第2章 计算机指令集结构

2.1 指令集结构的分类

1. 区别不同指令集结构的主要因素

   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的主要类型
   - 堆栈
   - 累加器
   - 通用寄存器组
3. 将指令集结构分为三种类型

- 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）

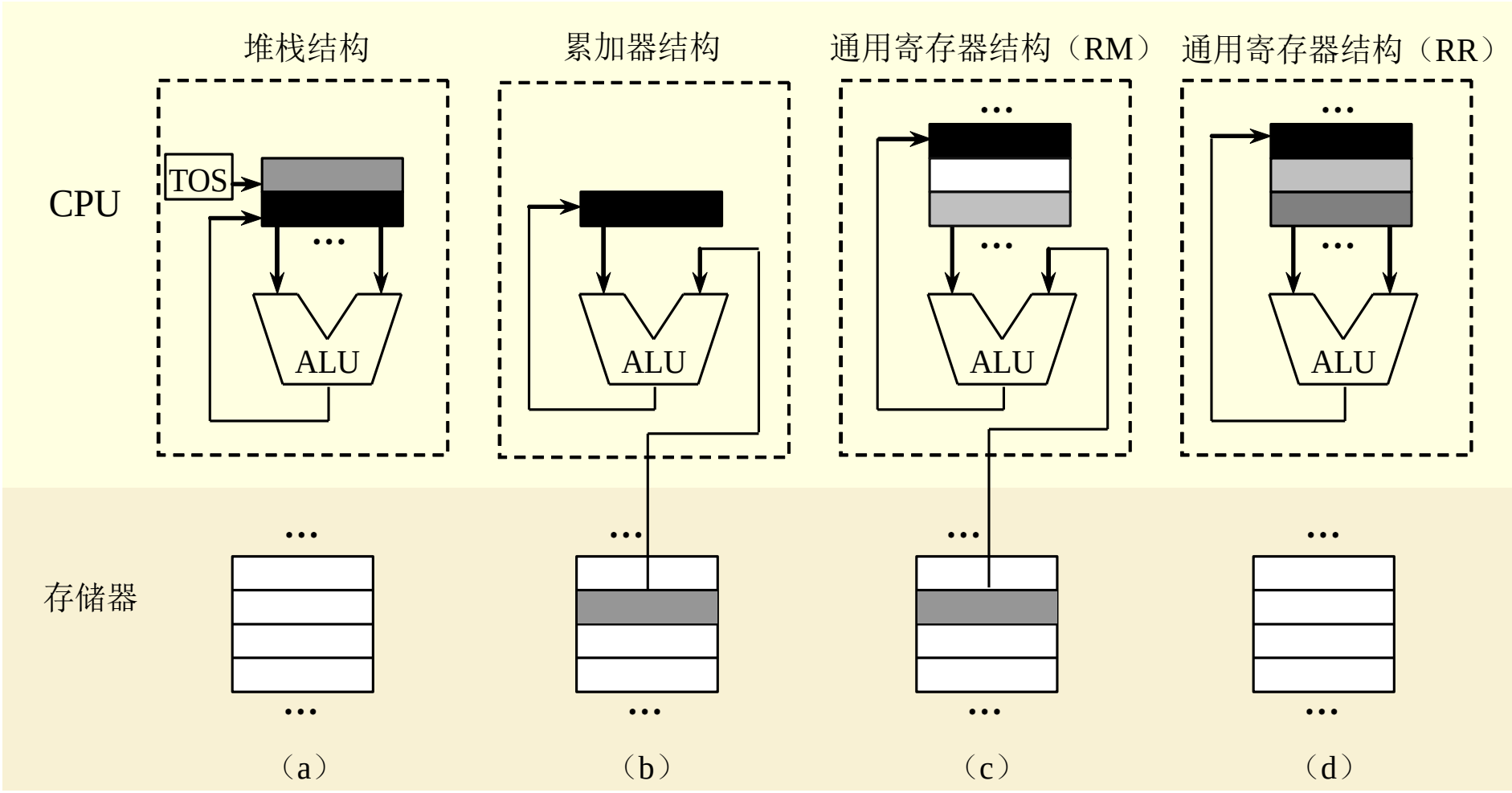
  （操作数可以来自存储器 ）

- 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. 对于不同类型的指令集结构，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：使用事先约定好的存储单元
5. 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

