

第2章 指令系统的设计

- 2. 1 [指令系统结构的分类](#)
- 2. 2 [寻址方式](#)
- 2. 3 [指令系统的设计与优化](#)
- 2. 4 [指令系统的发展和改进](#)
- 2. 5 [操作数的类型和大小](#)
- 2. 6 [MIPS指令系统结构](#)

2.1 指令系统结构的分类

1. 区别不同指令系统结构的主要因素

CPU中用来存储操作数的存储单元的类型

2. CPU中用来存储操作数的存储单元

- 堆栈
- 累加器
- 通用寄存器组

3. 将指令系统的结构分为三种类型

- 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，又可进一步分为：

- 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）

（所有操作数都是来自通用寄存器组）

也称为load-store结构，这个名称强调：只有load指令和store指令能够访问存储器。

4. 对于不同类型的结构，操作数的位置、个数以及操作数的给出方式（显式或隐式）也会不同。
- 显式给出：用指令字中的操作数字段给出
 - 隐式给出：使用事先约定好的单元

例：表达式 $Z=X+Y$ 在4种类型指令系统结构上的代码。

假设：X、Y、Z均保存在存储器单元中，并且不能破坏X和Y的值。

堆 栈	累加器	寄存器（RM型）	寄存器（RR型）
push X	load X	load R1, X	load R1, X
push Y	add Y	add R1, Y	load R2, Y
add	store Z	store R1, Z	add R3, R1, R2
pop Z			store R3, Z

5. 通用寄存器型结构

- 现代指令系统结构的主流
- 在灵活性和提高性能方面有明显的优势
 - 跟其它的CPU内部存储单元一样，寄存器的访问速度比存储器快。
 - 对编译器而言，能更加容易、有效地分配和使用寄存器。
 - 寄存器可以用来存放变量。
 - (1) 减少对存储器的访问，加快程序的执行速度；
(因为寄存器比存储器快)

(2) 用更少的地址位（相对于存储器地址来说）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。

6. 根据ALU指令的操作数的两个特征对通用寄存器型结构进一步细分

➤ ALU指令的操作数个数

□ 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

其中一个操作数既作为源操作数，又作为目的操作数。

➤ ALU指令中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。

7. ALU指令中操作数个数和存储器操作数个数的典型组合

ALU指令中存储器操作数的个数	ALU指令中操作数的最多个数	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	VAX
3	3	MM	VAX

8. 通用寄存器型结构进一步细分为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)

9. 3种通用寄存器型结构的优缺点

表中 (m, n) 表示指令的n个操作数中有m个存储器操作数。

指令系统结构类型	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系统结构相比，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。这种类型的指令系统结构现在已不用了。

2.2 寻址方式

1. 寻址方式：指令系统中如何形成所要访问的数据的地址。
 - 寻址方式可以指明指令中的操作数是一个常数、一个寄存器操作数或者是一个存储器操作数。
 - 对于存储器操作数来说，由寻址方式确定的存储器地址称为有效地址。

2. 一些操作数寻址方式

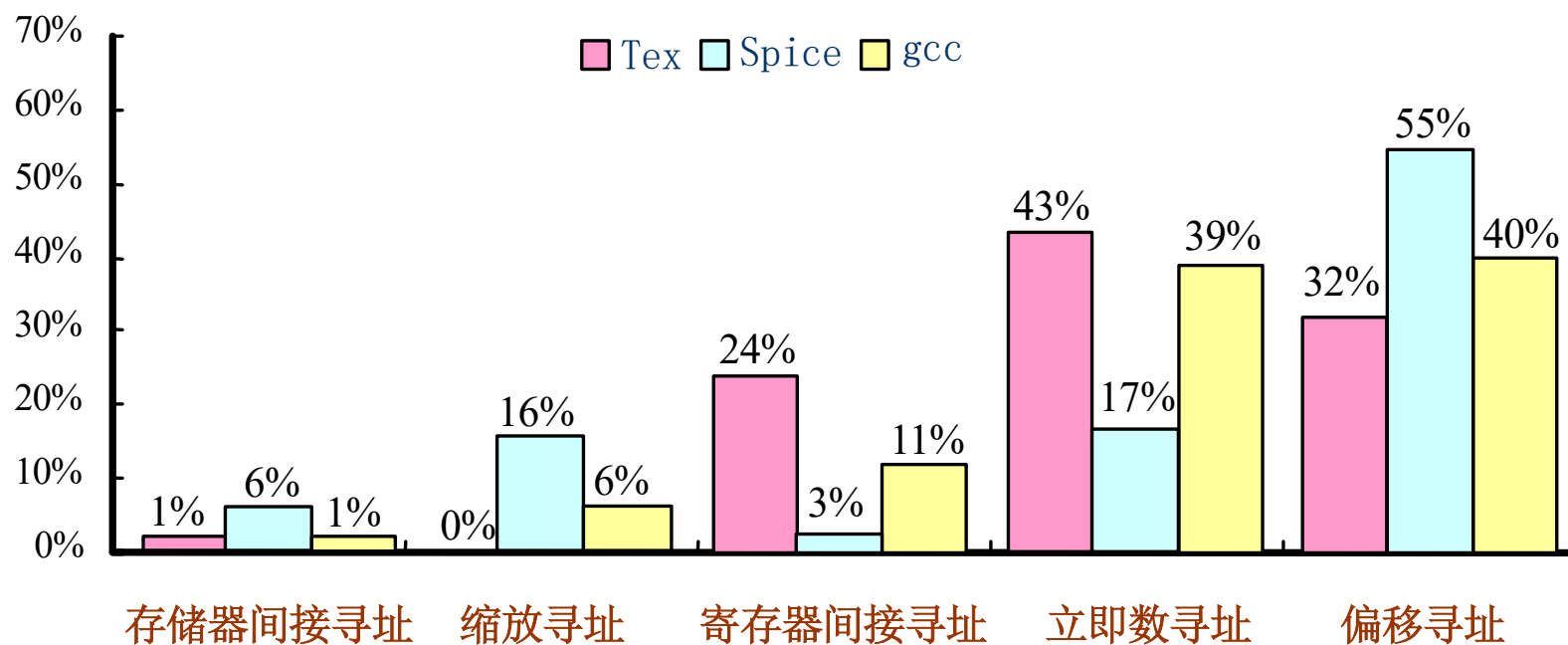
- ←: 赋值操作
- Mem: 存储器
- Regs: 寄存器组
- 方括号: 表示内容
 - Mem[]: 存储器的内容
 - Regs[]: 寄存器的内容
 - Mem[Regs[R1]]: 以寄存器R1中的内容作为地址的存储器单元中的内容

