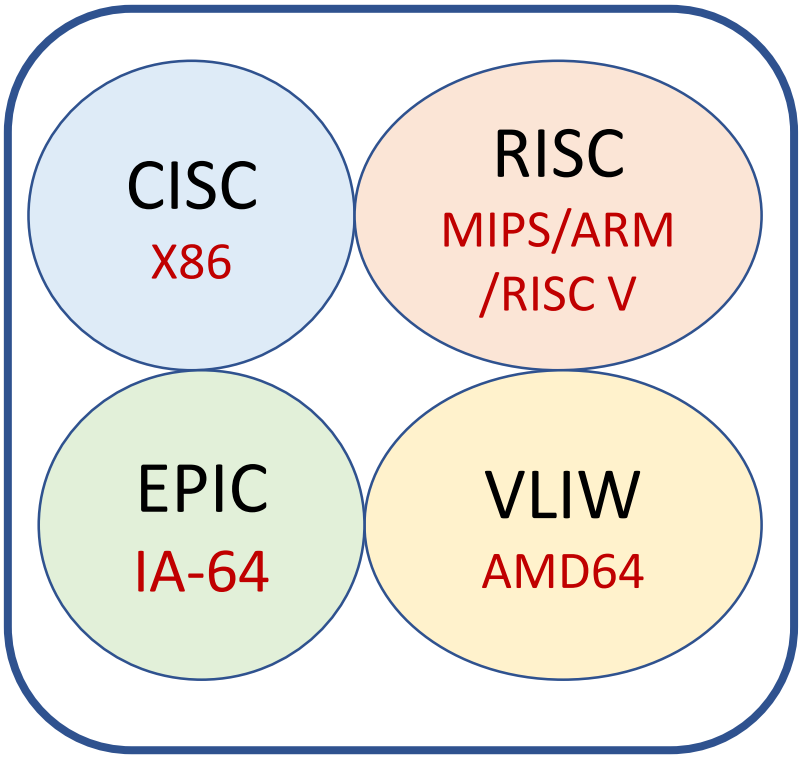
第2章 计算机指令集体系结构



第2章 计算机指令集体系结构

2.1 指令集体系结构的分类

- 堆栈结构
- 累加器结构
- 通用寄存器结构

2.2 寻址方式

2.3 指令系统的设计和优化

2.4 操作数的类型和大小

2.5 指令格式的设计和优化

2.6 MIPS指令集体系结构

2.1 指令集体系结构的分类

- 区别不同指令集体系结构的主要因素：
  - CPU中用来存储操作数的存储单元的类型：
    - 堆栈
    - 累加器
    - 通用寄存器组

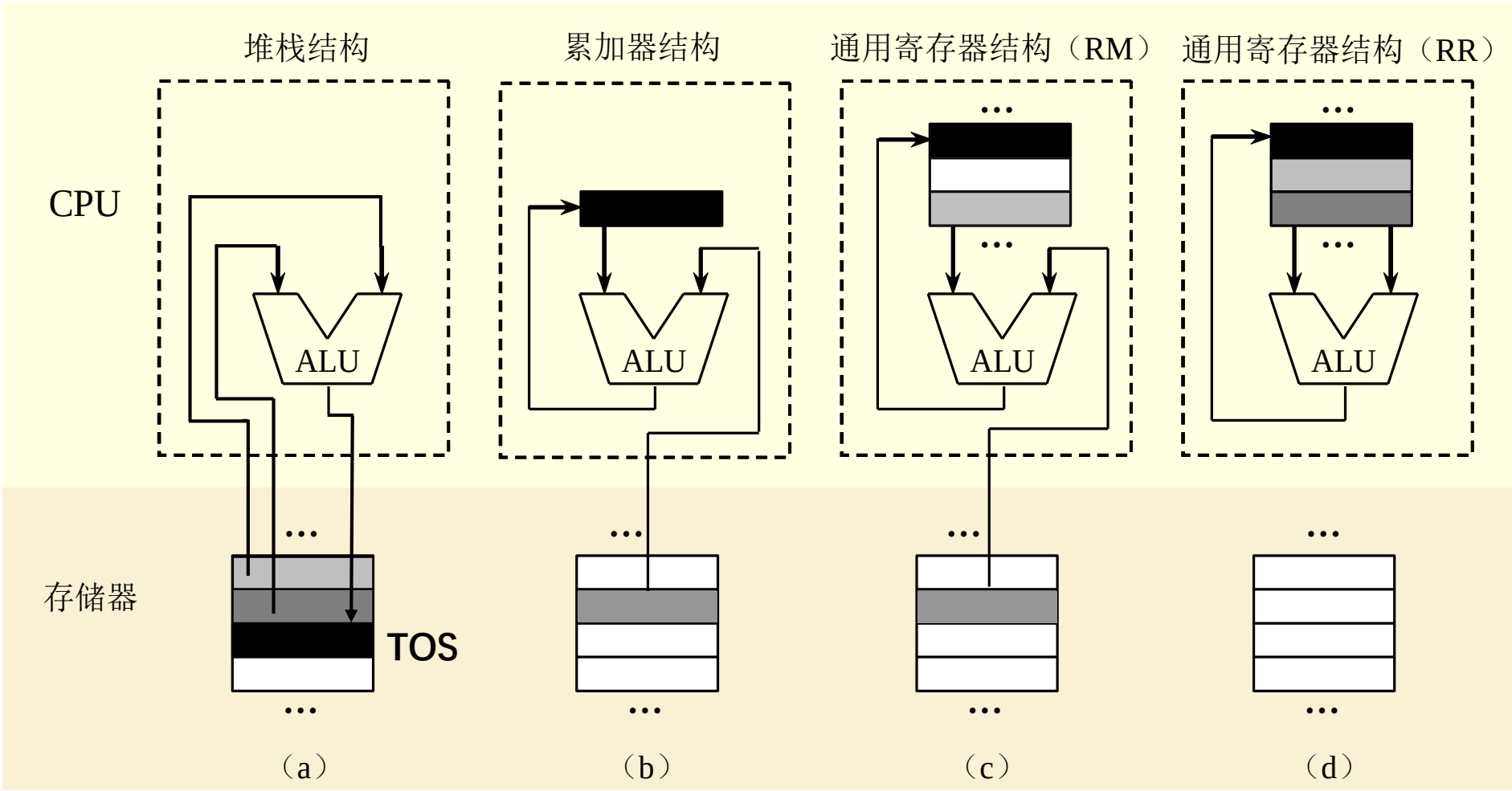
2.1 指令集体系结构的分类

- **指令集体系结构分为三种类型**
  - **堆栈结构**
  - **累加器结构**
  - **通用寄存器结构**（根据操作数的来源不同又可分为）
    - ✓**寄存器-存储器结构**（RM结构）：操作数可以来自存储器
    - ✓**寄存器-寄存器结构**（RR结构）：所有操作数均来自通用寄存器组
    - ✓**load-store结构**：只有load指令和store指令能够访问存储器

2.1 指令集体系结构的分类

- 对于不同类型的指令集体系结构，操作数的位置、个数以及操作数的给出方式（显式或隐式）会不同
  - **显式给出：用指令字中的操作数字段给出**
    - 操作数字段个数
    - 寻址方式
  - **隐式给出：使用事先约定好的存储单元**
    - 堆栈栈顶
    - 累加器
    - 特定寄存器
    - 特定存储单元

4种指令集结构的操作数的位置以及结果的去向



灰色块：操作数　　黑色块：结果　　TOS（Top Of Stack）：栈顶

2.1 指令集体系结构的分类

例: 表达式C=A+B在4种类型指令集体系结构上的代码

假设：A、B、C 均保存在存储器单元中，并且不能破坏A和B的值。

堆 栈	累加器	寄存器（RM型）	寄存器（RR型）
push A	load A	load R1, A	load R1, A
push B	add B	add R1, B	load R2, B
add	store C	store R1, C	add R3, R1, R2
pop C			store R3, C

2.1 指令集体系结构的分类

- **通用寄存器结构**
  - **现代指令集体系结构的主流**
  - **在灵活性和提高性能方面有明显的优势**
    - ✓ 寄存器的访问速度比存储器快
    - ✓ 对编译器，能更容易、有效地分配和使用寄存器
    - ✓ 寄存器可以用来存放变量：
      - 因为寄存器比存储器快，减少对存储器的访问，能加快程序的执行速度
      - 用更少的地址位（相对于存储器地址）来对寄存器进行寻址，从而有效地减少程序的目标代码的大小

2.1 指令集体系结构的分类

- 根据ALU指令的操作数的两个特征对通用寄存器型指令集体系结构进一步细分：
  - ALU指令的操作数个数　　ADD R_D, R_S, R_T
    - 3个操作数的指令：两个源操作数、一个目的操作数
    - 2个操作数的指令：　ADD R_D, R_S

      其中一个操作数既作为源操作数，又作为目的操作数
  - ALU指令的存储器操作数的个数

    可以是0～3中的某一个，为0表示没有存储器操作数

ALU指令中操作数个数和存储器操作数个数的典型组合

存储器操作数的个数	操作数的最多个数	结构类型	机器实例
0	3	RR	MIPS，SPARC，Alpha，PowerPC，ARM
1	2	RM	IBM 360/370，Intel 80x86，Motorola 68000
	3	RM	IBM 360/370
2	2	MM	Intel VAX
3	3	MM	Intel VAX

recap

1. **区别不同指令集结构的主要因素是什么?**

   CPU中用来存储操作数的存储单元类型，主要有通用寄存器、累加器、堆栈。

2. **寻址方式确定所要访问的哪些操作数的地址?**

   指明指令中的操作数是一个常数、一个寄存器操作数、一个存储器操作数

2.1 指令集体系结构的分类

- 通用寄存器型指令集体系结构进一步细分为3种类型：
  - 寄存器-寄存器型（RR型）
  - 寄存器-存储器型（RM型）
  - 存储器-存储器型（MM型）

- 3种通用寄存器型指令集体系结构的优缺点

  表中(*m*, *n*) 表示指令的*n* 个操作数中有*m* 个存储器操作数。

指令集结构类型	优 点	缺 点
寄存器－寄存器型（0，3）	指令字长固定，指令结构简洁，是一种简单的代码生成模型，各种指令的执行时钟周期数相近	与指令中含存储器操作数的指令集体系结构相比，指令条数多，目标代码不够紧凑，因而程序占用的空间比较大
寄存器－存储器型（1，2）	可以在ALU指令中直接对存储器操作数进行引用，而不必先用load指令进行加载。容易对指令进行编码，目标代码比较紧凑	指令中的两个操作数不对称。在一条指令中同时对寄存器操作数和存储器操作数进行编码，有可能限制指令所能够表示的寄存器个数。指令的执行时钟周期数因操作数的来源（寄存器或存储器）不同而差别比较大
存储器－存储器型（2，2）或（3，3）	目标代码最紧凑，不需要设置寄存器来保存变量	指令字长变化很大，特别是3操作数指令。而且每条指令完成的工作也差别很大。对存储器的频繁访问会使存储器成为瓶颈。这种类型的指令集体系结构现在已不用了

第2章 计算机指令集体系结构

2.1 指令集体系结构的分类

2.2 寻址方式

2.3 指令系统的设计和优化

2.4 操作数的类型和大小

2.5 指令格式的设计和优化

2.6 MIPS指令集体系结构

- 寄存器寻址
- 立即数寻址方式
- 偏移寻址方式

2.2 寻址方式

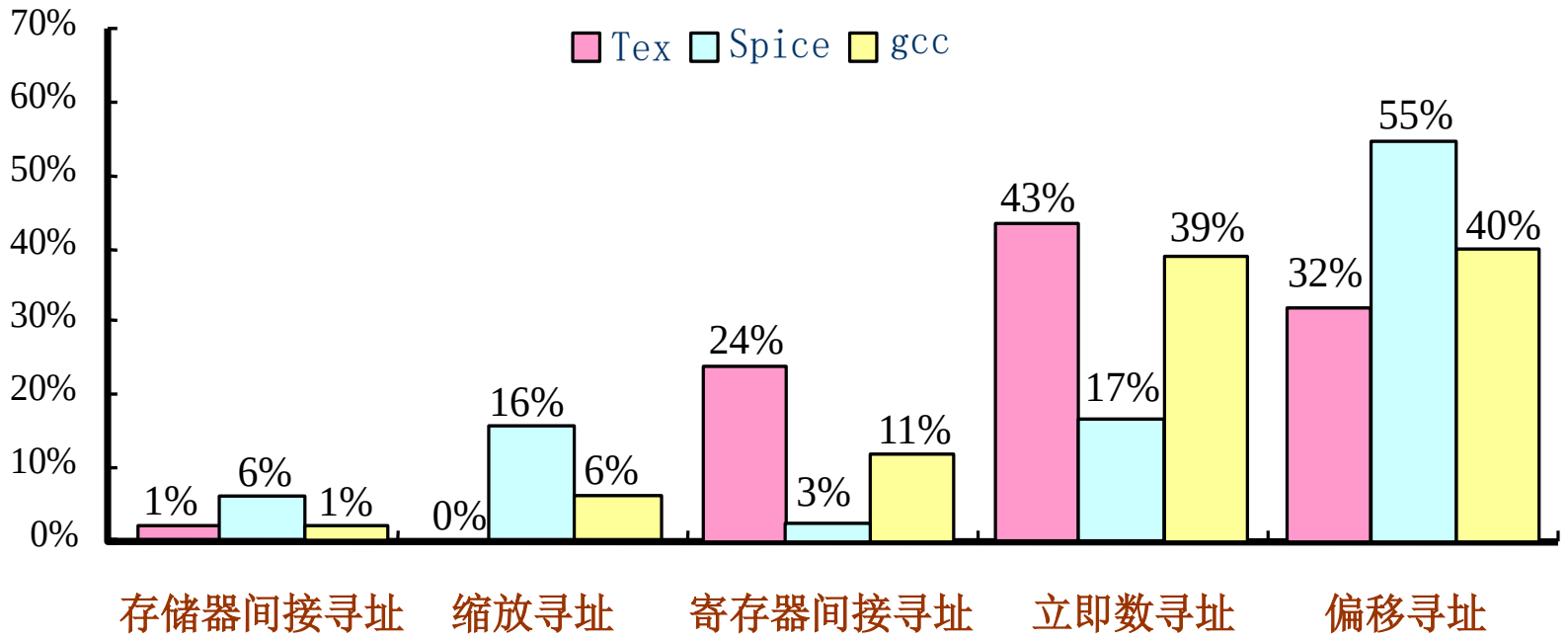
一种指令集体系结构如何确定所要访问的数据的地址?