例: 表达式C=A+B在4种类型指令集结构上的代码。

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

6. 通用寄存器结构

- 现代指令集结构的主流
- 在灵活性和提高性能方面有明显的优势
  - 跟其他的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。

7. 根据ALU指令的操作数的两个特征对通用寄存器型指令集结构进一步细分

- 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。

8. 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

9. 通用寄存器型指令集结构进一步细分为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）

10. 3种通用寄存器型指令集结构的优缺点

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

通用寄存器型指令集结构比堆栈型结构和累加器型结构更具有优势，因其简洁和两个源操作数，被现代计算机采用。同时对于实现指令流水的处理更加方便。

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。这种类型的指令集结构现在已不用了

2.2 寻址方式

1. 一种指令集结构如何确定所要访问的数据的地址？
2. 当前的指令集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

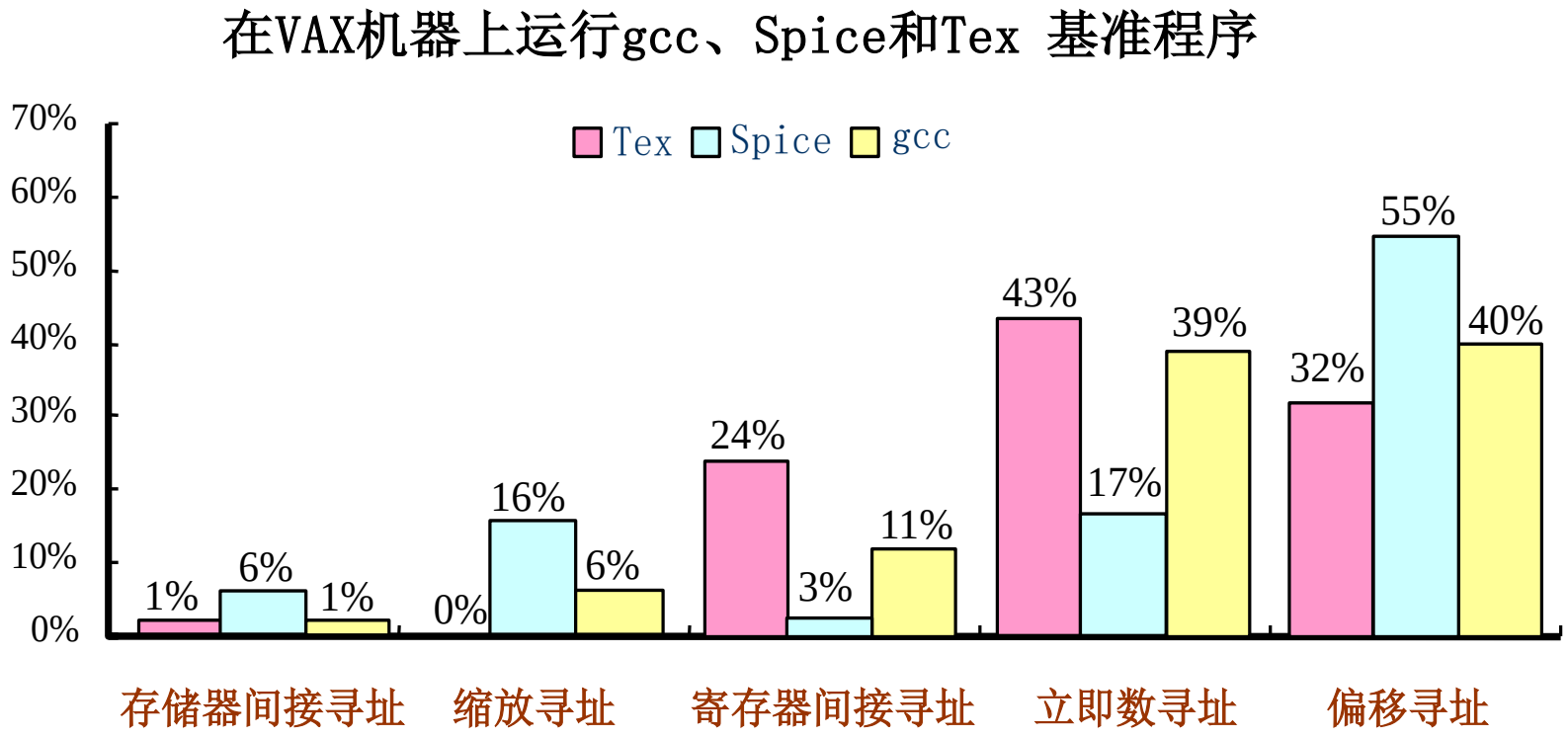
寻址方式	指令实例	含　　义
寄存器寻址	**Add R4 , R3**	**Regs[R4]←Regs[R4] + Regs[R3]**
立即值寻址	**Add R4 , #3**	**Regs[R4]←Regs[R4] + 3**
偏移寻址	**Add R4 , 100(R1)**	**Regs[R4]←Regs[R4] + Mem[100+Regs[R1]]**
寄存器间接寻址	**Add R4 , (R1)**	**Regs[R4]←Regs[R4] + Mem[Regs[R1]]**
索引寻址	**Add R3 , (R1 + R2)**	**Regs[R3]←Regs[R3] + Mem[Regs[R1]+Regs[R2]]**
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]**