寻址方式	指令实例	含 义
寄存器寻址	ADD R1 , R2	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R1}] + \text{Regs}[\text{R2}]$
立即值寻址	ADD R3 , #6	$\text{Regs}[\text{R3}] \leftarrow \text{Regs}[\text{R3}] + 6$
偏移寻址	ADD R3 , 120(R2)	$\text{Regs}[\text{R3}] \leftarrow \text{Regs}[\text{R3}] + \text{Mem}[120 + \text{Regs}[\text{R2}]]$
寄存器间接寻址	ADD R4 , (R2)	$\text{Regs}[\text{R4}] \leftarrow \text{Regs}[\text{R4}] + \text{Mem}[\text{Regs}[\text{R2}]]$
索引寻址	ADD R4 , (R2 + R3)	$\text{Regs}[\text{R4}] \leftarrow \text{Regs}[\text{R4}] + \text{Mem}[\text{Regs}[\text{R2}] + \text{Regs}[\text{R3}]]$
直接寻址或绝对寻址	ADD R4 , (1010)	$\text{Regs}[\text{R4}] \leftarrow \text{Regs}[\text{R4}] + \text{Mem}[1010]$
存储器间接寻址	ADD R2 , @(R4)	$\text{Regs}[\text{R2}] \leftarrow \text{Regs}[\text{R2}] + \text{Mem}[\text{Mem}[\text{Regs}[\text{R4}]]]$
自增寻址	ADD R1 , (R2)+	$\begin{aligned} \text{Regs}[\text{R1}] &\leftarrow \text{Regs}[\text{R1}] + \text{Mem}[\text{Regs}[\text{R2}]] \\ \text{Regs}[\text{R2}] &\leftarrow \text{Regs}[\text{R2}] + d \end{aligned}$
自减寻址	ADD R1, -(R2)	$\begin{aligned} \text{Regs}[\text{R2}] &\leftarrow \text{Regs}[\text{R2}] - d \\ \text{Regs}[\text{R1}] &\leftarrow \text{Regs}[\text{R1}] + \text{Mem}[\text{Regs}[\text{R2}]] \end{aligned}$
缩放寻址	ADD R1 , 80(R2)[R3]	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R1}] + \text{Mem}[80 + \text{Regs}[\text{R2}] + \text{Regs}[\text{R3}] * d]$