- **当前指令集体系结构中所采用的一些操作数寻址方式**
  - **←：赋值操作**
  - **Mem：存储器**
  - **Regs：寄存器组**
  - **方括号：表示内容**
    - Mem[ ]：存储器的内容
    - Regs[ ]：寄存器的内容
    - Mem[Regs[R1]]：以寄存器R1中的内容作为地址的存储器单元中的内容

寻址方式	指令实例	含　义
寄存器寻址	Add R4 , R3	Regs[R4]←Regs[R4] + Regs[R3]
立即数寻址	Add R4 , #3	Regs[R4]←Regs[R4] + 3
偏移寻址	Add R4 , 100(R1)	Regs[R4]←Regs[R4] + Mem[100+Regs[R1]]
寄存器间接寻址	Add R4 , (R1)	Regs[R4]←Regs[R4] + Mem[Regs[R1]]
索引寻址	Add R3 , (R1 + R2)	Regs[R3]←Regs[R3] + Mem[Regs[R1]+Regs[R2]]
直接寻址或 绝对寻址	Add R1 , (1001)	Regs[R1]←Regs[R1] + Mem[1001]
存储器间接寻址	Add R1 , @(R3)	Regs[R1]←Regs[R1] + Mem[Mem[Regs[R3]]]
自增寻址	Add R1 , (R2)+	Regs[R1]←Regs[R1] + Mem[Regs[R2]] Regs[R2]←Regs[R2] + d
自减寻址	Add R1, -(R2)	Regs[R2]←Regs[R2] − d Regs[R1]←Regs[R1]+Mem[Regs[R2]]
缩放寻址	Add R1 , 100(R2)[R3]	Regs[R1]←Regs[R1] + Mem[100 + Regs[R2] + Regs[R3]*d]

2.2 寻址方式

- 采用**多种寻址方式**可以显著地减少程序的指令条数，但可能增加计算机的实现复杂度以及指令的CPI。



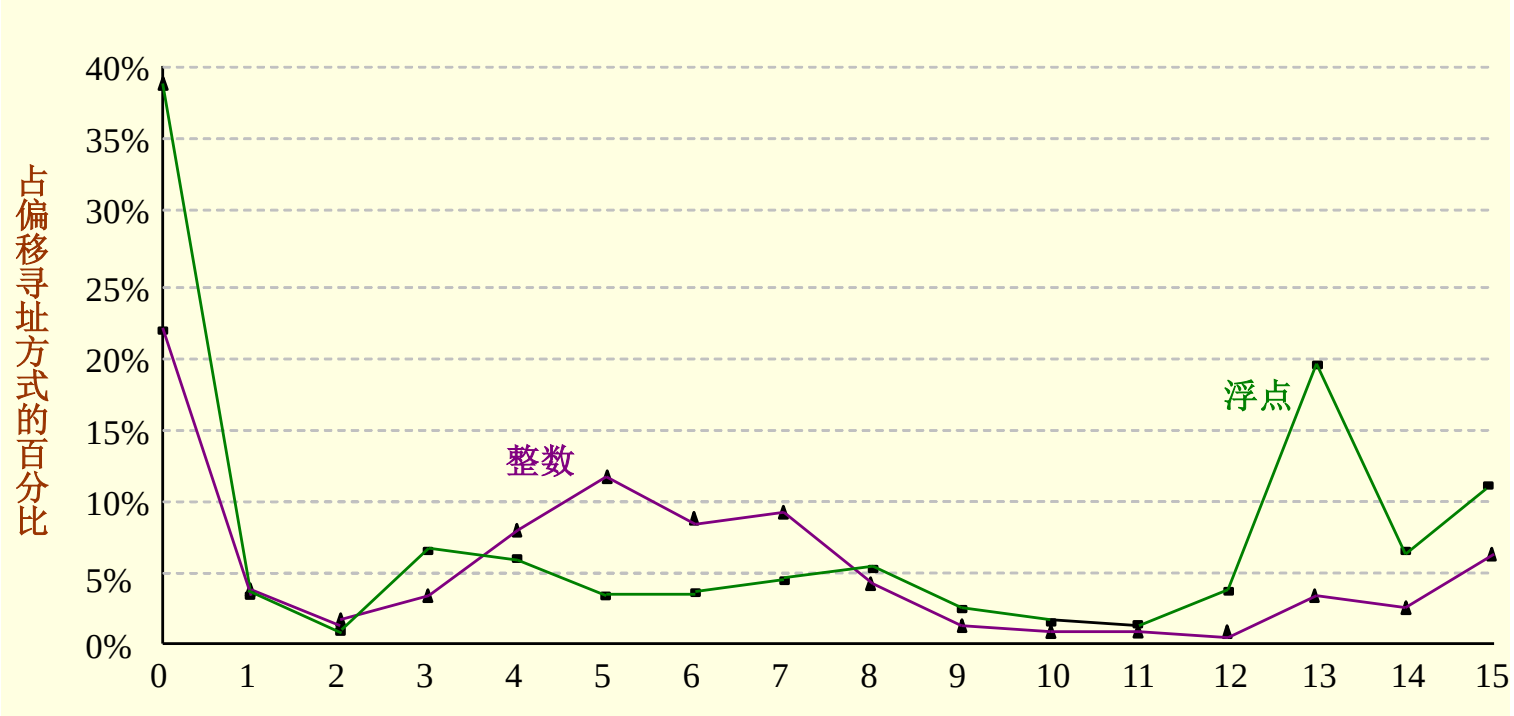
立即数寻址方式和**偏移寻址方式**的使用频度最高。

各种寻址方式的使用情况统计结果：在VAX机器上运行gcc、Spice和Tex基准程序

2.2 寻址方式

- **偏移量的取值范围**

在load-store结构的Alpha处理器上运行SPEC CPU2000基准程序



- **程序所使用的偏移量大小分布十分广泛：**主要是因为在存储器中所保存的数据并不是十分集中，需要使用不同的偏移量才能对其进行访问。
- **较小的偏移量和较大的偏移量均占有相当大的比例**

2.2 寻址方式

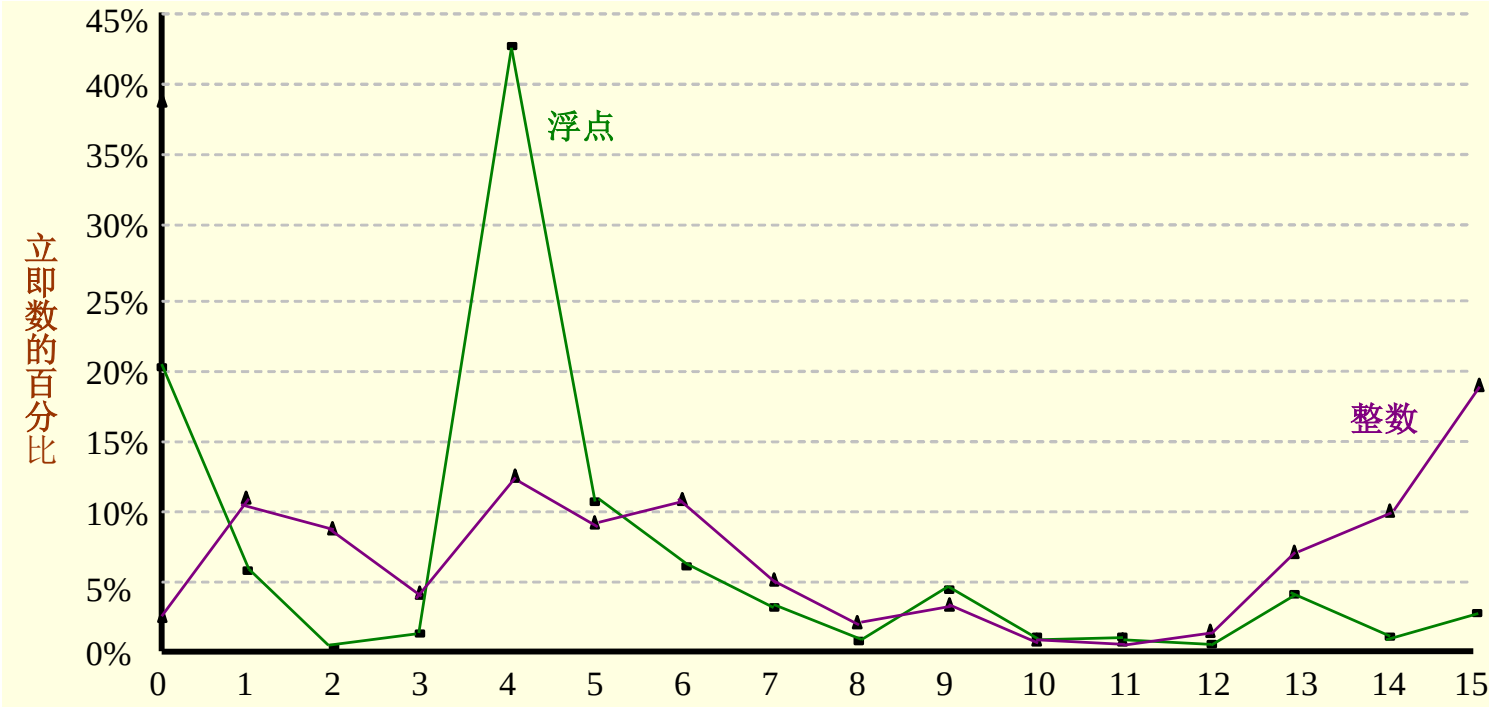
- 立即数寻址方式
  - 立即数寻址方式的使用频度

指令类型	使用频度	
	整型平均	浮点平均
load指令	23%	22%
ALU指令	25%	19%
所有指令	21%	16%

大约1/4的load指令和ALU指令采用了立即数寻址。

2.2 寻址方式

- **立即数的取值范围**



- 最常用的是**较小的立即数**;
- 有时也会用到较大的立即数（主要是用于地址计算）。
- 在指令集体系结构设计中，**至少要将立即数的大小**设置为**8～16**位
- 在VAX机（支持32位立即数）上做过类似的统计，结果表明**20%～25%**的立即数超过**16**位。

第2章 计算机指令集体系结构

2.1 指令集体系结构的分类

2.2 寻址方式

2.3 指令系统的设计和优化

2.4 操作数的类型和大小

2.5 指令格式的设计和优化

2.6 MIPS指令集体系结构

- 基本要求
- CISC指令集设计
- RISC指令集设计
- 控制指令

2.3 指令集体系结构的功能设计

- **指令集体系结构的功能设计**
  - **确定软、硬件功能分配：**即确定哪些基本功能应该由硬件实现，哪些功能由软件实现比较合适。
- **在确定哪些基本功能用硬件来实现时，主要考虑3个因素：速度、成本、灵活性**
  - **硬件实现的特点：**速度快、成本高、灵活性差
  - **软件实现的特点：**速度慢、价格便宜、灵活性好

2.3 指令集体系结构的功能设计

- **对指令集的基本要求：** 完整性、规整性、高效率、兼容性

（1）完整性：在一个有限可用的存储空间内，对于任何可解的问题，编制计算程序时，指令集所提供的指令足够用

- 要求**指令集功能齐全**、使用方便
- 下表为许多指令集体系结构都包含的一些指令类型
  - 前4类属于**通用计算机系统的基本指令**
  - 对于最后4种类型的操作，不同指令集体系结构的支持大不相同 。

2.3 指令集体系结构的功能设计

操作类型	实　例
算术和逻辑运算	算术运算和逻辑操作：加，减，乘，除，与，或等
数据传输	load，store
程序控制	分支，跳转，过程调用和返回，自陷等
操作系统支持	操作系统调用，虚拟存储器管理等
浮点	浮点操作：加，减，乘，除，比较等
十进制	十进制加，十进制乘，十进制到字符的转换等
字符串	字符串移动，字符串比较，字符串搜索等
图形	像素操作，压缩/解压操作等

2.3 指令集体系结构的功能设计

(2) **规整性：主要包括对称性和均匀性**

- **对称性：**所有与指令集有关的存储单元的使用、操作码的设置等都是对称的。

  例如：在存储单元的使用上所有通用寄存器都要同等对待。在操作码的设置上，如果设置了**A-B**的指令，就应该也设置**B-A**的指令。

- **均匀性：**指对于各种不同的操作数类型、字长、操作种类和数据存储单元，指令的设置都要同等对待。

  例如：如果某机器有**5**种数据表示，**4**种字长，**两种**存储单元，则要设置5×4×2=40种同一操作的指令。

2.3 指令集体系结构的功能设计

(3) **正交性：**指令中各个不同含义的字段在编码时应互不相关、相互独立

(4) **高效率：**指令的执行速度快、使用频度高

(5) **兼容性：**

- 向后兼容，可以适当增加指令、增加寻址方式，增加数据表示等；但不能减少任何已有的指令。

2.3 指令集体系结构的功能设计

- **设计指令集结构的两种截然不同的设计策略**

——产生了两类不同的计算机系统：

- **CISC（复杂指令集计算机）**

  增强指令功能，把越来越多的功能交由硬件来实现，并且指令的数量也是越来越多。

- **RISC（精简指令集计算机）**

  尽可能地**简化指令集**，不仅指令的条数少，而且指令的功能也比较简单。

2.3 指令集体系结构的功能设计

2.3.1 CISC指令集体系结构的功能设计

- CISC结构追求的目标：强化指令功能，减少程序的指令条数，以达到提高性能的目的。
- 增强指令功能主要是从以下几个方面着手：
  （1）面向目标程序增强指令功能
  （2）面向高级语言的优化实现改进指令集
  （3）面向操作系统的优化实现改进指令集

2.3 指令集体系结构的功能设计

2.3.1 CISC指令集体系结构的功能设计

- **增强指令功能的方法：**

（1）面向目标程序增强指令功能

对大量的目标程序及其执行情况进行统计分析，找出使用频度高、执行时间长的指令或指令串，指令用硬件加快其执行；指令串用一条新的指令来替代

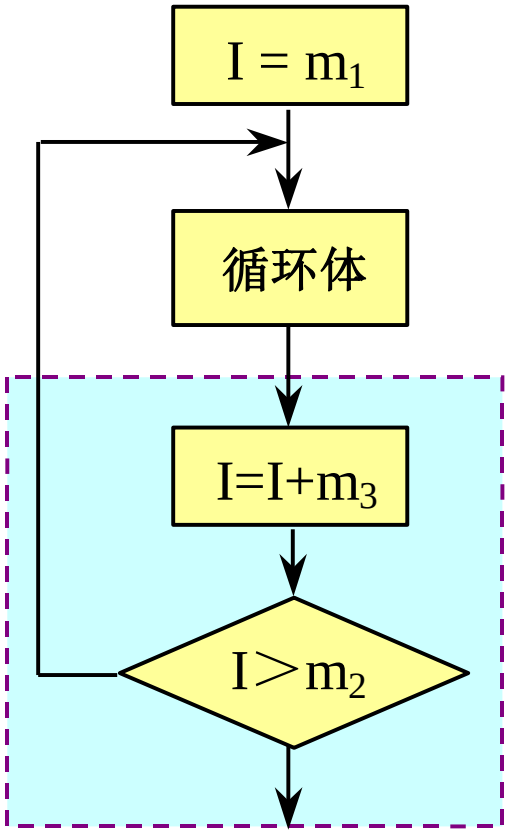
- 增强运算型指令的功能：sin( ), e^x
- 增强数据传送指令的功能
- 增强程序控制指令的功能
- 丰富的程序控制指令为编程提供了多种选择