*pC相对寻址是一种以程序计数器PC作为参考点的寻址方式，主要用于转移指令中指定目标指令的地址。

采用自增/自减以及缩放寻址方式中，用变量d来指明被访问数据项的大小（如4B或8B）。

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。

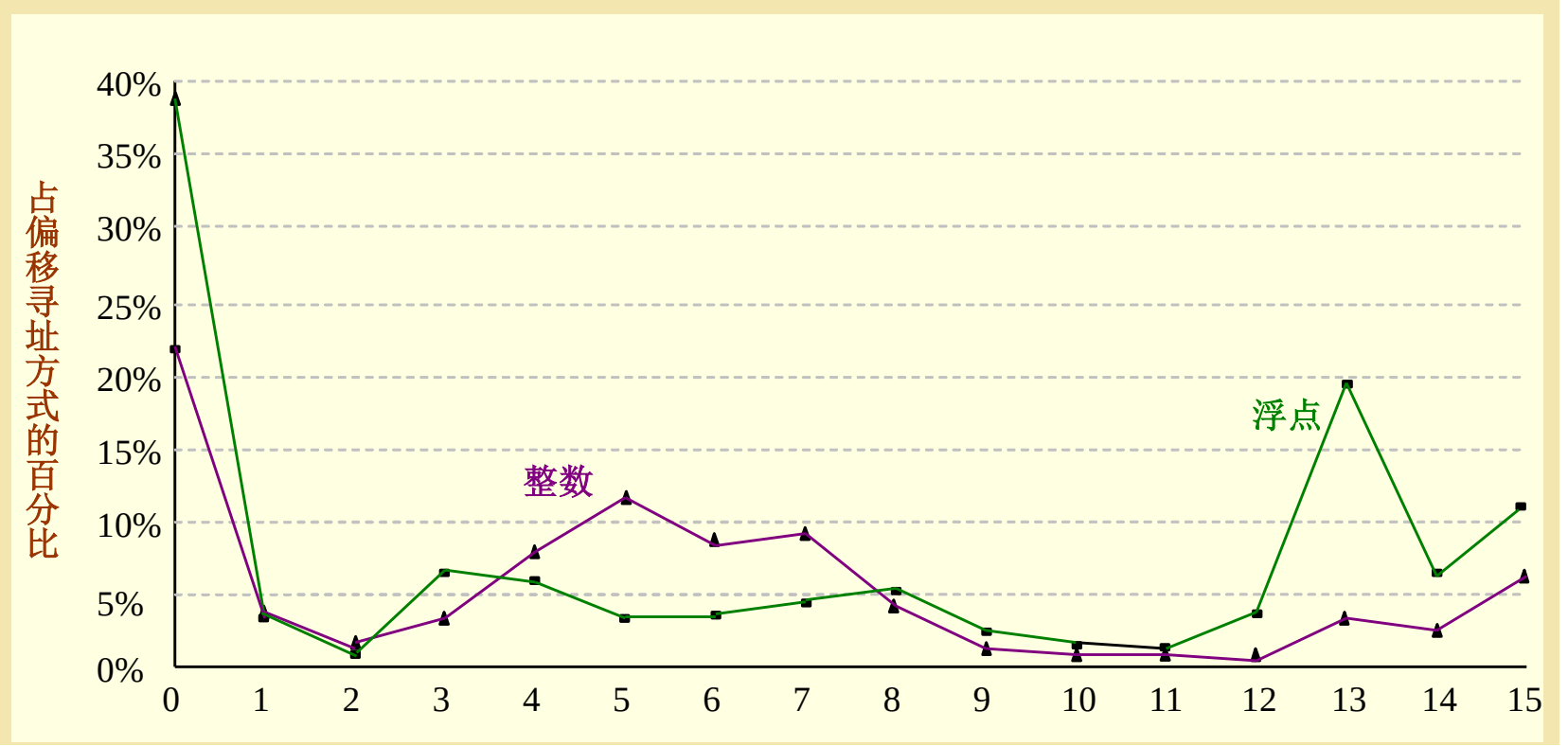
3. 各种寻址方式的使用情况统计结果



立即数寻址方式和偏移寻址方式的使用频度最高。

4. 偏移量的取值范围

在load-store结构的机器（Alpha）上运行SPEC CPU2000基准程序



从该图可以看出：

- 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

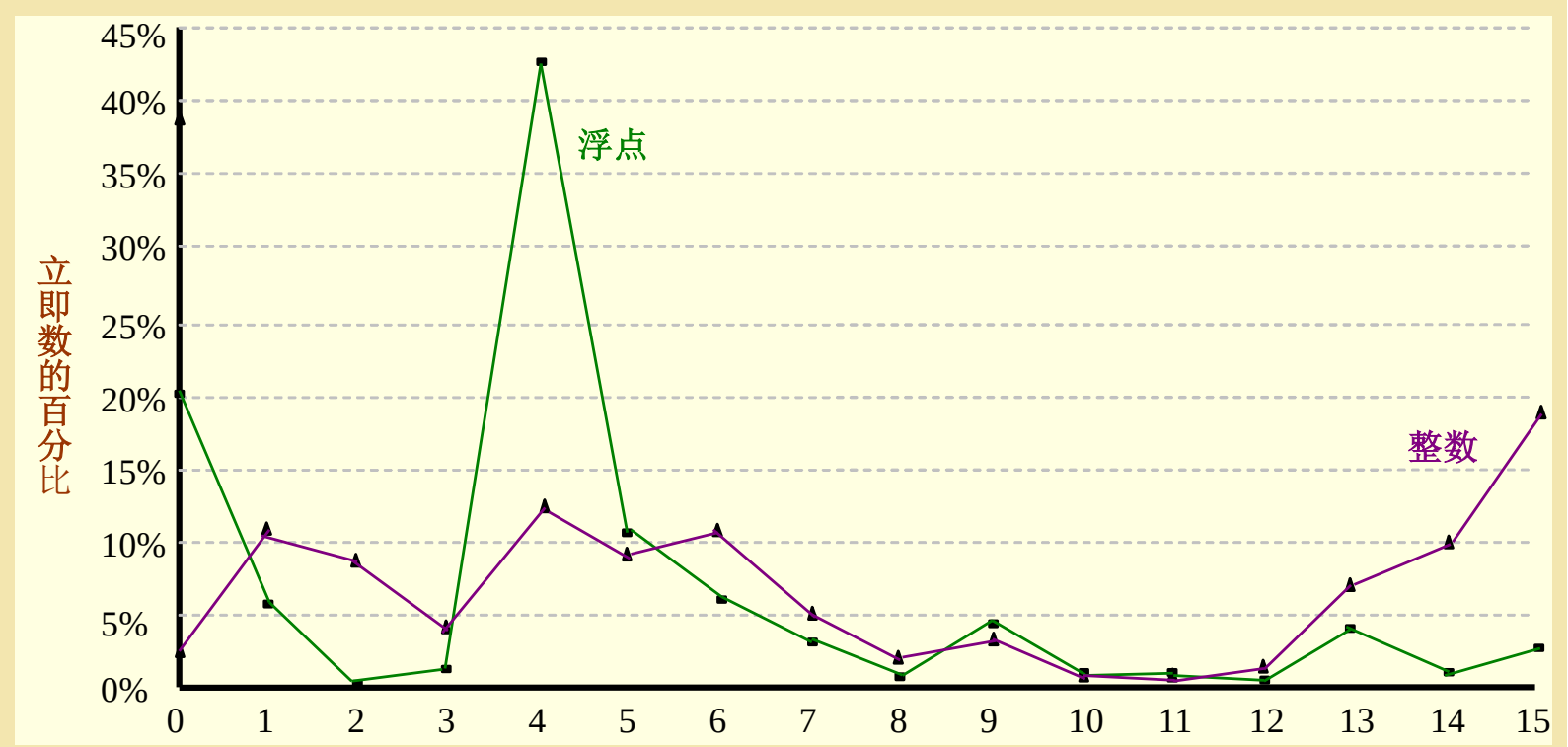
5. 立即数寻址方式