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。

3. 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



立即数寻址方式和偏移寻址方式的使用频度最高。

4. 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。

5. 两种表示寻址方式的方法

- 将寻址方式编码于操作码中，由操作码描述相应操作的寻址方式。

适合：处理机采用load-store结构，寻址方式只有很少几种。

- 在指令字中设置专门的寻址字段，用以直接指出寻址方式。

- 灵活，操作码短，但需要设置专门的寻址方式字段，而且操作码和寻址方式字段合起来所需要的总位数可能会比隐含方法的总位数多。

适合：处理机具有多种寻址方式，且指令有多个操作数。

6. 一个需要注意的问题：物理地址空间的信息如何存放？

如何在存储器中存放不同宽度的信息？

以IBM370为例子进行讨论。

- 信息有字节、半字（双字节）、单字（4字节）和双字（8字节）等宽度。
- 主存宽度为8个字节。采用按字节编址，各类信息都是用该信息的首字节地址来寻址。

➤ 允许它们任意存储

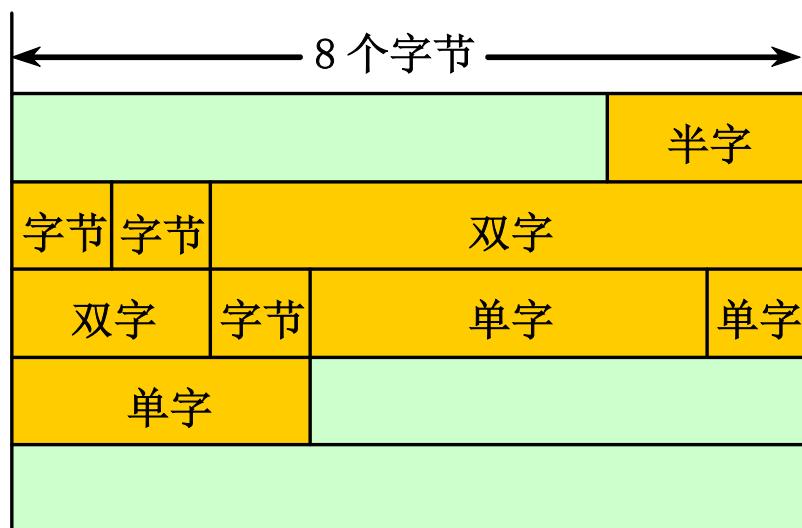
- 很可能会出现一个信息跨存储字边界而存储于两个存储单元中

➤ 信息宽度不超过主存宽度的信息必须存放在一个存储字内，不能跨边界。

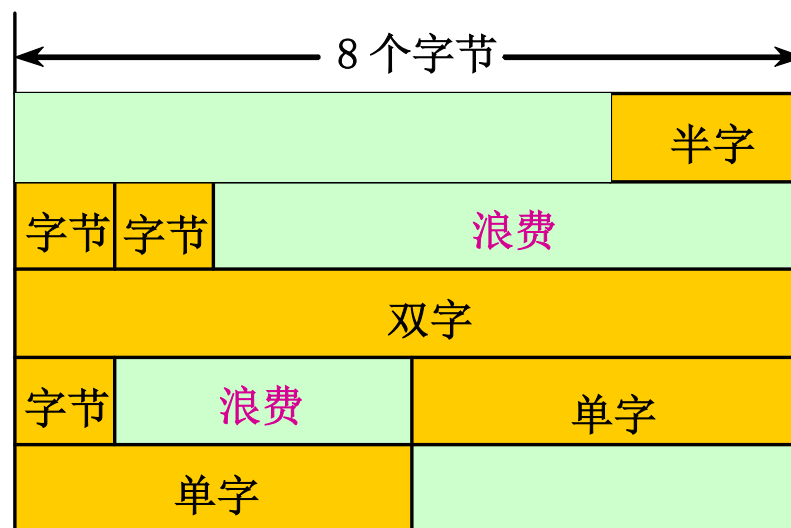
- **必须做到：**信息在主存中存放的起始地址必须是该信息宽度（字节数）的整数倍

信息存储的整数边界概念

- 满足以下条件
 - 字节信息的起始地址为：×...××××
 - 半字信息的起始地址为：×...×××0
 - 单字信息的起始地址为：×...××00
 - 双字信息的起始地址为：×...×000
- 存在存储空间的浪费，但保证访问速度。



(a)



(b)

2.3 指令系统的设计与优化

1. 指令系统的设计

- 首先考虑所应实现的基本功能，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。
- 包括
 - 指令的功能设计
 - 指令格式的设计

2. 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

3. 对指令系统的基本要求

完整性、规整性、正交性、高效率、兼容性

- **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，指令系统所提供的指令足够使用。

- 要求指令系统功能齐全、使用方便
- 下表为许多指令系统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，不同指令系统结构的支持大不相同。

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
控制	分支，跳转，过程调用和返回，自陷等
系统	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

➤ **规整性：**主要包括对称性和均匀性。

- **对称性：**所有与指令系统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 $5 \times 4 \times 2 = 40$ 种同一操作的指令。

- **正交性**：在指令中各个不同含义的字段，如操作类型、数据类型、寻址方式字段等，在编码时应互不相关、相互独立。
- **高效率**：指指令的执行速度快、使用频度高。
- **兼容性**：主要是要实现向后兼容，指令系统可以增加新指令，但不能删除指令或更改指令的功能。

4. 在设计指令系统时，有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（复杂指令系统计算机）

- 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（精简指令系统计算机）

- 尽可能地把指令系统简化，不仅指令的条数少，而且指令的功能也比较简单。

2.3.2 控制指令

1. 控制指令是用来改变控制流的。

- **跳转**：当指令是无条件改变控制流时，称之为跳转指令。
- **分支**：当控制指令是有条件改变控制流时，则称之为分支指令。

2. 能够改变控制流的指令

- 分支
- 跳转
- 过程调用
- 过程返回

3. 控制指令的使用频度

(load-store型指令系统结构的机器，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）。

4. 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。

5. 转移目标地址的表示

➤ 最常用的方法

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。

（PC相对寻址）

➤ 优点

- 有效地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。

6. 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。至少也得保存返回地址（放在专用的链接寄存器或堆栈中）。
- 过去有些指令系统结构提供了专门的保存机制来保存许多寄存器的内容。
- 现在较新的指令系统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。

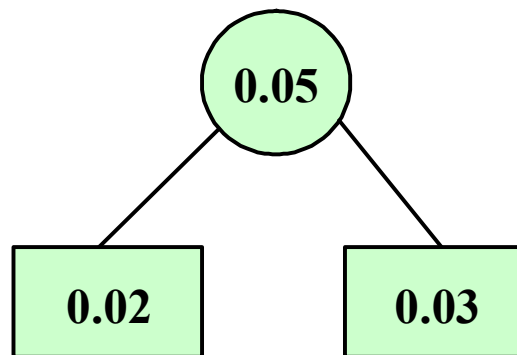
2.3.3 指令操作码的优化

- 指令由两部分组成：操作码、地址码
- 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和表示方式。
- 指令格式的优化：如何用最短的位数来表示指令的操作信息和地址信息。

1. 哈夫曼编码

- **基本思想**：当各种事件发生的概率不均等时，可以对发生概率最高的事件用最短的位数（时间）来表示（处理），而对于出现概率较低的事件，则可以用较长的位数（时间）来表示（处理），从而使总的平均位数（时间）缩短。
- **构造哈夫曼树的方法**
 - 将各事件按其使用频度从小到大依次排列；
 - 每次从中选择两个频度值最小的结点，将其合并成一个新的结点，并把新结点画在所选结点的上面，

- 然后用两条边把新结点分别与那两个结点相连。
 - 新结点的频度值是所选两个结点的频度值的和。
- 把新结点与其他剩余未结合的结点一起，再以上面的步骤进行处理，反复进行，直到全部结点都结合完毕、形成根结点为止。