2.3.1 CISC指令集体系结构的功能设计

例：循环在程序中占有相当大的比例，所以在指令上提供专门的支持。

- ✓ 循环控制部分通常用3条指令完成：
  - 一条加法指令
  - 一条比较指令
  - 一条分支指令
- ✓ 设置循环控制指令，用一条指令完成上述3条指令的功能

```
beq    $t0, $zero, L1
```



一般循环程序的结构

2.3.1 CISC指令集体系结构的功能设计

（2）面向高级语言的优化实现来改进指令集

➢ 缩小高级语言与机器语言的语义差距

高级语言与一般的机器语言的语义差距非常大，为高级语言程序的编译带来了一些问题。

- ✓ 编译器本身比较复杂。
- ✓ 编译生成的目标代码比较难以达到很好的优化。

2.3.1 CISC指令集体系结构的功能设计

- **增强对高级语言和编译器的支持**
  - ✓ **对源程序中各种高级语言语句的使用频度进行统计与分析：** 对使用频度高、执行时间长的语句，增强有关指令的功能，加快这些指令的执行速度，或者增加专门的指令，可以达到减少目标程序的执行时间和减少目标程序长度的目的。
  - ✓ **增强系统结构的规整性，减少体系结构中的各种例外情况**

（面向高级语言的计算机）

2.3.1 CISC指令集体系结构的功能设计

✓ **高级语言计算机**

- **间接执行高级语言机器**

  用汇编的方法把高级语言源程序翻译成机器语言程序。

- **直接执行高级语言的机器**

  直接把高级语言作为机器语言，直接由固件/硬件对高级语言源程序的语句逐条进行解释执行。这时既不用编译，也不用汇编。

2.3.1 CISC指令集体系结构的功能设计

（3）面向操作系统的优化实现改进指令集

- 操作系统和计算机体系结构是紧密联系的，操作系统的实现在很大程度上取决于指令集结构的支持。
- 指令集对操作系统的支持主要有：
  - ✓ 处理机工作状态和访问方式的切换
  - ✓ 进程的管理和切换
  - ✓ 存储管理和信息保护
  - ✓ 进程的同步与互斥，信号灯的管理等

  支持操作系统的有些指令属于特权指令，一般用户程序是不能使用的。

2.3.1 CISC指令集体系结构的功能设计

复杂指令集计算机（CISC）

- **特点：**

✓ 指令的控制执行是采用微程序控制技术，有专用的寄存器。

✓ 控制器十分复杂，占用了大量CPU芯片面积，有些复杂指令用的很少，难以用优化编译生成高效目标代码。

✓ 处理器的执行效率不高。

✓ 指令系统与软件之间语义差别越来越大，软件设计任务十分繁重，整个设计风格不是十分经济有效的。

- **缺点：**

指令系统庞大 → 硬件复杂 → 执行速度低

→ 编译程序复杂、长 → 部分指令使用效率低

2.3.1 CISC指令集体系结构的功能设计

70年代，指令系统已经非常复杂

机　　型 （生产年代）	IBM370/168 （1973）	VAX-11 （1978）	iAPX 432 （1982）	Dorado （1978）
指令种类	208	303	222	270
微程序容量	420K	480K	64K	136K
指令长度	16-48	16-456	6-321	8-24
采用的工艺	ECL MSI	TTL MSI	NMOS VLSI	ECL MSI
指令操作类型	存储器-存储器 存储器-寄存器 寄存器-寄存器	存储器-存储器 存储器-寄存器 寄存器-寄存器	面向堆栈 存储器-存储器	面向堆栈
Cache 容量	64KB	64KB	0	64KB

2.3 指令集体系结构的功能设计

2.3.2 RISC指令集体系结构的功能设计

- CISC指令集体系结构存在的问题：
  - 各种指令的使用频度相差悬殊
    - 据统计：只有20%的指令使用频度比较高，占运行时间的80%，而其余80%的指令只在20%的运行时间内才会用到
    - 使用频度高的指令也是最简单的指令

Intel 80x86最常用的10条指令

执行频度排序	80x86指令	指令执行频度（占执行指令总数的百分比）
1	load	22%
2	条件分支	20%
3	比较	16%
4	store	12%
5	加	8%
6	与	6%
7	减	5%
8	寄存器-寄存器间数据移动	4%
9	调用子程序	1%
10	返回	1%
合 计		95%

2.3.2 RISC指令集体系结构的功能设计

- **指令集庞大，指令条数很多，许多指令的功能又很复杂，使得控制器硬件非常复杂，导致的问题：**
  - 占用了大量的芯片面积（如CPU芯片面积一半以上），给VLSI设计造成很大的困难；
  - 增加了研制时间和成本，容易造成设计错误。
- **许多指令操作繁杂，CPI值比较大，执行速度慢。采用这些复杂指令有可能使整个程序的执行时间反而增加**
- **由于指令功能复杂，规整性不好，不利于采用流水技术来提高性能**

2.3.2 RISC指令集体系结构的功能设计

- 1975年，IBM公司率先组织力量开始研究指令系统的合理性问题
- 1979年研制出世界上第一台采用RISC思想的计算机IBM 801
- 1981年，Stanford大学的MIPS，MIPS Rxxx系列微处理器
- 1986年，IBM正式推出采用RISC体系结构的工作站IBM RT PC

➢ 共同特点：

- 采用load-store结构
- 指令字长为32位
- 采用高效的流水技术

2.3.2 RISC指令集体系结构的功能设计

- 设计RISC机器遵循的原则

- 指令条数少而简单。只选取使用频度很高的指令，在此基础上补充一些最有用的指令
- 采用简单而又统一的指令格式，并减少寻址方式；指令字长都为32位或64位
- 指令的执行在单个机器周期内完成。(采用流水线机制)
- 采用load-store结构：只有load和store指令才能访问存储器，其他指令的操作都是在寄存器之间进行
- 大多数指令都采用硬连逻辑来实现
- 强调优化编译器的作用，为高级语言程序生成优化代码
- 充分利用流水技术来提高性能

2.3.2 RISC指令集体系结构的功能设计

- **早期的RISC微处理器**
  - **1981年，Berkeley分校的Patterson 等人的32位微处理器RISC I：**
    - 31条指令，指令字长都是32位，78个通用寄存器，时钟频率为8 MHz
    - 控制部分所占的芯片面积只有约6%。商品化微处理器MC68000和Z8000分别为50%和53%
    - 性能比MC68000和Z8000快3 ~ 4倍
  - **1983年的RISC II：**
    - 39条指令，138个通用寄存器，时钟频率为12 MHz
    - 后来发展成了Sun公司的SPARC系列微处理器
  - **2014年的RISC V：** 40条指令，32个通用寄存器

2.3.2 RISC指令集体系结构的功能设计

- RISC思想的精华：减少CPI
  - 程序执行时间： Time = IC · CPI · T

CISC 与 RISC 的运算速度比较

类 型	指令条数 I	指令平均周期数 CPI	周期时间 T
CISC	1	2～15	33ns～5ns
RISC	1.3～1.4	1.1～1.4	10ns～2ns

  - 同类问题的程序长度：RISC比CISC长30%-40%
  - CPI：RISC比CISC小2倍-10倍
  - 速度：RISC要比CISC快3倍左右

2.3.2 RISC指令集体系结构的功能设计

- **RISC设计思想也可以用于CISC中**
  - **X86处理机的CPI在不断缩小，**
    - 8088的CPI大于20
    - 80286的CPI大约是5.5
    - 80386的CPI进一步减小到4左右
    - 80486的CPI已经接近2
    - Pentium处理机的CPI已经与RISC十分接近
  - **目前，超标量处理机、超流水线处理机的CPI已经达到0.5，实际上用 IPC (Instruction Per Cycle)更确切。**

第2章 计算机指令集体系结构

2.1 指令集体系结构的分类

2.2 寻址方式

2.3 指令系统的设计和优化

2.4 操作数的类型和大小

2.5 指令格式的设计和优化

2.6 MIPS指令集体系结构

1. 指令集功能设计
2. RISC指令集设计
3. 控制指令
   - 分支、跳转
   - 过程调用、返回

2.3 指令集体系结构的功能设计

2.3.3 控制指令

- 控制指令是改变控制流的指令
  - 跳转：当指令是无条件改变控制流时，称为跳转指令
  - 分支：当指令是有条件改变控制流时，则称为分支指令
- 能够改变控制流的指令
  - 分支
  - 跳转
  - 过程调用
  - 过程返回

2.3.3 控制指令

- 控制指令的使用频度

（load-store型指令集结构的机器，基准程序为SPEC CPU2000）

指令类型	使用频度	
	整型平均	浮点平均
调用/返回	19%	8%
跳转	6%	10%
分支	75%	82%

改变控制流的大部分指令是分支指令（条件转移）

常用的3种表示分支条件的方法及其优缺点

名 称	检测分支条件的方法	优 点	缺 点
条件码 (CC)	检测由ALU操作设置的一些特殊的位（即CC）	可以自由设置分支条件	条件码是增设的状态。而且它限制了指令的执行顺序，因为要保证条件码能顺利地传送给分支指令
条件寄存器	比较指令把比较结果放入任何一个寄存器，检测时就检测该寄存器	简单	占用了一个寄存器
比较与分支	比较操作是分支指令的一部分，通常这种比较是受到一定限制的	用一条指令（而不是两条）就能实现分支	当采用流水方式时，该指令的操作可能太多，在一拍内做不完

2.3.3 控制指令

- **转移目标地址的表示**
  - **最常用的方法：** **PC相对寻址**

    在指令中提供一个偏移量，由该偏移量和程序计数器（PC）的值相加而得出目标地址。
  - **优点**
    - 有效减少表示该目标地址所需要的位数
    - 位置无关（代码可被装载到主存的任意位置执行）
  - **关键：确定偏移量字段的长度**
    - 模拟结果表明：采用**4～8位**的偏移量字段（以指令字为单位）就能表示大多数控制指令的转移目标地址了

2.3.3 控制指令

- **分支指令：PC相对寻址**

  - 在符号汇编语句中,分支语句的目标位置是用绝对地址方式写的
    - e.g., beq $0,$0,fact

      means PC ← 0x00400100
  - 不过在实现中,要用相对于PC的地址来定义
    - e.g., beq $0,$0,0x3f

      means PC ← 0x00400100

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

2.3.3 控制指令　● 分支指令：PC相对寻址

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

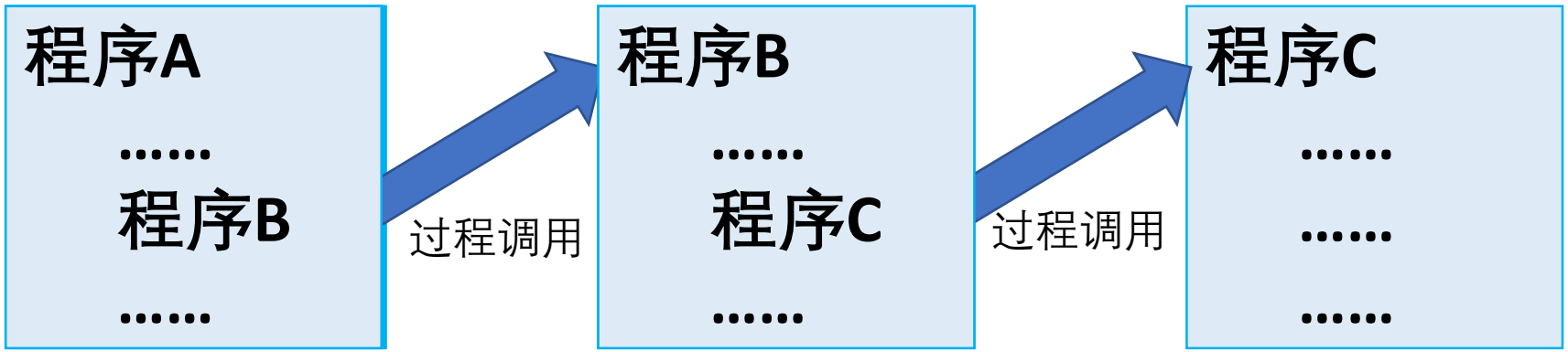
2.3 指令集体系结构的功能设计

2.3.3 控制指令

- **过程调用和返回**
  - 除了要改变控制流之外，可能还要保存机器状态，至少得保存返回地址（放在专用的链接寄存器或堆栈中）。
  - 过去有些指令集体系结构提供了专门的保存机制来保存许多寄存器的内容。
  - 现在较新的指令集体系结构则要求由编译器生成load和store指令来保存或恢复寄存器的内容。

