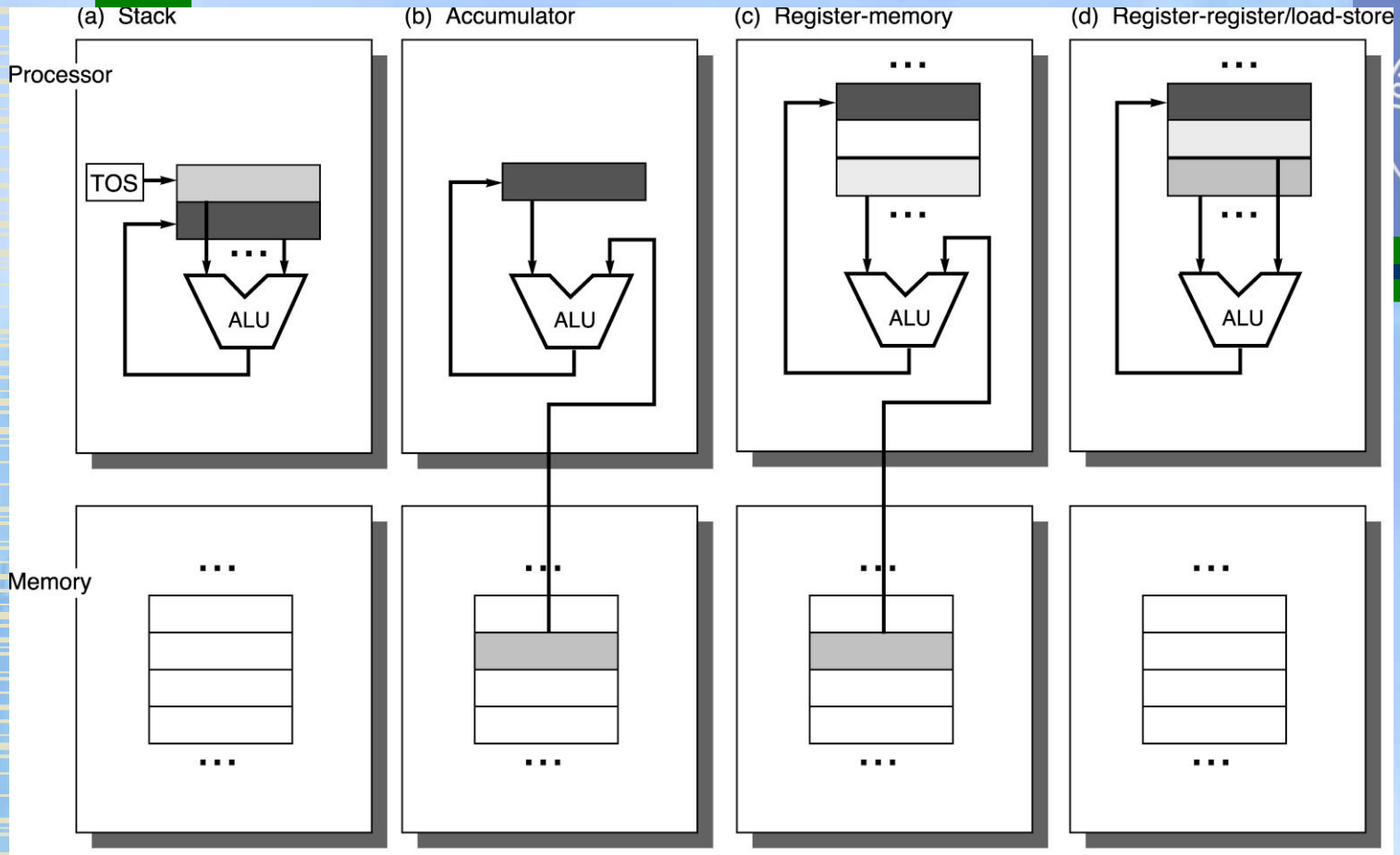
指令系统

- 指令系统的分类
- 数据表示
- 寻址技术
- 指令格式的优化设计
- 指令系统的功能设计
- 综合实例： **MIPS** 指令集

指令系统的分类

- 分类标准
  - 根据 CPU 中操作数的存储方法分类（主要分类准则）
  - 根据指令中显式操作数个数分类
  - 根据操作数能否放在存储器中分类
- 分类
  - 堆栈型指令系统
  - 累加器型指令系统
  - 寄存器型指令系统

以 C=A+B 为例说明不同指令系统的特点



Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

- ✓ 灰色块：操作数
- ✓ 黑色块：结果
- ✓ TOS：栈顶

三种指令系统的特点

- 堆栈型指令系统

优点：指令长度短，代码密度高，占用存储空间小。

缺点：代码效率低，执行效率不高。

- 累加器型指令系统

优点：指令长度短，代码密度高，代码效率高。

缺点：执行效率不高。

- 寄存器型指令系统

优点：指令简单，执行效率高，对编译程序支持好。

缺点：指令长度长。

现在通用寄存器型已成为主流结构

通用寄存器型指令系统的分类

根据分类标准 2 、3 可以分为：

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Register-register	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia TM5200
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