画哈夫曼树的一个基本步骤

- 操作码优化的程度可以用信息熵来衡量。

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

- 表示用二进制编码表示n个码点时，理论上的最短平均编码长度。

例2.1 假设某模型机有7条指令，这些指令的使用频度如表左边所示。

- (1) 计算这7条指令的操作码编码的最短平均码长；
- (2) 画出哈夫曼树，写出这7条指令的哈夫曼编码，并计算该编码的平均码长和信息冗余量。

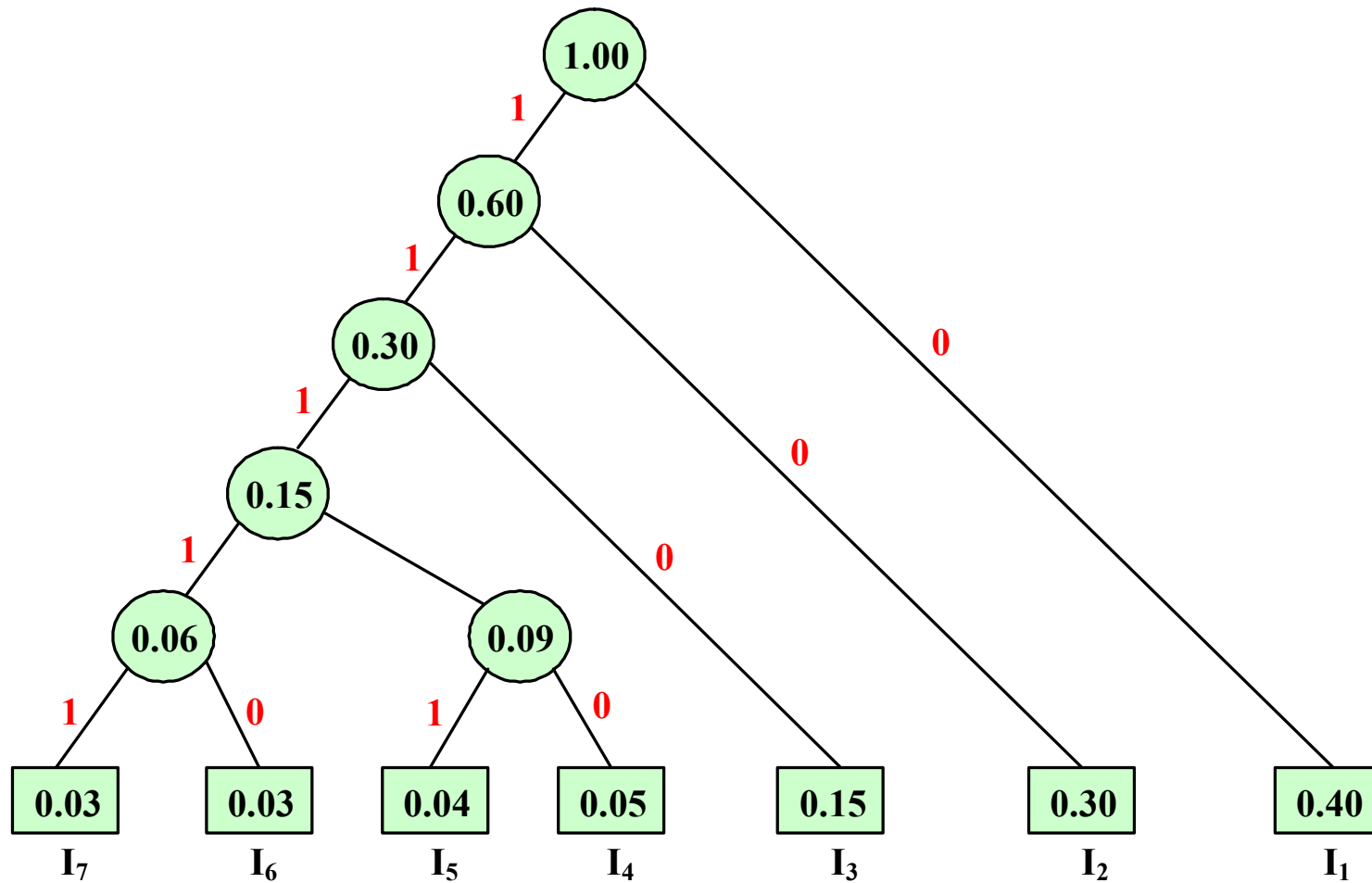
指令	频度 p_i	操作码使用 哈夫曼编码	操作码 长度 l_i	利用哈夫曼概念 的扩展操作码	操作码 长度 l_i
I_1	0.40	0	1	0 0	2
I_2	0.30	1 0	2	0 1	2
I_3	0.15	1 1 0	3	1 0	2
I_4	0.05	1 1 1 0 0	5	1 1 0 0	4
I_5	0.04	1 1 1 0 1	5	1 1 0 1	4
I_6	0.03	1 1 1 1 0	5	1 1 1 0	4
I_7	0.03	1 1 1 1 1	5	1 1 1 1	4

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

(2) 其哈夫曼树如图所示，该树的每个叶结点分别对应于一条指令。在该树中，对每个结点向下的两个分支，分别用二进制“1”和“0”来表示。

从该哈夫曼树可以很容易地写出哈夫曼编码。

具体方法：对于任意一条指令 I_i ($i=1, 2, \dots, 7$)，从哈夫曼树根结点出发、沿一条路径连接到叶结点 I_i ，把途中所经过的各分支的“0”和“1”按从左到右的顺序记录下来，便是该指令的哈夫曼编码。上表中列出了所有指令的哈夫曼编码。