2.3.3 控制指令 ● 过程调用和返回

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



2.3.3 控制指令 ● 过程调用和返回

- 子程序调用通过跳转与链接指令 jal 进行

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

- 子程序返回通过寄存器跳转指令 jr 进行

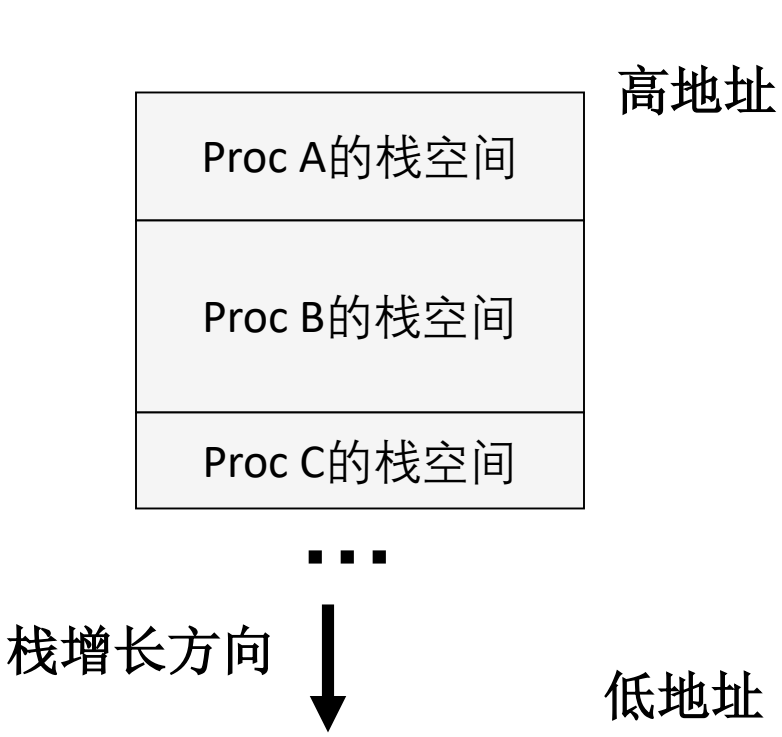
```
jr $ra             # 跳转到寄存器指定的地址
```

jal 和 j 的区别：jal将跳转的地址写入$ra

2.3.3 控制指令 ● 过程调用和返回

➢ 嵌套过程调用

上层程序调用下层程序时，将自己的变量压栈保存



Proc A

调用 Proc B

调用 Proc C

从Proc C返回

从Proc B返回

从Proc A返回

2.3.3 控制指令

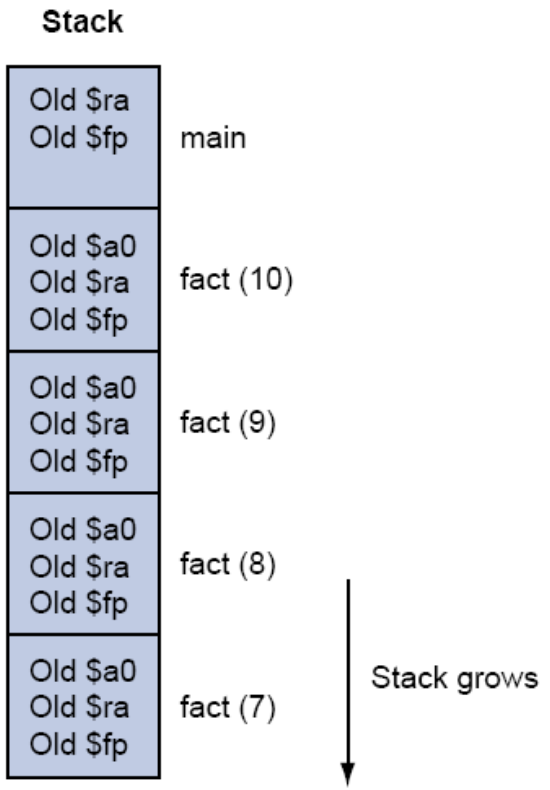
● 过程调用和返回

➢ 嵌套过程调用

```
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
```

• 计算n!

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



➢ 嵌套过程调用

- 计算n!

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

第2章 计算机指令集体系结构

2.1 指令集体系结构的分类

2.2 寻址方式

2.3 指令系统的设计和优化

2.4 操作数的类型和大小

2.5 指令格式的设计和优化

2.6 MIPS指令集体系结构

- 类型
- 大小
- 访问频度

2.4 操作数的类型和大小

- **数据表示：计算机硬件能够直接识别、指令集可以直接调用的数据类型**
  - 所有数据类型中最常用、相对比较简单、用硬件实现比较容易的几种：定点整数、浮点实数、逻辑数、字符
- **数据结构：由软件进行处理和实现的各种数据类型**
  - 研究：这些数据类型的逻辑结构与物理结构之间的关系，并给出相应的算法

系统结构设计者要解决的问题：如何确定数据表示？

（软硬件取舍折中的问题）

2.4 操作数的类型和大小

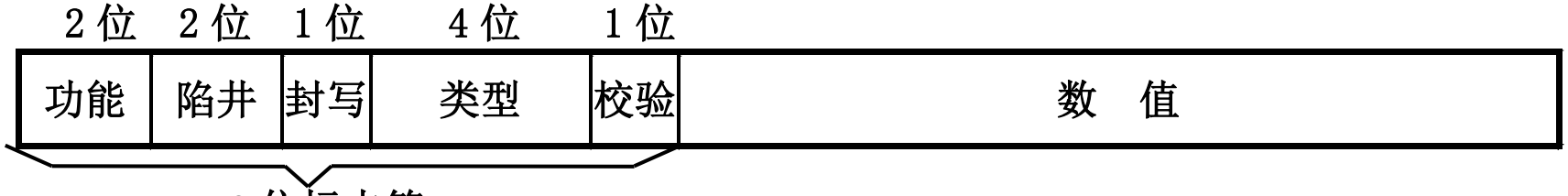
1. 表示操作数类型的方法（有两种）

- **由指令中的操作码指定操作数的类型**
  - **最常用的方法**
- **带标志符的数据表示**
  - **给数据加上标识，由数据本身给出操作数类型。**
  - **优点：简化指令集，**可由硬件自动实现一致性检查和类型转换，缩小了机器语言与高级语言的语义差距，简化编译器等
  - **缺点：**由于需要在执行过程中动态检测标志符，动态开销比较大，所以**采用这种方案的机器很少见**

2.4 操作数的类型和大小

- 在B5000大型机中，每个数据有一位标志符
- 在B6500和B7500大型机中，每个数据有三位标志符
- 在R-2巨型机中采用10位标志符

带有标志符的数据表示方式



在 R-2 巨型机中带标志符的数据表示方式

2.4 操作数的类型和大小

- R-2巨型机中的标志符

  功能位2位：操作数、指令、地址、控制字

  陷井位2位：由软件定义四种捕捉方式

  封写位1位：指定数据是只读的还是可读可写

  类型位4位：二进制,十进制,定点数,浮点数,复数,字符串,单精度,双精度；绝对地址、相对地址、变址地址、未连接地址等

  校验位1位：奇偶校验位

- 标志符由编译器或其它系统软件建立，对程序员透明

  程序（包括指令和数据）的存储量分析: 数据存储量增加，指令存储量减少

2.4 操作数的类型和大小

例1：假设X处理机的数据不带标志符，其指令字长和数据字长均为32位；Y处理机的数据带标志符，数据字长增加至35位，其中3位是标志符，其指令字长由32位减少至30位。并假设一条指令平均访问两个操作数，每个操作数平均被访问R次。分别计算这两种不同类型的处理机中程序所占用的存储空间。

解：X和Y处理机程序占用的存储空间总和分别为：

$$B_X = 32I + \frac{2\times 32I}{R} \qquad B_Y = 30I + \frac{2\times 35I}{R}$$

✓程序占用存储空间的比值：

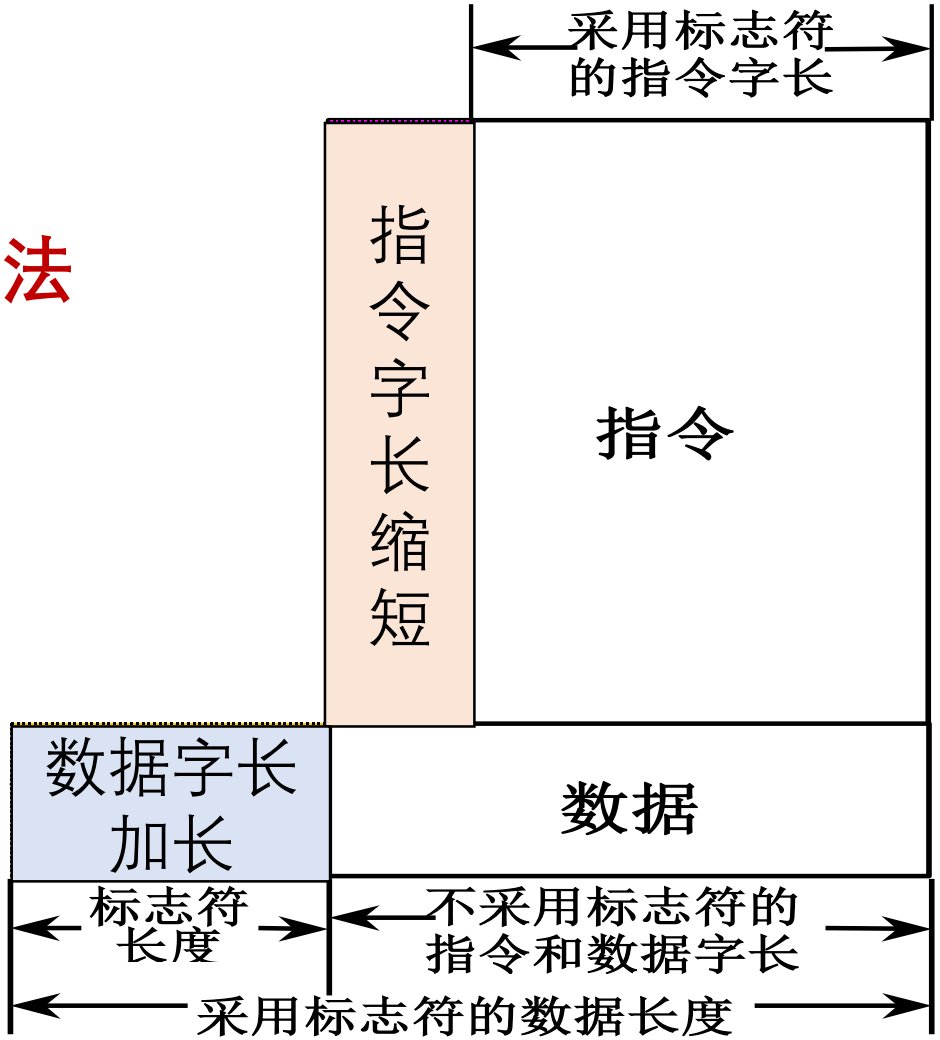
$$\frac{B_Y}{B_X}=\frac{30I+\frac{2\times35I}{R}}{32I+\frac{2\times32I}{R}}=\frac{15R+35}{16R+32}$$

当R＞3时，有 $\frac{B_Y}{B_X}<1$

在实际应用中经常是 R＞10，即带标志符的处理机所占用的存储空间通常要小。

2.4 操作数的类型和大小

常规数据表示方法与
带标志符数据表示方法
的比较



2.4 操作数的类型和大小

2. 操作数的大小：操作数的位数或字节数

主要的大小：字节（8位）、半字（16位）、字（32位）、双字（64位）

- **字符：**用ASCII码表示，为一个字节大小。
- **整数：**用二进制补码表示，其大小可以是字节、半字或单字。
- **浮点操作数：**单精度浮点数（1个字）、双精度浮点数（双字）。一般都采用IEEE 754浮点标准
- **十进制操作数类型**
  - 压缩十进制（二进制编码十进制BCD码）：用4位二进制编码表示数字0～9，并将两个十进制数字合并到一个字节中存储
  - 非压缩十进制：**将十进制数直接用字符串来表示**

2.4 操作数的类型和大小

3. 访问不同操作数大小的频度（SPEC基准程序）

操作数大小	访问频度	
	整型平均	浮点平均
字节	7%	0%
半字	19%	0%
单字	74%	31%
双字	0%	69%

- 基准程序对单字和双字的数据访问具有较高的频度

一台32位的机器应该支持8、16、32位整型操作数以及32位和64位的IEEE 754标准的浮点操作数。

第2章 计算机指令集体系结构

2.1 指令集体系结构的分类

2.2 寻址方式

2.3 指令系统的设计和优化

2.4 操作数的类型和大小

2.5 指令格式的设计和优化

2.6 MIPS指令集体系结构

- 操作码的编码方法
  - Huffman编码
  - 扩展编码
  - 定长编码
  - 变长编码
- 表示寻址方式的方法

2.5 指令格式的设计和优化

1. 指令由两部分组成：**操作码、操作数/地址码**

2. 指令格式的设计：确定指令字的编码方式

   包括**操作码字段**和**地址码字段**的编码及其表示方式

3. 操作码的编码比较简单和直观

   - **Huffman编码：**减少操作码的平均位数，但所获得的编码是变长的，不规整，不利于硬件处理
   - **固定长度的操作码：**保证操作码的译码速度。

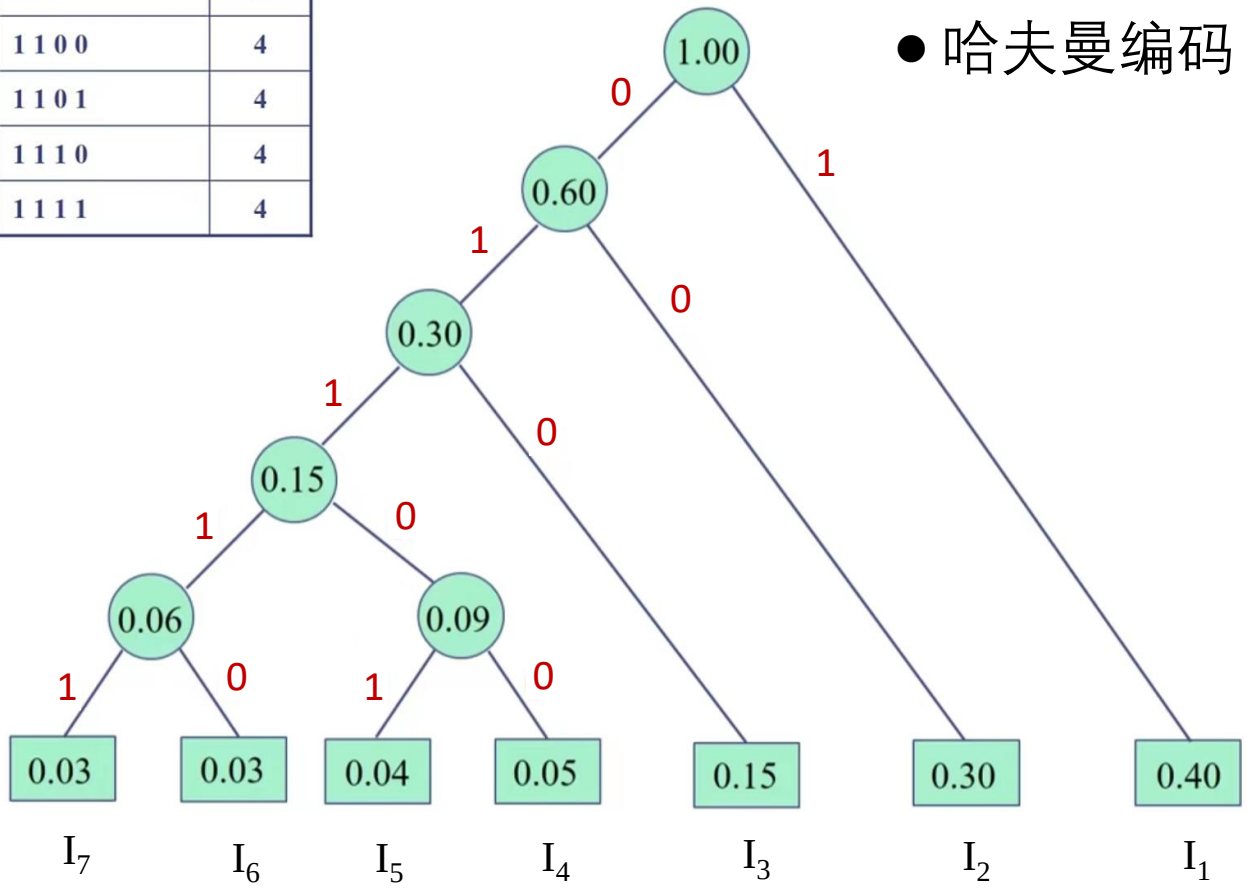
2.5 指令格式的设计和优化

指令	频度p_i	操作码使用哈夫曼编码	操作码长度l_i	利用哈夫曼概念的扩展操作码	操作码长度l_i
I_1	0.40	0	1	0 0	2
I_2	0.30	1 0	2	0 1	2
I_3	0.15	1 1 0	3	1 0	2
I_4	0.05	1 1 1 0 0	5	1 1 0 0	4
I_5	0.04	1 1 1 0 1	5	1 1 0 1	4
I_6	0.03	1 1 1 1 0	5	1 1 1 0	4
I_7	0.03	1 1 1 1 1	5	1 1 1 1	4