- 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。

➢ 立即数的取值范围


45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
立即数的百分比
浮点
整数
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

- 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。

2.3 指令集结构的功能设计

1. 指令集结构的功能设计
   - 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。
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. 对指令集的基本要求

完整性、规整性、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。

- 高效率：指指令的执行速度快、使用频度高。

4. 在设计指令集结构时，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）
  - 尽可能地把指令集简化，不仅指令的条数少，而且指令的功能也比较简单。

2.3.1 CISC指令集结构的功能设计

1. 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。

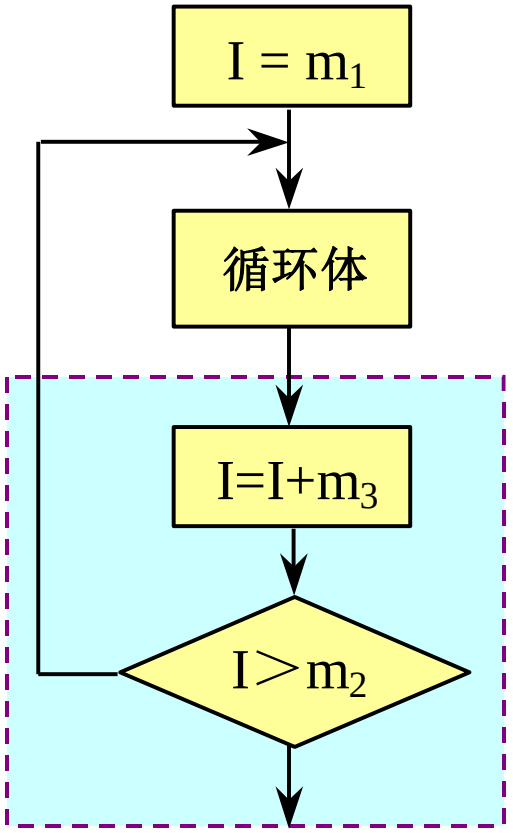
2. 增强指令功能主要是从以下几个方面着手：

- 面向目标程序增强指令功能
  - 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

    丰富的程序控制指令为编程提供了多种选择,但增加硬件成本和复杂度.