哈夫曼树举例

该哈夫曼编码的平均码长是：

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

其信息冗余量为

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

- **优缺点：**可以减少操作码的平均位数，但所获得的编码是变长度的，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。
- 平均长度

$$\sum_{i=1}^7 p_i l_i = 2.3$$

比哈夫曼编码的2.2大，但很接近于2.2，而且比定长的3位编码小很多。

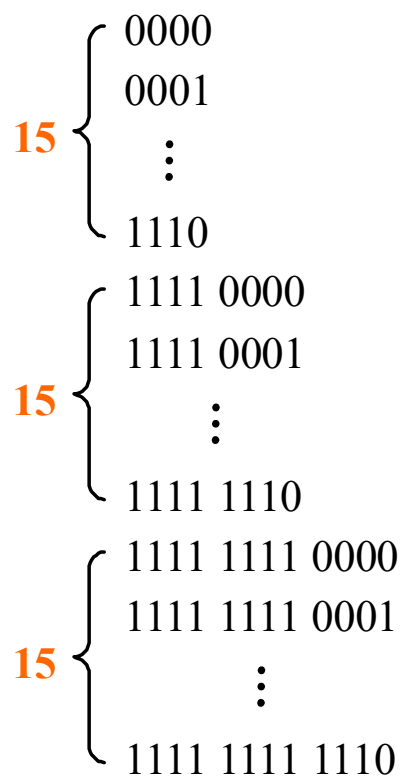
2. 等长扩展码

➤ 为了便于分级译码，一般都采用**等长扩展码**。

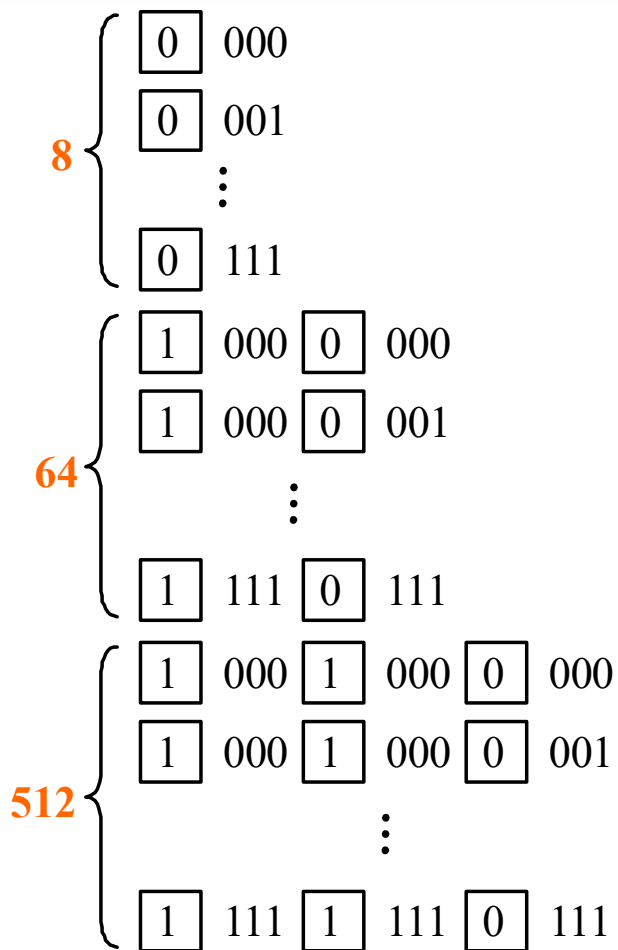
（在早期的计算机上）

例如：**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。若在前**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 的值在前**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。