最短平均码长：

$$H = -\sum_{i=1}^{7} p_i \log_2 p_i = 2.17$$

● 哈夫曼编码



平均码长：

$$\mathrm{L} = \sum_{i=1}^{7} p_i l_i = 2.20$$

信息冗余量：

$$\frac{2.20-2.17}{2.20} \approx 1.36\%$$

2.5 指令格式的设计和优化

- **哈夫曼编码缺点：**变长度，不规整，不利于硬件处理
- **扩展操作码**
  - ✓ 位于定长二进制编码和哈夫曼编码之间的一种编码方案。
  - ✓ 采用有限几种固定长度的码长，仍然采用高概率的用短码、低概率用长码的哈夫曼压缩思想，使操作码平均长度缩短。
  - ✓ 上表中的指令，采用**2-4**的扩展操作码，可以得到如表右边所示的编码方案。
  - ✓ 用两位的00、01、10分别用于表示使用频度高的I_1、I_2、I_3，然后用11作为高位扩展出4个4位的二进制编码，用于表示剩下的4条指令。

平均码长： $\sum_{i=1}^{7} p_i l_i = 2.3$

2.5 指令格式的设计和优化

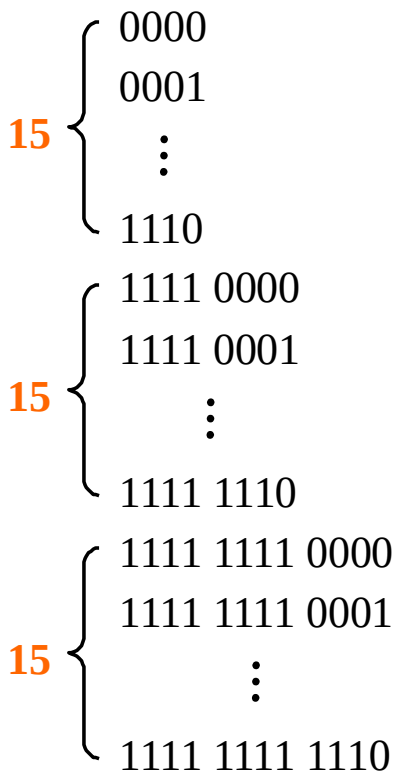
- **等长扩展码**：便于分级译码（早期）

15/15/15法和8/64/512法

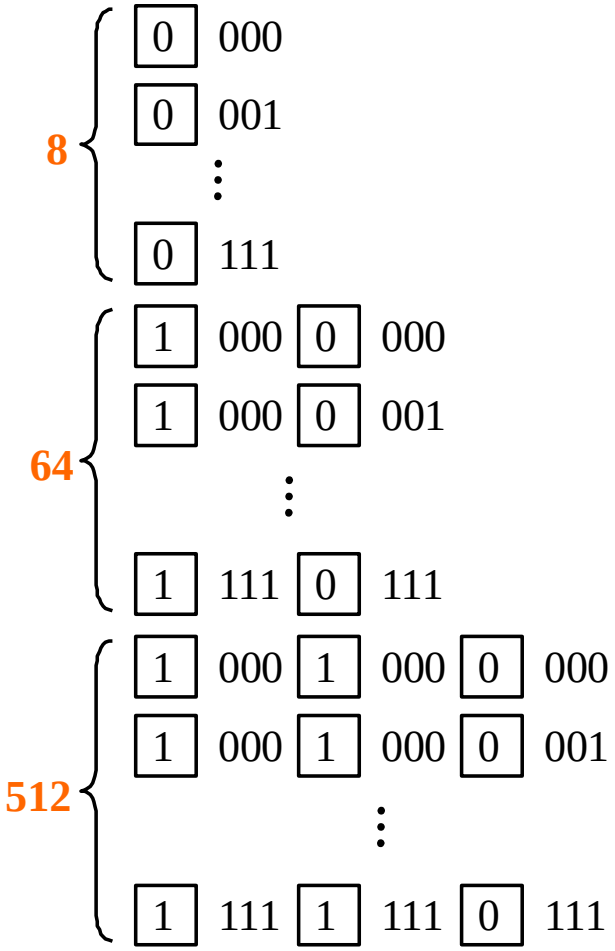
- ✓ 选用哪种编码法取决于指令使用频度p_i的分布
- ✓ 若在头15种指令中p_i的值都比较大，但在后30种指令后急剧减少，则应选择15/15/15法；
- ✓ 若p_i的值在头8种指令中较大，之后的64种指令的p_i值也不太低，则应选择8/64/512法。
- ✓ 衡量标准：看哪种编码法能使平均码长最短。

2.5 指令格式的设计和优化

- 等长扩展码



15/15/15 编码法



8/64/512 编码法

2.5 指令格式的设计和优化

- **定长编码**

  - ✓ 所有指令的操作码都是同一的长度（如8位）

    许多计算机都采用（特别是RISC结构的计算机）
  - ✓ 保证操作码的译码速度、减少译码的复杂度
  - ✓ 以程序的存储空间为代价来换取硬件实现上的好处

2.5 指令格式的设计和优化

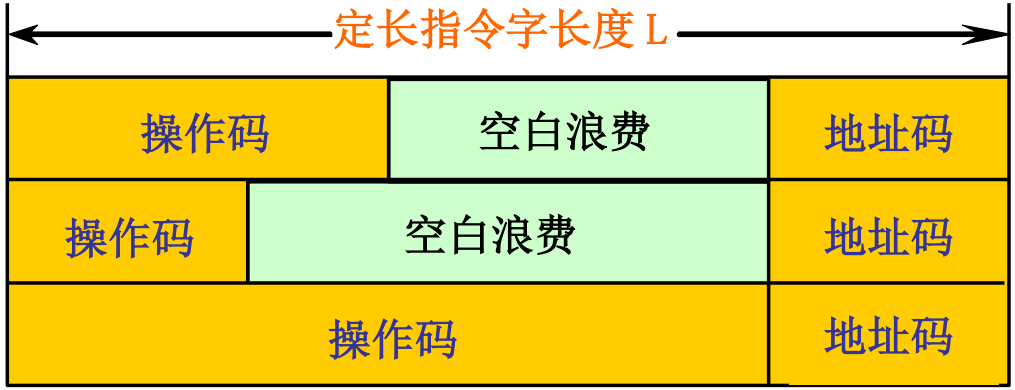
- **定长编码**

  ✓ 所有指令的操作码都是同一的长度（如8位）

  许多计算机都采用（特别是RISC结构的计算机）

  ✓ 保证操作码的译码速度、减少译码的复杂度

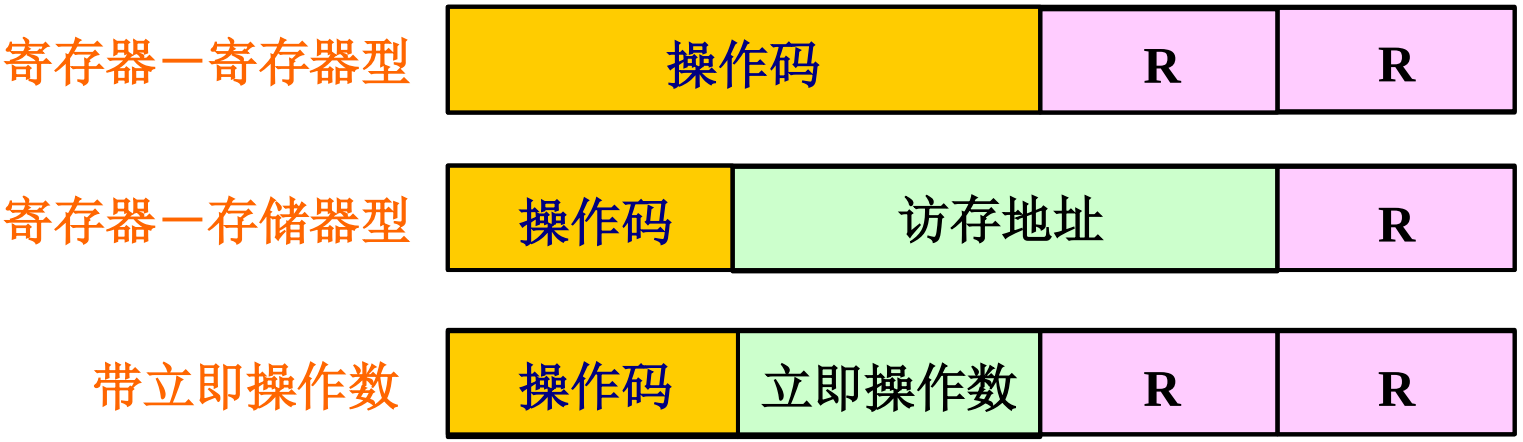
  ✓ 以程序的存储空间为代价来换取硬件实现上的好处



2.5 指令格式的设计和优化

- **采用地址个数可变和/或地址码长度可变的方案优化**
  - ✓ 利用操作码缩短所带来的好处
  - ✓ 最常用的操作码最短，其地址字段个数最多

    能够使指令的功能增强，从总体上减少所需的指令条数



2.5 指令格式的设计和优化

3. 考虑因素

- 机器中寄存器的个数和寻址方式的数目对指令平均字长的影响以及它们对目标代码大小的影响
- 所设计的指令格式便于硬件处理，特别是流水实现
- 指令字长应该是字节（8位）的整数倍，而不能是随意的位数

4. 指令集的3种编码格式

- 变长编码格式、定长编码格式、混合型编码格式

2.5 指令格式的设计和优化

- **变长编码格式**
  - 当指令集的寻址方式和操作种类很多时，这种编码格式是最好的。（VAX, X86）

操作码	地址描述符 1	地址码 1	…	地址描述符 n	地址码 n

优点：用最少的二进制位来表示目标代码

缺点：可能会使各条指令的字长和执行时间相差很大

2.5 指令格式的设计和优化

- **定长编码格式** （MIPS， Power PC）
  - 将操作类型和寻址方式一起编码到操作码中
  - 当寻址方式和操作类型非常少时，这种编码格式非常好
  - 可以有效地降低译码的复杂度，提高译码的速度
  - 大部分RISC的指令集均采用这种编码格式

操作码	地址码 1	地址码 2	地址码 3

2.5 指令格式的设计和优化

- **混合型编码格式** （IBM 360/370）
  - 提供若干种固定的指令字长
  - 以期达到既能够减少目标代码长度，又能降低译码复杂度的目标

第2章 计算机指令集体系结构

2.1 指令集体系结构的分类

2.2 寻址方式

2.3 指令系统的设计和优化

2.4 操作数的类型和大小

2.5 指令格式的设计和优化

2.6 MIPS指令集体系结构

- 寄存器
- 数据表示
- 寻址方式
- 指令格式
- MIPS操作
- 控制指令
- 浮点操作

https://www.mips.com/products/architectures/mips64/

2.6 MIPS指令集体系结构

介绍 MIPS 64 的一个子集，简称为MIPS

2.6.1 MIPS的寄存器

- 32个64位通用寄存器（GPRs）
  - R0，R1，…，R31
  - 也被称为整数寄存器
  - R0的值永远是0
- 32个64位浮点数寄存器（FPRs）
  - F0，F1，…，F31
  - 存放32个单精度浮点数（32位），只用到FPR的一半
  - 也可以存放32个双精度浮点数（64位）
- 一些特殊寄存器
  - 可以与通用寄存器交换数据。
  - 浮点状态寄存器用来保存有关浮点操作结果的信息

2.6 MIPS指令集体系结构

2.6.2 MIPS的数据表示

- MIPS的数据表示
  - **整数** 字节（8位） 半字（16位）
    字（32位） 双字（64位）
  - **浮点数**
    单精度浮点数（32位） 双精度浮点数（64位）
- **字节、半字或者字在装入64位寄存器时，用零扩展或者用符号位扩展来填充该寄存器的剩余部分。装入以后，对它们将按照64位整数的方式进行运算。**

2.6 MIPS指令集体系结构

2.6.3 MIPS的数据寻址方式

- 只有立即数寻址与偏移量寻址两种寻址

  立即数字段和偏移量字段都是16位的。

```
lw  $s0,0($s5)
```

- 寄存器间接寻址是通过把0作为偏移量来实现的
- 16位绝对寻址是通过把R0（其值永远为0）作为基址寄存器来完成的

```
lw  $s0,100($0)
```

- MIPS的存储器是按字节寻址的，地址为64位
- 所有存储器访问都必须是边界对齐的

2.6 MIPS指令集体系结构

2.6.4 MIPS 32 的指令格式

- 寻址方式编码到操作码中
- 所有的指令都是32位的
- 操作码占6位
- 3种指令格式

4.10 Instruction Fetch

4.10.1 Instruction Fields

For MIPS instructions, the layout of the bit fields in instructions is little-endian, regardless of the endianness mode in which the processor is executing. Bit 0 of an instruction is always the right-most bit in the instruction, while bit 31 is always the left-most bit in the instruction. The major opcode is always the left-most 6 bits in the instruction.

4.10.2 MIPS32 and MIPS64 Instruction Placement and Endianness

For the MIPS32 and MIPS64 base architectures, instructions are always 32 bits. All instructions are naturally aligned in memory (address bits 1:0 are 0b00).