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”大于转移”指令完成上述3条指令的功能，支持循环程序快速执行,减少目标代码长度。



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集

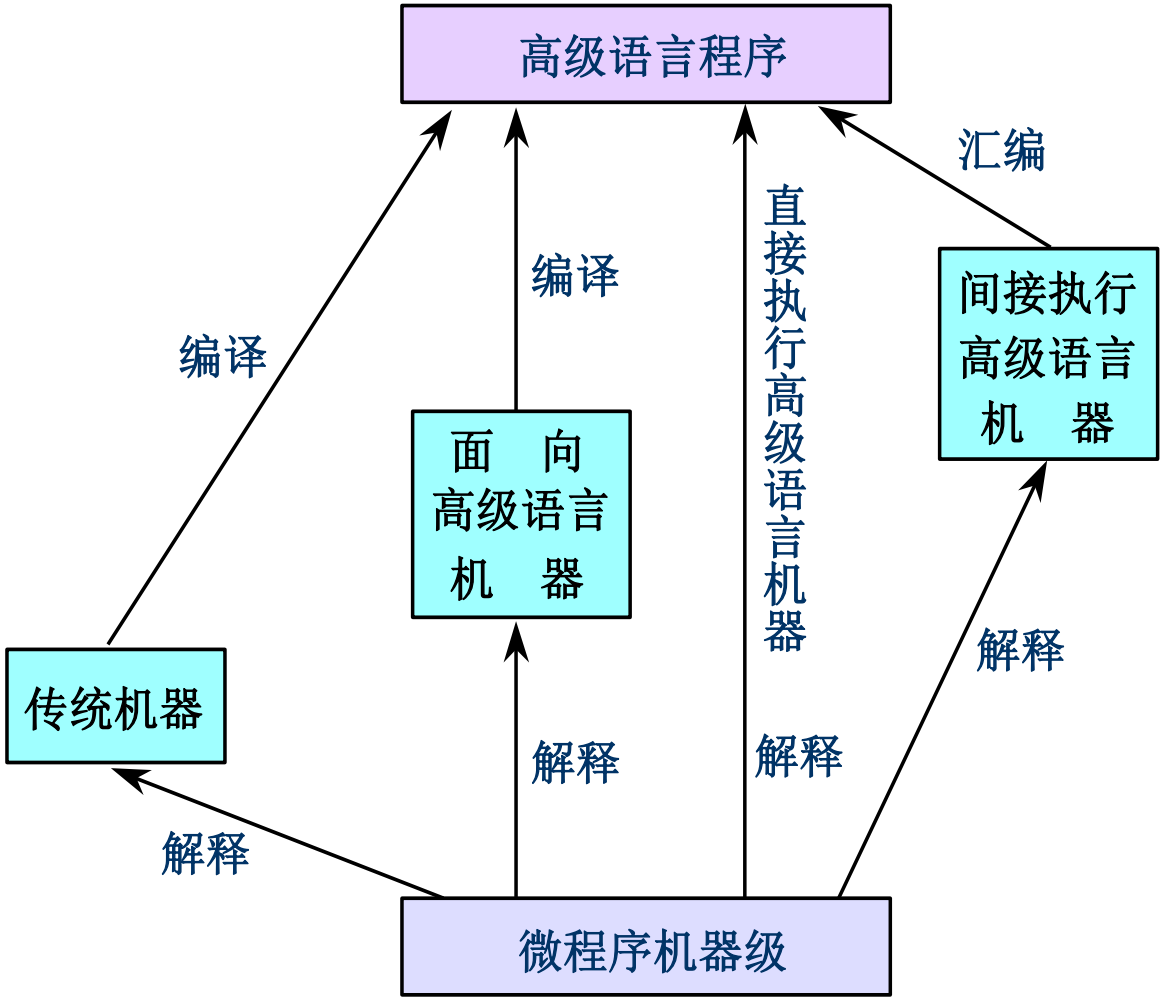
（缩小高级语言与机器语言的语义差距）是这样的

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。

改进指令集,增加对高级语言和编译器的支持,缩小高级语言与机器语言的语义差距,以提高系统整体性能.