15/15/15 编码法



8/64/512 编码法

3. 定长操作码

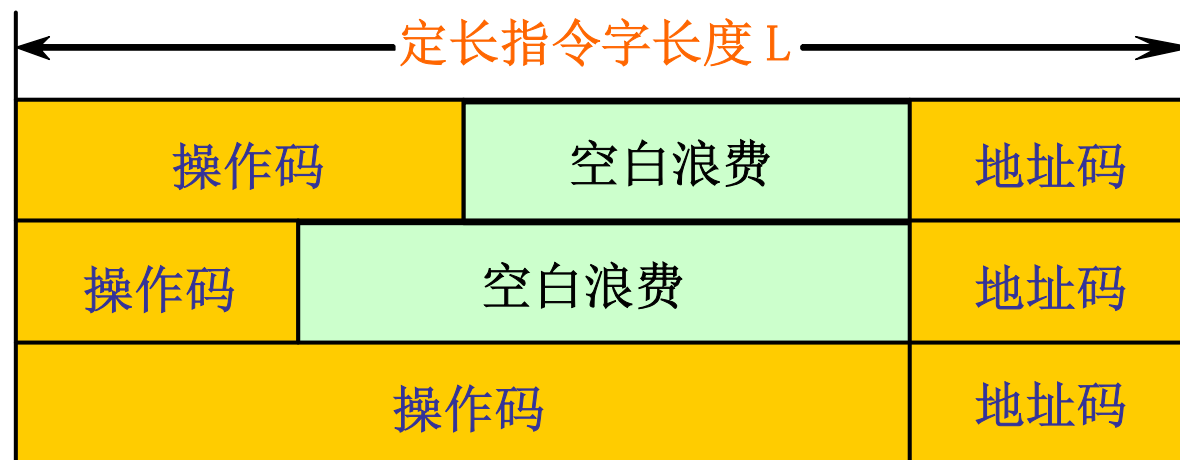
- **固定长度的操作码**：所有指令的操作码都是同一的长度（如8位）。

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。

2.3.4 指令字格式的优化

1. 如果指令字的宽度固定，地址码的长度和个数固定，则操作码的缩短并不能带来好处，只是使指令字中出现空白浪费。



2. 采用地址个数可变和/或地址码长度可变的方案

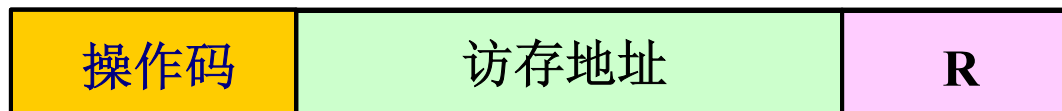
- 利用操作码缩短所带来的好处
- 最常用的操作码最短，其地址字段个数最多。

能够使指令的功能增强，从总体上减少所需的指令条数。

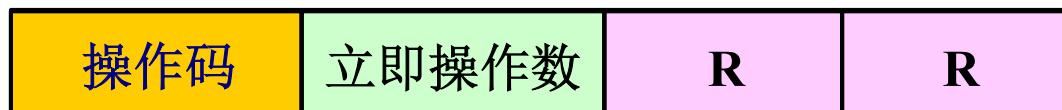
寄存器—寄存器型



寄存器—存储器型



带立即操作数



3. 考虑因素

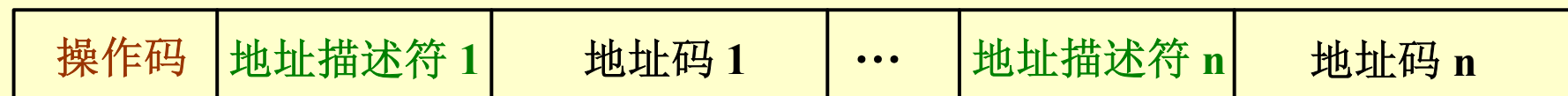
- 计算机中寄存器的个数和寻址方式的数目对机器的指令字长有很大的影响；
- 指令字的平均长度增加了，程序的平均长度也就增加了；
- 在指令系统的设计中，要在指令字长与寄存器的个数以及寻址方式的个数之间进行折中。

4. 指令系统的3种编码格式

可变长度编码格式、固定长度编码格式、混合型编码格式

➤ 可变长度编码格式

- 当指令系统的寻址方式和操作种类很多时，这种编码格式是最好的。
- 用最少的二进制位来表示目标代码。
- 可能会使各条指令的字长和执行时间相差很大。



➤ 固定长度编码格式

- 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的指令系统均采用这种编码格式。

操作码	地址码 1	地址码 2	地址码 3
-----	-------	-------	-------

➤ 混合型编码格式

- 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。

操作码	地址描述符	地址码
-----	-------	-----

操作码	地址描述符 1	地址描述符 2	地址码
-----	---------	---------	-----

操作码	地址描述符	地址码 1	地址码 2
-----	-------	-------	-------

2.4 指令系统的发展和改进

2.4.1 沿CISC方向发展和改进指令系统

1. CISC指令系统的一大特点

指令数量多、功能多样

2. 增强指令功能主要是从以下3个方面着手：

面向目标程序增强指令功能

面向高级语言的优化实现来改进指令系统

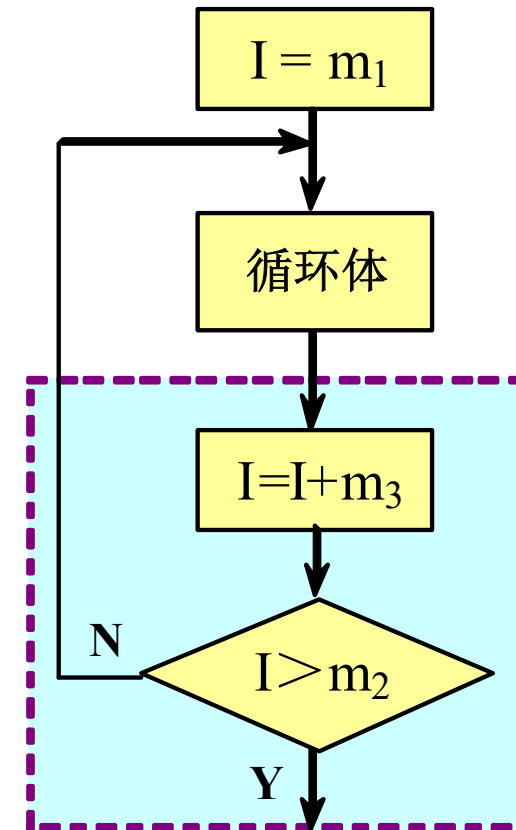
面向操作系统的优化实现改进指令系统

➤ 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，找出那些使用频度高、执行时间长的指令或指令串。对于使用频度高的指令，用硬件加快其执行；对于使用频度高的指令串，用一条新的指令来替代。
- 既能减少目标程序的执行时间，也能有效地缩短程序的长度。
- 可以从以下几个方面来改进：
 - 增强运算型指令的功能
 - 增强数据传送指令的功能
 - 增强程序控制指令的功能

例如：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。



一般循环程序的结构

- 面向高级语言的优化实现来改进指令系统
(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。

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

- 增强对高级语言和编译器的支持
 - 对高级语言中使用频度高、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，减少系统结构中的各种例外情况。

（面向高级语言的计算机）

- 高级语言计算机

- ① 间接执行高级语言机器

- 高级语言作为机器的汇编语言。

- ② 直接执行高级语言的机器

- 直接把高级语言作为机器语言。

- (一种比较激进的方法)

- 采用“**比较简单的系统结构+软件**”的做法能够在较低成本和复杂度的前提下，提供更高的性能和灵活性。

➤ 面向操作系统的优化实现改进指令系统

- 操作系统和计算机系统结构是紧密联系的，操作系统的实现在很大程度上取决于系统结构的支持。
- 指令系统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。

2.4.2 沿RISC方向发展和改进指令系统

1. CISC指令系统结构存在的问题

(1979年开始, Patterson等人的研究)

- 各种指令的使用频度相差悬殊, 许多指令很少用。
 - 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%

- 指令系统庞大，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（如占用CPU芯片总面积的一半以上），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。
- 许多指令由于操作繁杂，其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. 设计RISC机器遵循的原则

- 指令条数少、指令功能简单。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)
- 只有load和store指令才能访问存储器，其它指令的操作都是在寄存器之间进行；
(即采用load-store结构)
- 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的代码；
- 充分利用流水技术来提高性能。