2.6 MIPS指令集体系结构

- **load指令：** lw r1, 100(r2)

  访存有效地址：Regs[rs]＋immediate

  从存储器取来的数据放入寄存器rt

- **store指令：** sw r1, 100(r2)

  访存有效地址：Regs[rs]＋immediate

  要存入存储器的数据放在寄存器rt中

- **立即数指令：** addi r1, r2, 4

  Regs[rt] ← Regs[rs] op immediate

- **分支指令：** beq r1, r2, Loop

  转移目标地址：Regs[rs]＋immediate，rt无用

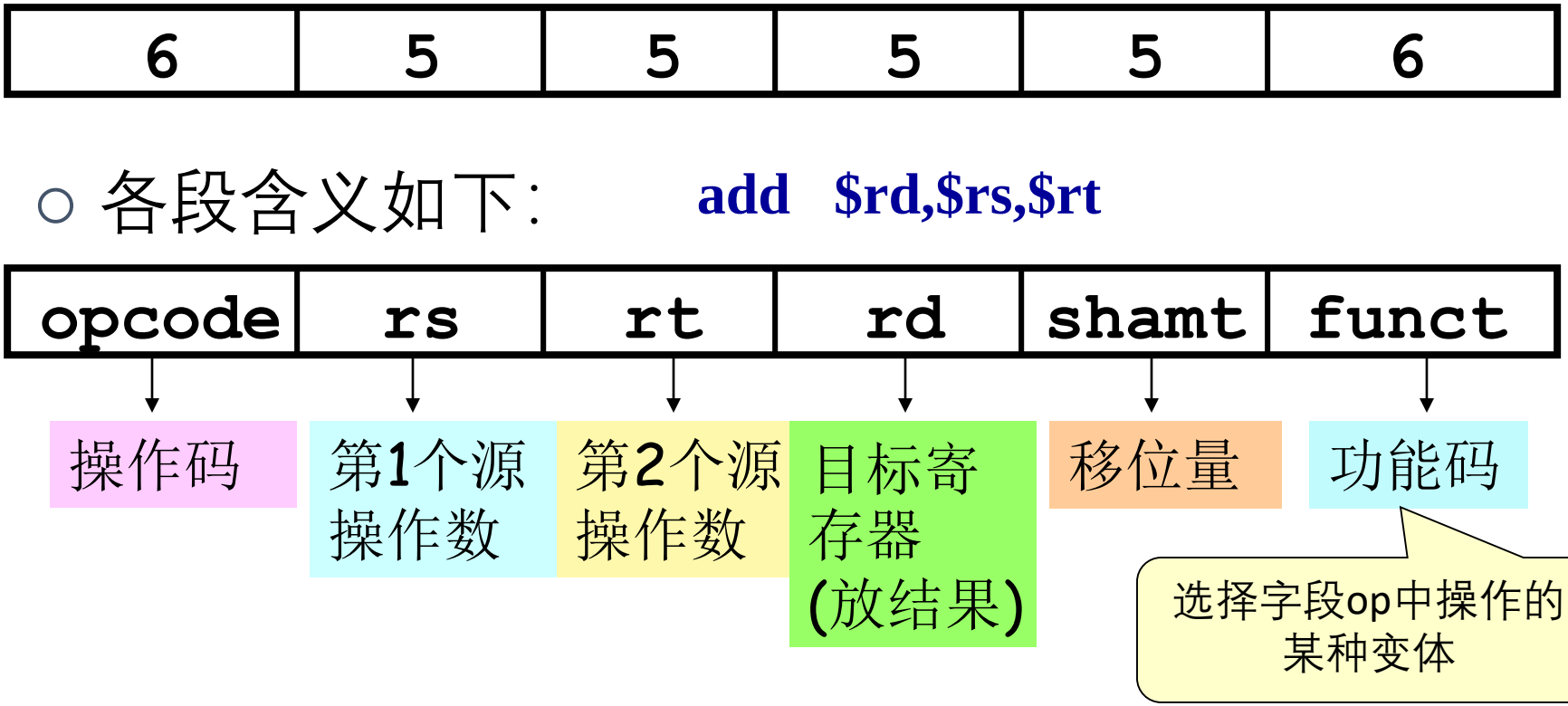
- **寄存器跳转、寄存器跳转并链接：** jr ra / jal

  转移目标地址为Regs[rs]

MIPS指令格式

(1) R-型 指令

- 一条32位的MIPS R型指令按下表bit数划分为6个字段： 6 + 5 + 5 + 5 + 5 + 6 = 32bit

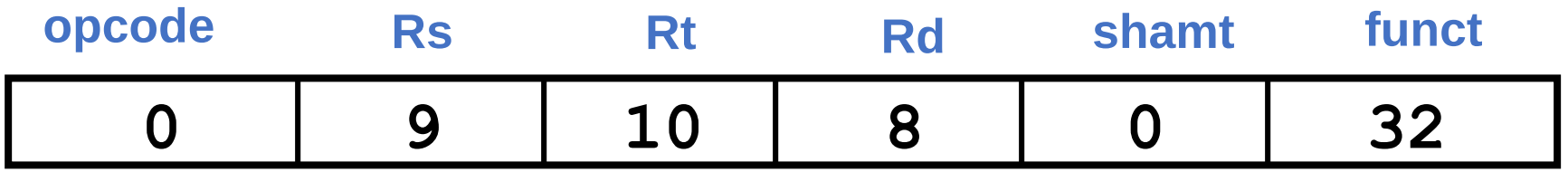


(1) R-Format 例子

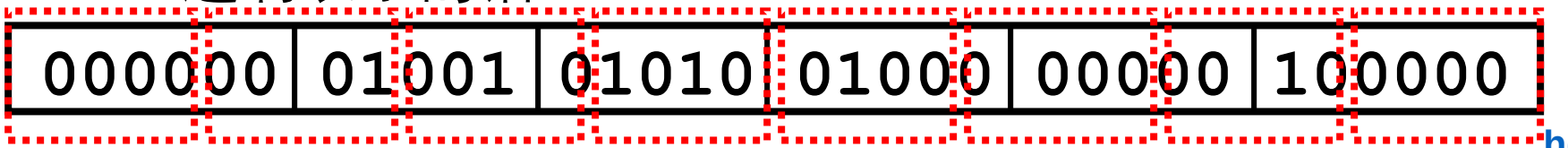
- MIPS R型指令: `add $rd,$rs,$rt`

  `add $8,$9,$10`

10进制表示:



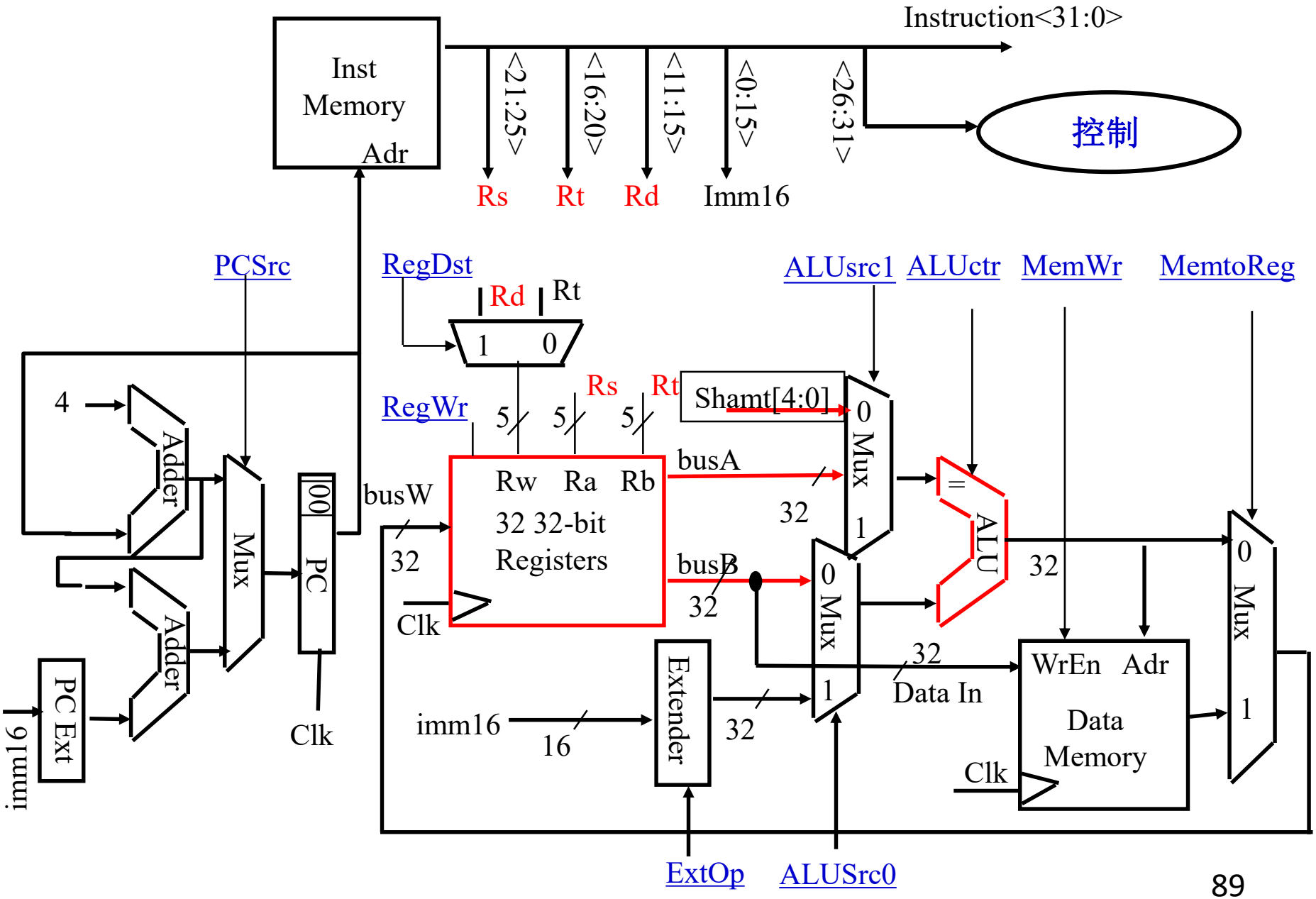
二进制表示的话:



16进制: $012A\ 4020_{hex}$

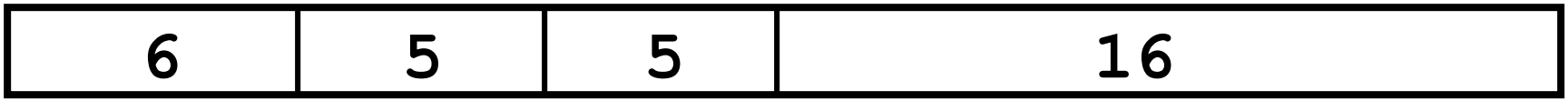
10进制: $19,546,144_{ten}$

数据通路

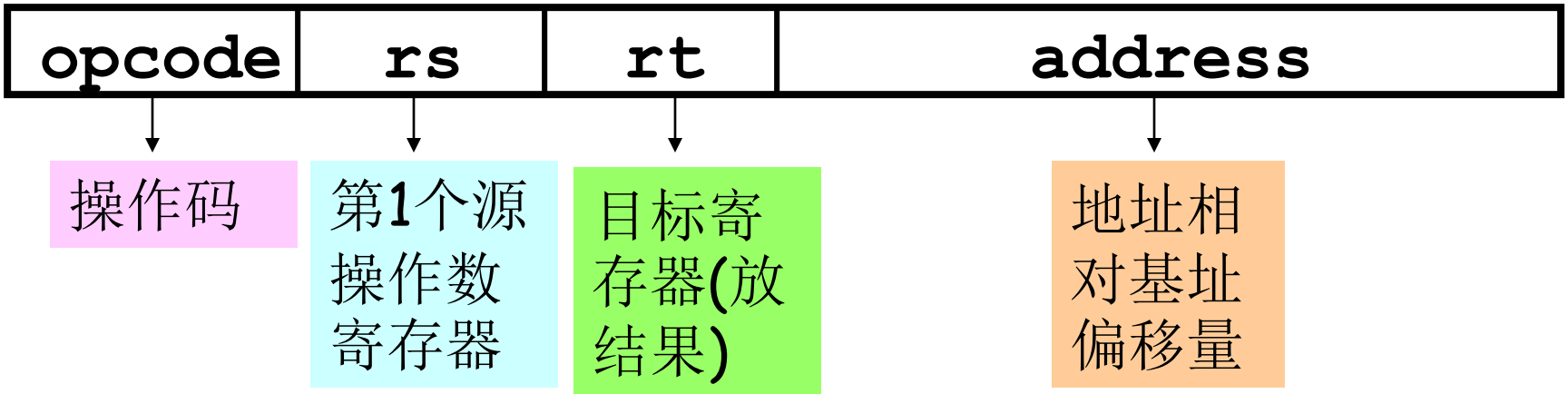

Instruction<31:0>
Inst Memory
Adr
<21:25>
<16:20>
<11:15>
<0:15>
<26:31>
控制
Rs
Rt
Rd
Imm16
PCSrc
RegDst
ALUsrc1
ALUctr
MemWr
MemtoReg
Rd
Rt
1
0
4
Adder
Adder
Mux
00
PC
Clk
RegWr
5
5
5
Rs
Rt
Shamt[4:0]
busW
Rw
Ra
Rb
32 32-bit
Registers
busA
busB
32
32
32
Clk
0
Mux
1
=
ALU
0
Mux
1
0
Mux
1
imm16
PC Ext
imm16
16
Extender
32
32
Data In
WrEn
Adr
Data
Memory
Clk
ExtOp
ALUSrc0

(2) I-型 指令

- 一条32位的MIPS I型指令按下表bit数划分为4个字段:
  6 + 5 + 5 + 1 6 = 32bit



- 各段含义如下:

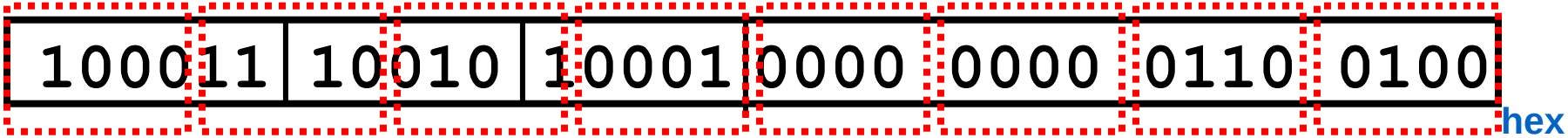


(2) I-型 指令

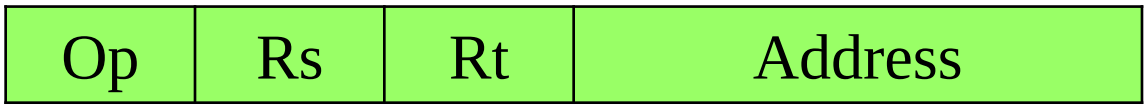
- **I-型 指令:**

例1: 装入/存储指令

lw $s1, 100($s2)　　8E510064



- 装入/存储指令、分支指令和立即数运算指令的格式
  - 数据装入: Rt = Mem[Rs + Address]
  - 数据存储: Mem[Rs + Address] = Rt



符号扩展

（2） I-Format 例子2

例2：立即数运算

addi $21,$22,-50

opcode = 8 (具体是什么操作)
rs = 22 (操作数寄存器)
rt = 21 (目的寄存器)
immediate = -50 (默认是十进制)