高级语言程序
汇编
编译
直接执行高级语言机器
编译
间接执行高级语言机器
面 向 高级语言 机 器
传统机器
解释
解释
解释
解释
微程序机器级

- 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，减少系统结构中的各种例外情况,对高级语言和编译器有力支持。缩小高级语言与机器语言的语义差距.

（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

高级语言成为机器的汇编语言，这时高级语言和机器语言是一一对应的。用汇编的方法把高级语言源程序翻译成机器语言程序。

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，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。

- 面向操作系统的优化实现改进指令集
  - 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。
  - 指令集对操作系统的支持主要有：目前**必备的功能**
    - 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。

2.3.2 RISC指令集结构的功能设计

1. CISC指令集结构存在的问题

（1979年开始，Patterson等人的研究）

- 各种指令的使用频度相差悬殊
  - 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）
- 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。

  （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RISCII：

- 指令条数为39，通用寄存器个数为138，时钟频率为12 MHz。
- 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

2.3.3 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）。

4. 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

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）。
- 过去有些指令集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。

2.4 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。
- **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。
   - 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. 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。

2.5 指令格式的设计

1. 指令由两部分组成：操作码、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和表示方式。
3. 操作码的编码比较简单和直观
   - Huffman编码法

     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。

4. 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。

- 设置专门的地址描述符，由地址描述符表示相应操作数的寻址方式。

  适合：处理机具有多种寻址方式，且指令有多个操作数。

5. 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。

6. 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

➢ 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。
- 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。

操作码	地址描述符 1	地址码 1	…	地址描述符 n	地址码 n

➢ 定长编码格式

- 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

➢ 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。

2.6 MIPS指令集结构

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
   - 也被称为整数寄存器
   - 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

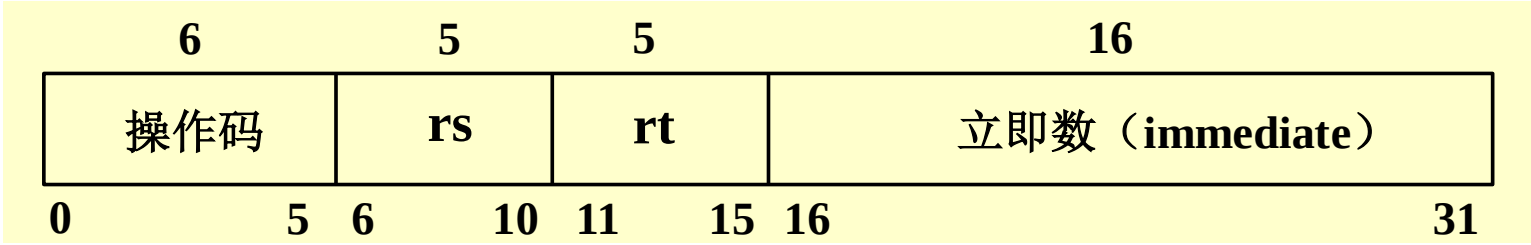
   立即数字段和偏移量字段都是16位的。

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

- 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：Regs[rs]＋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：Regs[rs]＋immediate