3. 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8MHz；
 - 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RISCII:

- 指令条数为39，通用寄存器个数为138，时钟频率为12MHz。
- 后来发展成了Sun公司的SPARC系列微处理器。

➤ 1981年，Stanford大学Hennessy等人的MIPS
后来发展成了MIPS Rxxx系列微处理器。

➤ IBM的801

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

2.5 操作数的类型和大小

- **数据表示：**计算机硬件能够直接识别、指令系统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。
- **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)

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。
- 给数据加上标识，由数据本身给出操作数类型。
 - **优点：**简化指令系统，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. 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）、和双字（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**
 - 压缩十进制或二进制编码十进制（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。

3. 访问不同操作数大小的频率

(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。

2.6 MIPS指令系统结构

介绍MIPS64的一个子集，简称为MIPS。

2.6.1 MIPS的寄存器

1. 32个64位通用寄存器（GPRs）

- R0, R1, ..., R31
- 也称为整数寄存器
- R0的值永远是0

2. 32个64位浮点数寄存器（FPRs）

- F0, F1, ..., F31

- 用来存放32个单精度浮点数（32位），也可以用来存放32个双精度浮点数（64位）。
- 存储单精度浮点数（32位）时，只用到FPR的一半，其另一半没用。

3. 一些特殊寄存器

- 它们可以与通用寄存器交换数据。
- 例如浮点状态寄存器：用来保存有关浮点操作结果的信息。

2.6.2 MIPS的数据表示

1. 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）

2. 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。

2.6.3 MIPS的数据寻址方式

1. 立即数寻址与偏移量寻址；

立即数字段和偏移量字段都是16位的。

2. 寄存器间接寻址是通过把0作为偏移量来实现的；

3. 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；

4. MIPS的存储器是按字节寻址的，地址为64位；

5. 所有存储器访问都必须是边界对齐的。

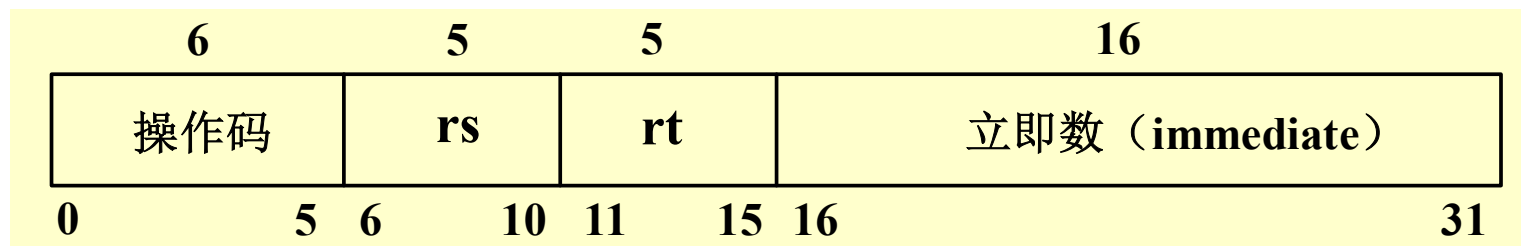
2.6.4 MIPS的指令格式

1. 寻址方式编码到操作码中
2. 所有的指令都是32位的
3. 操作码占6位
4. 3种指令格式

3种格式中，同名字段的位置固定不变。

➤ I类指令

- 包括所有的load和store指令，立即数指令，分支指令，寄存器跳转指令，寄存器链接跳转指令。
- 立即数字段为16位，用于提供立即数或偏移量。



□ load指令

访存有效地址: $\text{Regs}[\text{rs}] + \text{immediate}$

从存储器取来的数据放入寄存器rt

□ store指令

访存有效地址: $\text{Regs}[\text{rs}] + \text{immediate}$

要存入存储器的数据放在寄存器rt中

□ 立即数指令

$\text{Regs}[\text{rt}] \leftarrow \text{Regs}[\text{rs}] \text{ op } \text{immediate}$

□ 分支指令

转移目标地址: $\text{Regs}[\text{rs}] + \text{immediate}$, rt无用

□ 寄存器跳转、寄存器跳转并链接

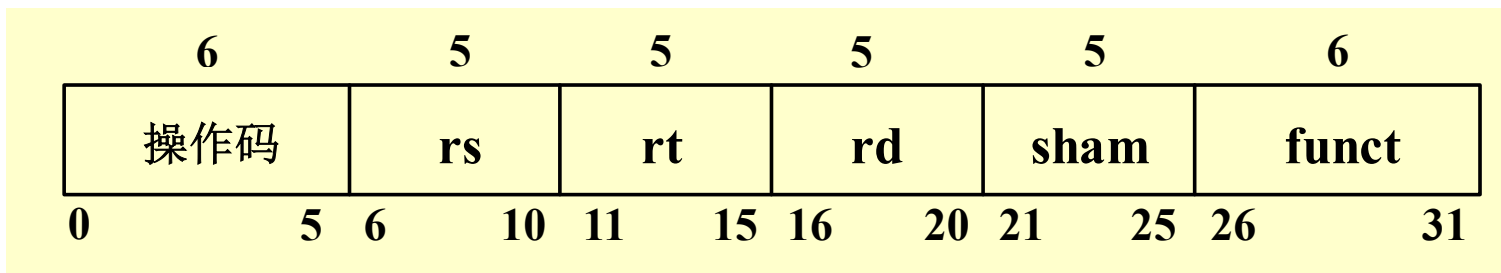
转移目标地址为 $\text{Regs}[\text{rs}]$

➤ R类指令

- 包括ALU指令，专用寄存器读/写指令，move指令等。
- ALU指令

$\text{Regs}[\text{rd}] \leftarrow \text{Regs}[\text{rs}] \text{ funct } \text{Regs}[\text{rt}]$

funct为具体的运算操作编码



➤ J类指令

- 包括跳转指令，跳转并链接指令，自陷指令，异常返回指令。
- 在这类指令中，指令字的低26位是偏移量，它与PC值相加形成跳转的地址。



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
- $x, y \leftarrow z$: 把z传送到x和y