练习：请把指令翻译成机器码。

（2） I-Format 例子 2

例2： addi $21,$22,-50

十进制指令格式:

8	22	21	-50

二进制指令格式:

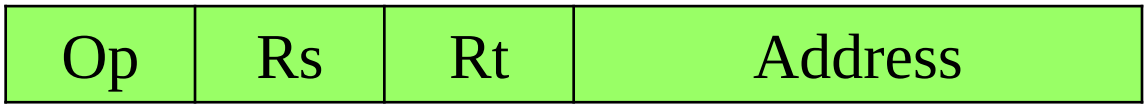
001000	10110	10101	1111111111001110

十六进制表示: $22D5\ FFCE_{hex}$

十进制表示: $584,449,998_{ten}$

（2）I-型 指令例子3

- **例3**：**分支指令**
  - if(Rs <relation> Rt) goto (PC+4)+Address

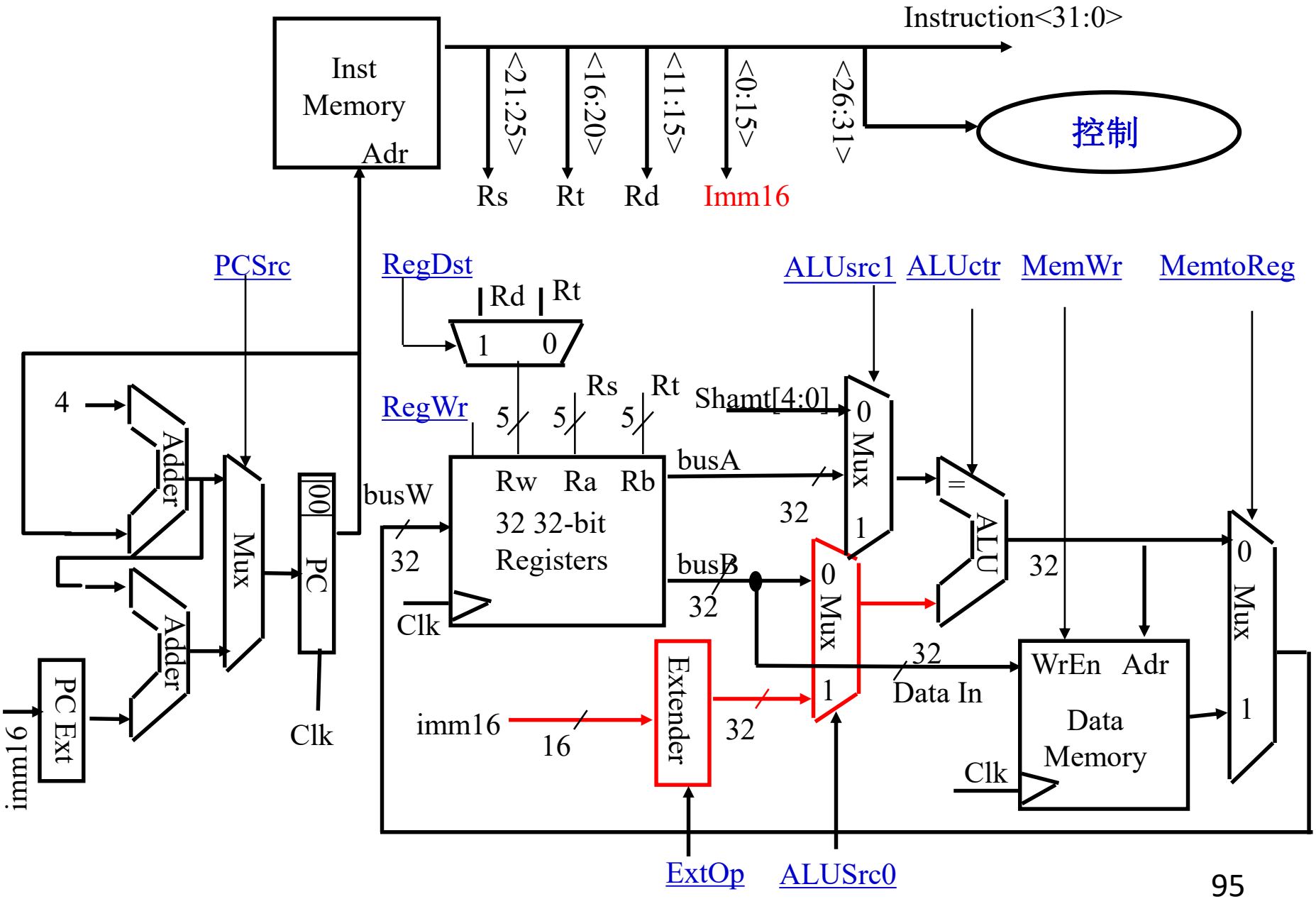


6位　5位　5位　16位

beq $t0, $t1, Target

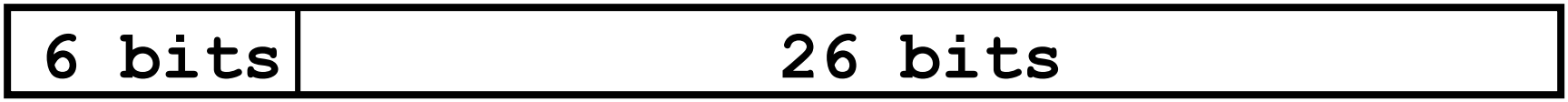
- Address: 目标程序段地址，以立即数方式给出
- 分支指令采用的寻址方式为**PC相对寻址**——分支目标的地址是PC+4与指令中的偏移量之和

数据通路


Instruction<31:0>
Inst Memory
Adr
<21:25>
<16:20>
<11:15>
<0:15>
<26:31>
Rs
Rt
Rd
Imm16
控制
PCSrc
RegDst
ALUsrc1
ALUctr
MemWr
MemtoReg
Rd
Rt
1
0
4
Adder
Adder
Mux
00
PC
Clk
imm16
PC Ext
RegWr
5
5
5
Rs
Rt
busW
32
Rw
Ra
Rb
32 32-bit
Registers
Clk
busA
32
Shamt[4:0]
0
Mux
1
busB
32
0
Mux
1
=
ALU
32
0
Mux
1
32
Data In
WrEn
Adr
Data
Memory
Clk
imm16
16
Extender
32
ExtOp
ALUSrc0

(3) J-Format 指令

- J型指令格式:



- 每部分的名字是:

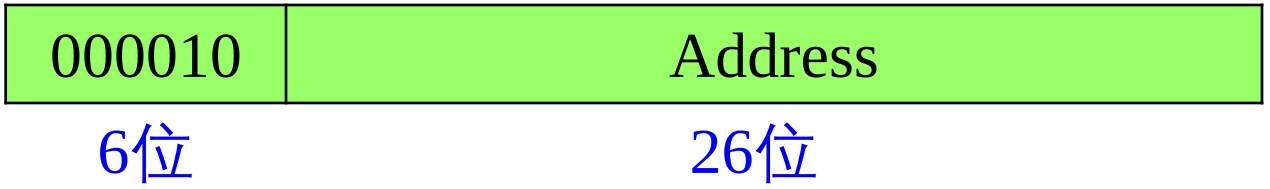
- **关键点:**

✓ 必须让 opcode 部分与R和I型指令一致,以便于电路理解执行

✓ 其他字段都节省出来给跳转的目的地址用以表示很大的跳转范围.

(3) J-Format 指令

- **例**：跳转指令的格式



- 跳转指令采用**伪直接寻址**——跳转地址为指令中的26位常数与PC中的高位拼接得到

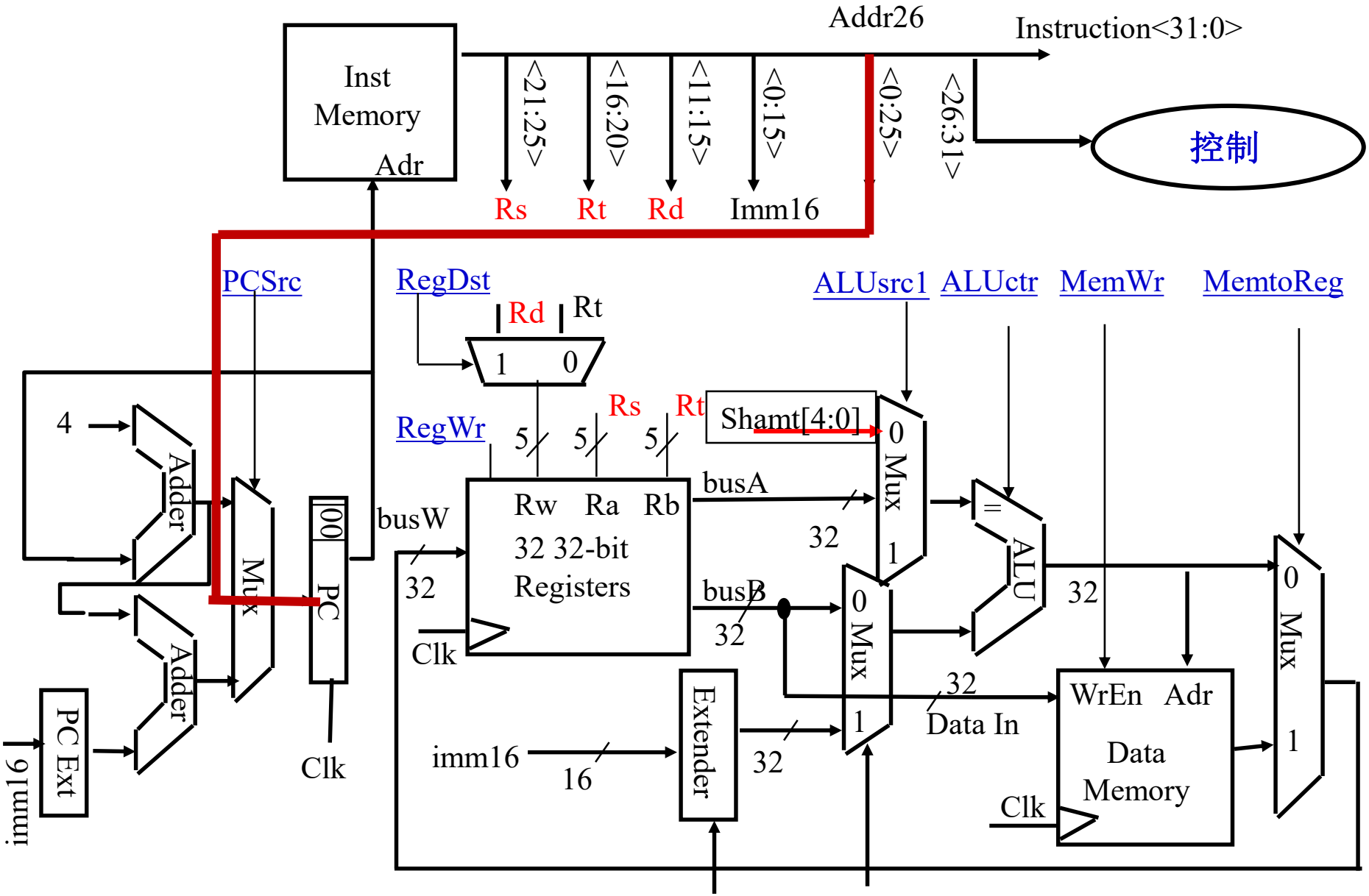
例：　j 10000　　08002710_{hex}

0000 1000 0000 0000 0010 0111 0001 0000

(3) J-Format 指令

- 总之:
  - **新的PC = { PC[31..28], 目标地址, 00 }**
- 一定要明白上面的每一项是从哪里来的!
- **Note: { , , } 表示合并**
  { 4 bits , 26 bits , 2 bits } = 32 bit address
  - { 1010, 11111111111111111111111111, 00 } = 10101111111111111111111111111100
  - Note: 书上用 ||, Verilog里用{ , , }

MIPS处理器数据通路



2.6 MIPS指令集体系结构

2.6.5 MIPS的操作

（1）MIPS指令可以分为四大类

- load和store
- ALU操作
- 分支与跳转
- 浮点操作

（2）符号的意义

- **$x \leftarrow_n y$：从y传送n位到x**
- **x，y←z：把z传送到x和y**

2.6 MIPS指令集体系结构

- **下标：表示字段中具体的位**
  - 对于指令和数据，按从最低位到最高位（即从右到左）的顺序依次进行编号，最低位为第0位，次高位为第1位，依此类推
  - 下标可以是一个数字，也可以是一个范围

  例如： $Regs[R4]_{63}$：寄存器R4的符号位

  $Regs[R4]_{0\text{-}3}$：R4的低4位

- **Mem：表示主存**
  - 按字节寻址，可以传输任意个字节

- **上标：用于表示对字段进行复制的次数**

  例如： 0^{32}：一个32位长的全0字段

2.6 MIPS指令集体系结构

- **符号##：用于两个字段的拼接，并且可以出现在数据传送的任何一边**

  举例：R8、R10：64位的寄存器，则

拼接8位

$$Regs[R8]_{0..31} \leftarrow_{32} (Mem\ [Regs[R6]]_{63})^{24}\ \#\#\ Mem\ [Regs[R6]]$$

最高位　　复制24次

✓**表示的意义是：**

以R6的内容作为地址访问内存，得到的字节按符号位扩展为32位后存入R8的低32位，R8的高32位（即$Regs[R8]_{32..63}$）不变。

（3）load和store指令

指令举例	指令名称	含 义
LD R2，20(R3)	装入双字	Regs[R2]$\leftarrow_{64}$ Mem[20+Regs[R3]]
LW R2，40(R3)	装入字	Regs[R2]$\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{32} ## Mem[40+Regs[R3]]
LB R2，30(R3)	装入字节	Regs[R2]$\leftarrow_{64}$ (Mem[30+Regs[R3]]$_0$)^{56} ## Mem[30+Regs[R3]]
LBU R2，40(R3)	装入无符号字节	Regs[R2]$\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R2，30(R3)	装入半字	Regs[R2]$\leftarrow_{64}$ (Mem[30+Regs[R3]]$_0$)^{48} ## Mem[30+Regs[R3]]## Mem[31+Regs[R3]]
L.S F2，60(R4)	装入半字	Regs[F2]$\leftarrow_{64}$ Mem[60+Regs[R4]] ## 0^{32}
L.D F2，40(R3)	装入双精度浮点数	Regs[F2]$\leftarrow_{64}$ Mem[40+Regs[R3]]
SD R4，300(R5)	保存双字	Mem[300+Regs[R5]]$\leftarrow_{64}$ Regs[R4]
SW R4，300(R5)	保存字	Mem[300+Regs[R5]]$\leftarrow_{32}$ Regs[R4]
S.S F2，40(R2)	保存单精度浮点数	Mem[40+Regs[R2]]$\leftarrow_{32}$ Regs[F2] $_{32..63}$
SH R5，502(R4)	保存半字	Mem[502+Regs[R4]]$\leftarrow_{16}$ Regs[R5] $_{0..15}$