不同通用寄存器型指令系统的特点

Type	Advantages	Disadvantages
Register-register (0,3)	Simple, fixed-length instruction encoding. Simple code-generation model. Instructions take similar numbers of clocks to execute (see App. A).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1,2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2,2) or (3,3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

指令系统的选择

- 针对应用需求，对指令中各属性分布进行分析，根据指令执行效率确定指令集风格
- 根据各种指令的各属性分布确定寄存器数及操作数个数
- 必须考虑对 **OS** 和编译程序的支持

数据表示

- 基本概念
- 基本数据表示
- 高级数据表示
- 数据表示设计

基本概念

- 数据类型

计算机系统中可以使用和处理的各种数据的类型，主要有：整数、布尔数、字符、文件、图、表、树、阵列、队列、链表、栈、向量、串等。

- 数据表示

能由硬件直接识别和引用（即有相应运算指令和有硬件支持）的数据类型，例如：定点数据表示、逻辑数据表示、浮点数据表示等。

- 数据结构

由软件实现的带有结构的数据元素的集合，例如：串、队列、栈、向量、阵列、链表、树、图等。

三者之间的关系

- 数据表示和数据结构都是数据类型的子集；
- 数据表示是数据类型中最常用、也是相对较简单，用硬件实现相对比较容易的；
- 数据结构由软件进行实现，转换成数据表示。

确定哪些数据类型用数据表示实现，是软件与硬件的取舍问题。

结 论

数据类型是指令系统的核心内容，系统结构设计者在设计时应首先确定：

哪些数据类型全部用硬件实现，即数据表示；哪些数据类型用软件实现，即数据结构；哪些数据类型可由硬件给予适应的支持，即由软件和硬件共同来实现，并确定软件和硬件的适当比例关系。

基本数据表示

- 内容

  定点数、浮点数、十进制数、逻辑数、字符等。

- 目的

  支持数据结构，提高系统效率和性能 / 价格。

- 设计

  根据应用需求，设计各种参数、指标。

- 举例

  浮点数数据表示的设计。

浮点数数据表示设计

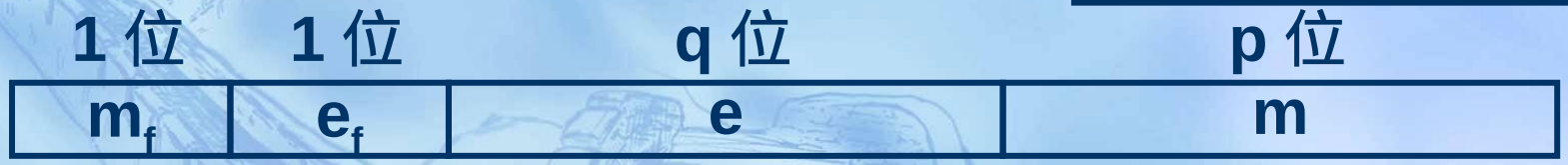
- 浮点数格式
- 浮点数尾数基值选择
- 浮点数尾数下溢处理
- 浮点数格式设计

浮点数格式

两个符号：
- m_f：尾数符号
- e_f：阶码符号

$$N = m \cdot r_m^e$$

两个数值：
- m：尾数的值
- e：阶码的值



两个基：
- r_m：尾数的基
- r_e：阶码的基

两个字长：
- p：尾数的长度
- q：阶码的长度

浮点数尾数基值的选择

- 表数范围
  随 r_m 加大，范围加大。
- 表数个数
  随 r_m 加大，个数增多。
- 表数精度
  随 r_m 加大，精度变低。
- 运算精度损失
  随 r_m 加大，损失变小。
- 运算速度
  随 r_m 加大，速度变快。