  要存入存储器的数据放在寄存器rt中

- 立即数指令

  Regs[rt] ← Regs[rs] op immediate

- 分支指令

  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

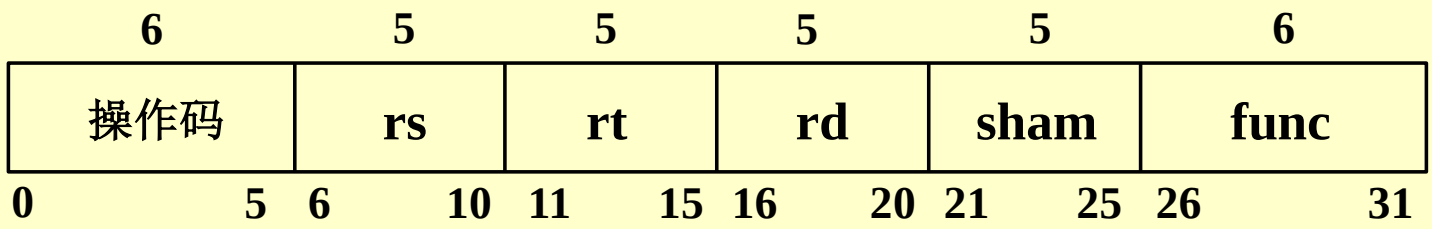
  转移目标地址为Regs[rs]

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

Regs[rd]← Regs[rs] funct Regs[rt]

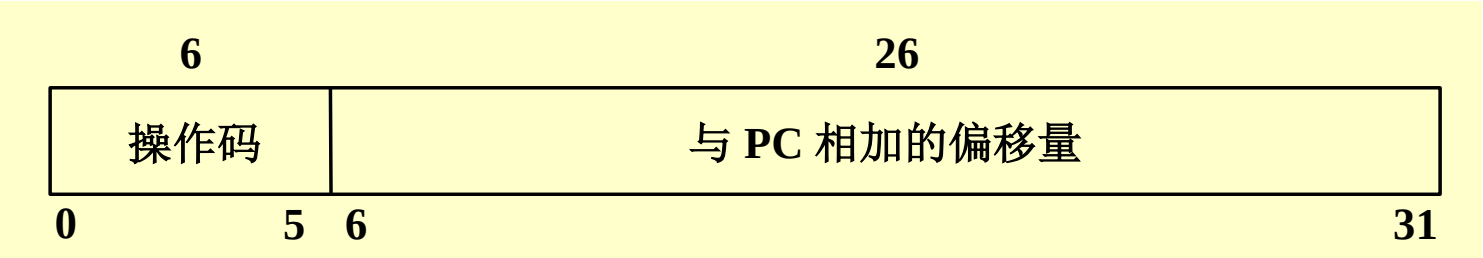
func为具体的运算操作编码



OP：指令的基本操作---操作码
Rs：第一个源操作数寄存器
Rt：第二个源操作寄存器
Rd：存放结果的目的操作寄存器
Shamt：偏移量，用于移位指令
▲Funct：函数，对操作码进行补充

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←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：0^{32}：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

$$Regs[R8]_{32..63} \leftarrow_{32} (Mem\ [Regs[R6]]_0)^{24}\ \#\#\ Mem\ [Regs[R6]]$$

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$Regs[R8]_{0..31}$）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	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)	装入单精度浮点数	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] $_{0..31}$
SH R5，502(R4)	保存半字	Mem[502+Regs[R4]] $\leftarrow_{16}$ 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	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。

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 $_{36..63}$ ← name<<2
JAL name	跳转并链接	Regs[R31] ←PC+4；PC $_{36..63}$ ←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]

跳转指令目标地址26位，左移两位后28位；分支指令偏移地址16位，左移两位后18位。

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。

```c
int mac (int a, int b, int c)
{
  return a+(b*c);
};
```

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], edi
mov     DWORD PTR [rbp-8], esi
mov     DWORD PTR [rbp-12], edx
mov     eax, DWORD PTR [rbp-8]
imul    eax, DWORD PTR [rbp-12]
mov     edx, eax
mov     eax, DWORD PTR [rbp-4]
add     eax, edx
pop     rbp
ret
```

```
push    {r7}
sub     sp, sp, #20
add     r7, sp, #0
str     r0, [r7, #12]
str     r1, [r7, #8]
str     r2, [r7, #4]
ldr     r3, [r7, #8]
ldr     r2, [r7, #4]
mul     r2, r3, r2
ldr     r3, [r7, #12]
add     r3, r3, r2
mov     r0, r3
adds    r7, r7, #20
mov     sp, r7
ldr     r7, [sp], #4
bx      lr
```