（4）ALU指令

- 寄存器-寄存器型（RR型）指令或立即数型
- 算术和逻辑操作：加、减、与、或、异或和移位等

指令举例	指令名称	含义
DADDU R1，R2，R3	无符号加	Regs[R1]← Regs[R2]+ Regs[R3]
DADDIU R4，R5，#6	加无符号立即数	Regs[R4]← Regs[R5]+6
LUI R1，#4	把立即数装入到一个字的高16位	Regs[R1]← 0^{32} ## 4 ## 0^{16}
DSLL R1，R2，#5	逻辑左移	Regs[R1]← Regs[R2]<<5
DSLT R1，R2，R3	置小于	If(Regs[R2]< Regs[R3]) Regs[R1]← 1 else Regs[R1]←0

- **R0的值永远是0，它可以用来合成一些常用的操作**

```
DADDIU R1，R0，#100   //给寄存器R1装入常数100
DADD R1，R0，R2   //把寄存器R2中的数据传送到寄存器R1
```

2.6.6 MIPS的控制指令

- 由一组跳转和一组分支指令来实现控制流的改变
- 典型的MIPS控制指令

指令举例	指令名称	含义
J name	跳转	PC $_{0..28}$← name<<2
JAL name	跳转并链接	Regs[R31]←PC+4；PC $_{0..27}$←name<<2； ((PC+4)－2^{27})≤name＜((PC+4)+2^{27})
JALR R3	寄存器跳转并链接	Regs[R31]←PC+4；PC← Regs[R3]
JR R5	寄存器跳转	PC← Regs[R5]
BEQZ R4，name	等于零时分支	if(Regs[R4]== 0) PC←name ； ((PC+4)－2^{17})≤name＜((PC+4)+2^{17})
BNE R3，R4，name	不相等时分支	if(Regs[R3]!= Regs[R4]) PC←name ((PC+4)－2^{17})≤name＜((PC+4)+2^{17})
MOVZ R1，R2，R3	等于零时移动	if(Regs[R3]==0) Regs[R1]← Regs[R2]

2.6.6 MIPS的控制指令

（1）跳转指令

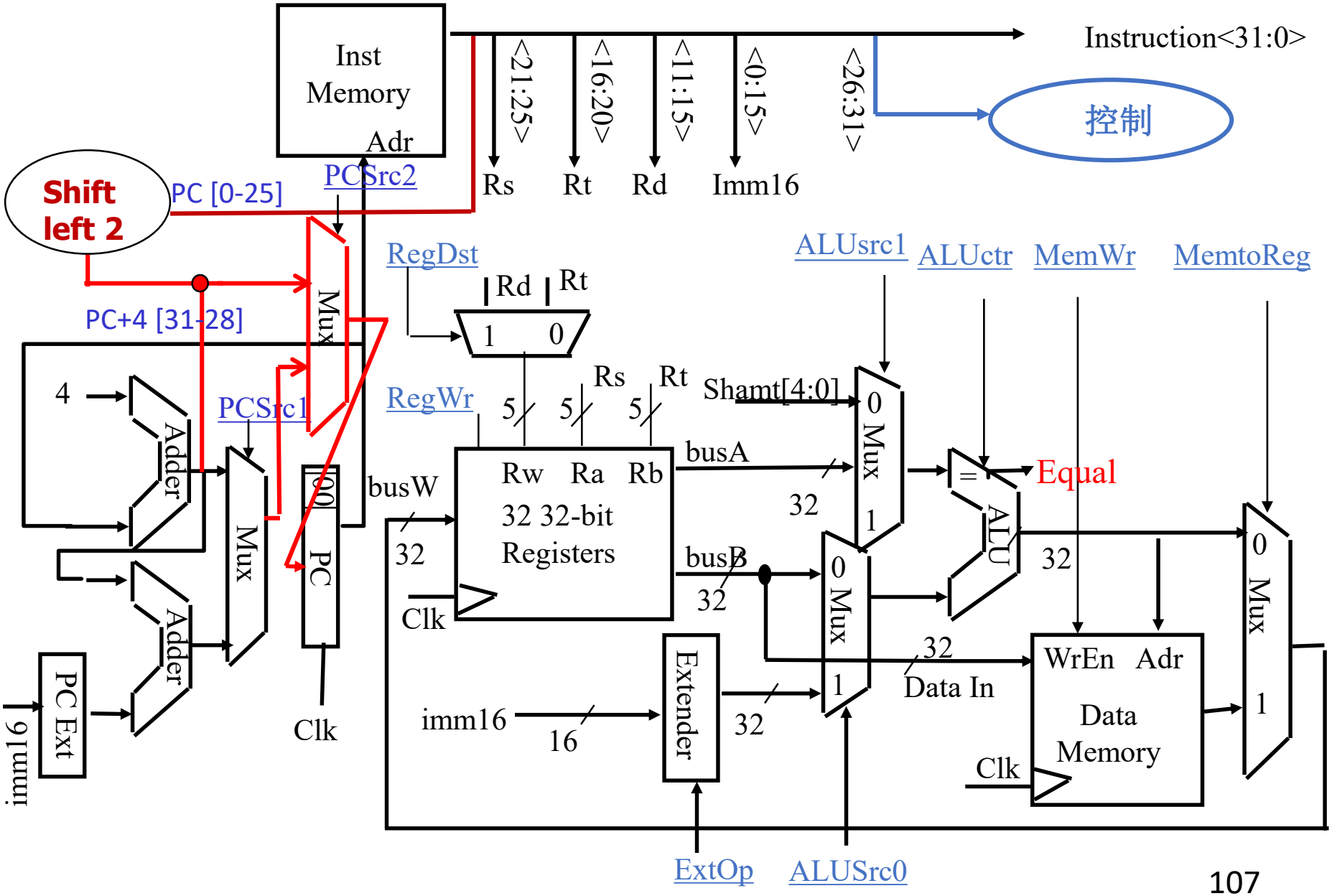
根据跳转指令确定目标地址的方式不同以及跳转时是否链接，可以把跳转指令分成4种。

- **确定目标地址的方式**
  - 把指令中的**26**位偏移量左移**2**位（因为指令字长都是4个字节）后，替换程序计数器的低**28**位。
  - **间接跳转：**由指令中指定的一个寄存器来给出转移目标地址。

000010	Address

6位 **26位**

分支指令（条件转移） • j address



2.6.6 MIPS的控制指令

（1）跳转指令

- **跳转的两种类型**
  - **简单跳转：**把目标地址送入程序计数器。
  - **跳转并链接：**把目标地址送入程序计数器，把返回地址（即顺序下一条指令的地址）放入寄存器**R31**

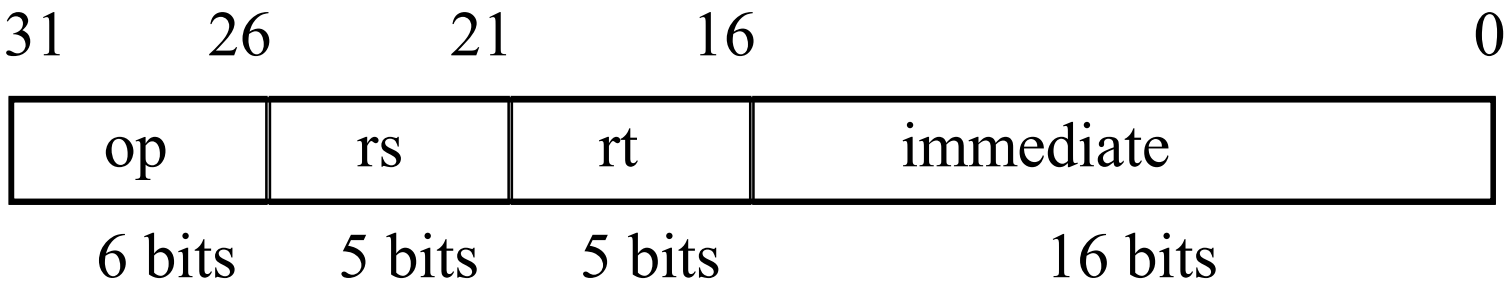
数据通路怎么改?

2.6 MIPS指令集体系结构

（2）分支指令（条件转移）　**beq　rs, rt, imm16**

- **分支条件由指令确定**：如测试某个寄存器的值是否为零
- **提供一组比较指令，用于比较两个寄存器的值：**如 slt
- **有的分支指令可以直接判断寄存器内容是否为负，或者比较两个寄存器是否相等**
- **分支的目标地址**

由16位带符号偏移量左移两位后和PC相加的结果来决定



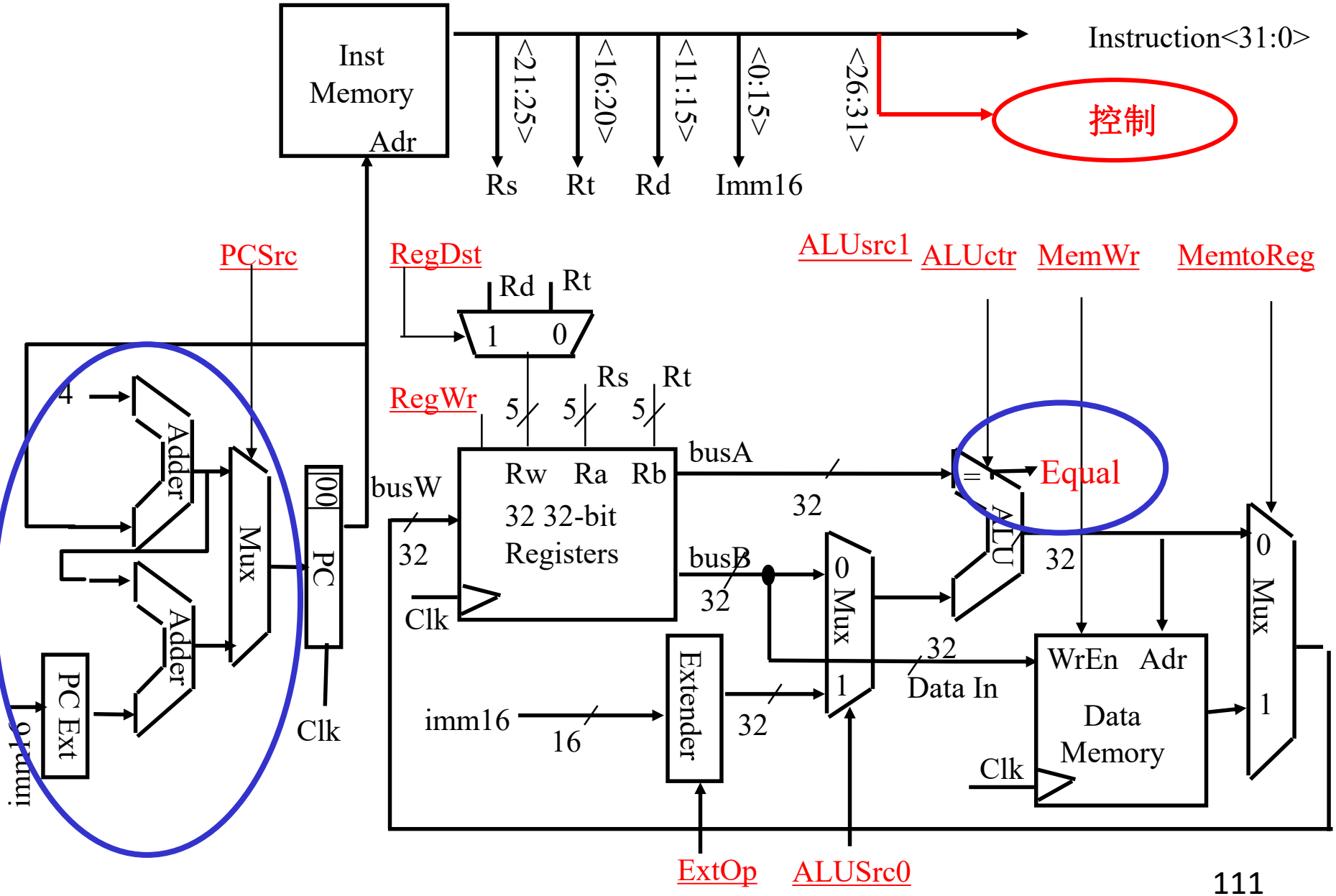
- **Datapath generates condition (equal)**

2.6 MIPS指令集体系结构

（2）分支指令（条件转移）

- 浮点条件分支指令：
  - 对特殊浮点状态寄存器（FP）中的一个位进行置位
  - 有一对分支指令可以检测到这个位
  - BC1T：浮点真分支
  - BC1F：浮点假分支
  - 测试FP状态寄存器中的对比位，并转移，相对PC+4转移16位偏移量

分支指令（条件转移） • beq rs, rt, imm16



2.6 MIPS指令集体系结构

2.6.7 MIPS的浮点操作

- 由操作码指出操作数是单精度（SP）或双精度（DP）
  - 后缀S：表示操作数是单精度浮点数
  - 后缀D：表示是双精度浮点数
- 浮点操作

  包括加、减、乘、除，分别有单精度和双精度指令。
- 浮点数比较指令
  - 根据比较结果设置浮点状态寄存器中的某一位，分支指令BC1T（若真则分支）或BC1F（若假则分支）测试该位，以决定是否进行分支

第2章 计算机指令集体系结构

2.1 指令集体系结构的分类

2.2 寻址方式

2.3 指令集体系结构的功能设计

2.4 操作数的类型和大小

2.5 指令格式的设计和优化

2.6 MIPS指令集体系结构

课本作业：2.11, 2.12, 2.13

RISC-V指令集架构：简单可靠（40条指令）、模块化

MIT 6.004 Fall 2018

特性	x86 或 ARM 架构	RISC-V
架构篇幅	数千页	少于三百页
模块化	不支持	支持模块化可配置的指令子集
可扩展性	不支持	支持可扩展定制指令
指令数目	指令数繁多，不同的架构分支彼此不兼容	一套指令集支持所有架构。基本指令子集仅 40 余条指令，以此为共有基础，加上其他常用模块子集指令总指令数也仅几十条
易实现性	硬件实现的复杂度高	硬件设计与编译器实现非常简单： ➢ 仅支持小端格式 ➢ 存储器访问指令一次只访问一个元素 ➢ 去除存储器访问指令的地址自增自减模式 ➢ 规整的指令编码格式 ➢ 简化的分支跳转指令与静态预测机制 ➢ 不使用分支延迟槽（Delay Slot） ➢ 不使用指令条件码（Conditional Code） ➢ 运算指令的结果不产生异常（Exception） ➢ 16 位的压缩指令有其对应的普通 32 位指令 ➢ 不使用零开销硬件循环