- **下标：**表示字段中具体的位；
 - 对于指令和数据，按从最高位到最低位（即从左到右）的顺序依次进行编号，最高位为第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]₀：寄存器R4的符号位
Regs[R4]₅₆₋₆₃：R4的最低字节
- **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：0³²：一个32位长的全0字段

- **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

$$\text{Regs}[\text{R8}]_{32-63} \leftarrow_{32} (\text{Mem} [\text{Regs}[\text{R6}]]_0)_{24} \quad \text{## Mem} [\text{Regs}[\text{R6}]]$$

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 $\text{Regs}[\text{R8}]_{0-31}$ ）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	$\text{Regs}[R2] \leftarrow_{64} \text{Mem}[20+\text{Regs}[R3]]$
LW R2, 40(R3)	装入字	$\text{Regs}[R2] \leftarrow_{64} (\text{Mem}[40+\text{Regs}[R3]])_0^{32} \text{##}$ $\text{Mem}[40+\text{Regs}[R3]]$
LB R2, 30(R3)	装入字节	$\text{Regs}[R2] \leftarrow_{64} (\text{Mem}[30+\text{Regs}[R3]])_0^{56} \text{##}$ $\text{Mem}[30+\text{Regs}[R3]]$
LBU R2, 40(R3)	装入无符号字节	$\text{Regs}[R2] \leftarrow_{64} 0^{56} \text{##} \text{Mem}[40+\text{Regs}[R3]]$
LH R2, 30(R3)	装入半字	$\text{Regs}[R2] \leftarrow_{64} (\text{Mem}[30+\text{Regs}[R3]])_0^{48} \text{##}$ $\text{Mem}[30+\text{Regs}[R3]] \text{##} \text{Mem}[31+\text{Regs}[R3]]$
L.S F2, 60(R4)	装入半字	$\text{Regs}[F2] \leftarrow_{64} \text{Mem}[60+\text{Regs}[R4]] \text{##} 0^{32}$
L.D F2, 40(R3)	装入双精度浮点数	$\text{Regs}[F2] \leftarrow_{64} \text{Mem}[40+\text{Regs}[R3]]$
SD R4, 300(R5)	保存双字	$\text{Mem}[300+\text{Regs}[R5]] \leftarrow_{64} \text{Regs}[R4]$
SW R4, 300(R5)	保存字	$\text{Mem}[300+\text{Regs}[R5]] \leftarrow_{32} \text{Regs}[R4]$
S.S F2, 40(R2)	保存单精度浮点数	$\text{Mem}[40+\text{Regs}[R2]] \leftarrow_{32} \text{Regs}[F2]_{0..31}$
SH R5, 502(R4)	保存半字	$\text{Mem}[502+\text{Regs}[R4]] \leftarrow_{16} \text{Regs}[R5]_{48..63}$

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	$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + \text{Regs}[R3]$
DADDIU R4, R5, #6	加无符号立即数	$\text{Regs}[R4] \leftarrow \text{Regs}[R5] + 6$
LUI R1, #4	把立即数装入到一个字的高16位	$\text{Regs}[R1] \leftarrow 0^{32} \text{ ## } 4 \text{ ## } 0^{16}$
DSLL R1, R2, #5	逻辑左移	$\text{Regs}[R1] \leftarrow \text{Regs}[R2] \ll 5$
DSLT R1, R2, R3	置小于	$\text{If } (\text{Regs}[R2] < \text{Regs}[R3])$ $\text{Regs}[R1] \leftarrow 1 \text{ else } \text{Regs}[R1] \leftarrow 0$

R0的值永远是0，它可以用来合成一些常用的操作。

例如：

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

1. 由一组跳转和一组分支指令来实现控制流的改变
2. 典型的MIPS控制指令

指令举例	指令名称	含义
J name	跳转	$PC_{36..63} \leftarrow \text{name} \ll 2$
JAL name	跳转并链接	$\text{Regs}[R31] \leftarrow PC+4$; $PC_{36..63} \leftarrow \text{name} \ll 2$; $((PC+4) - 2^{27}) \leq \text{name} < ((PC+4) + 2^{27})$
JALR R3	寄存器跳转并链接	$\text{Regs}[R31] \leftarrow PC+4$; $PC \leftarrow \text{Regs}[R3]$
JR R5	寄存器跳转	$PC \leftarrow \text{Regs}[R5]$
BEQZ R4, name	等于零时分支	if ($\text{Regs}[R4] == 0$) $PC \leftarrow \text{name}$; $((PC+4) - 2^{17}) \leq \text{name} < ((PC+4) + 2^{17})$
BNE R3, R4, name	不相等时分支	if ($\text{Regs}[R3] \neq \text{Regs}[R4]$) $PC \leftarrow \text{name}$ $((PC+4) - 2^{17}) \leq \text{name} < ((PC+4) + 2^{17})$
MOVZ R1, R2, R3	等于零时移动	if ($\text{Regs}[R3] == 0$) $\text{Regs}[R1] \leftarrow \text{Regs}[R2]$

3. 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。
- 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。

4. 分支指令（条件转移）

- 分支条件由指令确定

例如：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。

例如：“置小于”指令

- 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

- 一条浮点条件分支指令：通过测试浮点状态寄存器来决定是否进行分支。

2.6.7 MIPS的浮点操作

1. 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

2. 浮点操作

包括加、减、乘、除，分别有单精度和双精度指令。

3. 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，以便于后面的分支指令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。