r_m 的选择应根据应用需要来综合平衡：

- 巨 / 大 / 中型机 r_m 宜取大

  尾数字长较长可以弥补精度的损失。
- 小 / 微型机 r_m 宜取小

  提高的精度可以弥补尾数字长较短的不足。

浮点数尾数下溢的处理

- 问题

在浮点数操作（相加、相乘、右移等）过程中产生的下溢会造成精度的损失。

- 解决

设计下溢处理方法，有多种方法，不同的方法有不同的优点和缺点，其出发点和应用场合也不一样，应根据需要进行选择。

- 截断法
- 舍入法
- 恒置“1”法
- 查表舍入法

浮点数格式设计

- 尾数

码制可以采用原码或补码，数制可以采用整数或小数，基可以采用二进制、四进制、八进制、十进制或十六进制。多数机器采用原码、小数表示，尾数的基 $r_m=2$ 。

- 阶码

码制可以采用移码或补码，数制采用整数，基 $r_e=2$ 。一般机器都采用移码、整数表示。

浮点数格式设计

- 设计重点

在表数范围和表数精度给定的情况下，如何确定最短的尾数字长 **p** 和阶码字长 **q** 。

- 研究对象
  - 阶码长度 **q**

    影响表数范围。
  - 尾数长度 **p**

    影响表数精度。
  - 尾数基值 r_m

    影响表数范围、精度及数在数轴上分布离散程度。

高级数据表示

- 内容

  堆栈、向量、数组（队列）、记录、自定义数据表示等。

- 目的

  支持数据结构，提高系统效率和性能 / 价格。

- 举例

  自定义数据表示。

自定义数据表示

- 存在问题
- 解决方法
  - 带标志符的数据表示法
  - 数据描述符表示法

存在问题

在高级语言与机器语言之间存在着很大的语义差距（例如：运算符和数据类型之间的关系），增加了编译程序的负担，能否在设计机器语言时，缩短与高级语言之间的差距？

带标志符的数据表示法

- 思想

  每个数据的格式为：

  标志符由编译器或其它系统软件建立，对一般高级语言程序员和计算机用户透明。

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

带标志符的数据表示法

- 优点
  - 简化指令系统和程序设计
  - 简化编译程序
  - 便于硬件实现一致性校验
  - 能由硬件自动完成数据类型的变换
  - 为软件调试和应用软件开发提供支持
  - 支持了数据库系统的实现与数据类型无关的要求
- 缺点
  - 数据和指令的长度可能不一致
  - 指令执行速度降低，程序设计时间、编译时间和调试时间缩短
  - 硬件复杂度增加

存储空间分析

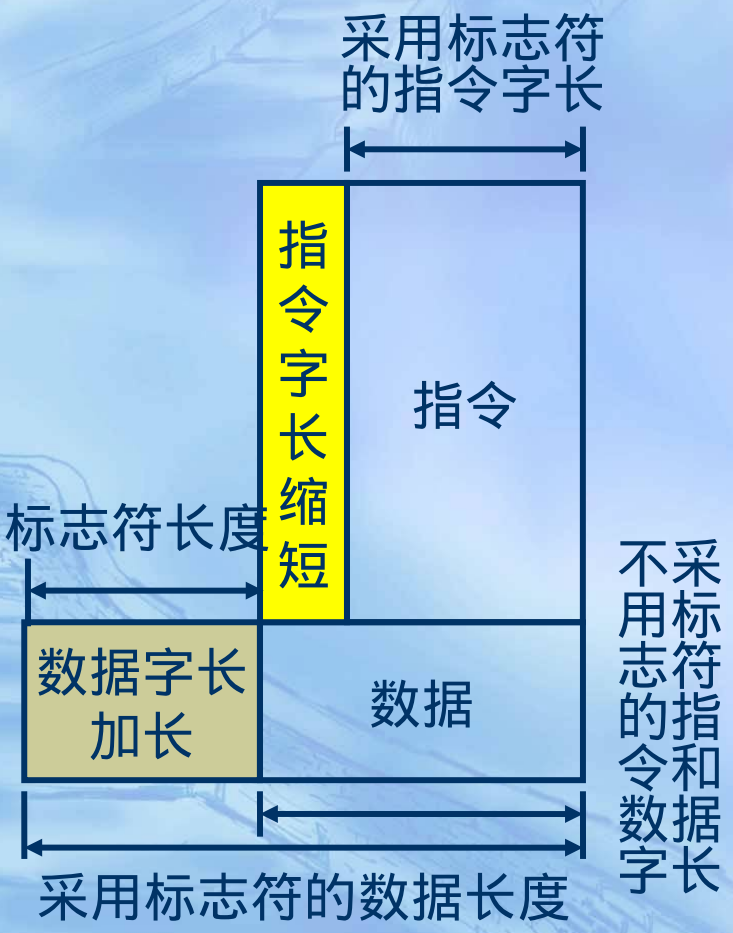
- 问题

  存储空间将会增加。

- 解决

  合理地设计和使用会不增反降。

- 原因
  - 数据字增加，指令字缩小
  - 编译程序缩短，目的程序缩短



存储空间分析

例：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。分别计算一个有 I 条指令的程序在这两种不同类型的处理机中所占用的存储空间。

答： 程序在 X 中的存储空间

$$B_X = 32I + \frac{2\times 32I}{R}$$

程序在 Y 中的存储空间：

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

二者的比值为：

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

分析：当 R > 3 时有 $B_Y/B_X < 1$。在实际应用中经

数据描述符表示法

- 思想

对于许多连续存放的同属性数据，例如：向量、矩阵、多维数组等，可以采用一个数据描述符作用于这样的一组数据，而没有必要让每个数据都带标志符。

- 例子

以 **Burroughs** 公司生产的 **B-6700** 机中采用的数据描述符表示方法进行介绍。

B-6700 中的格式

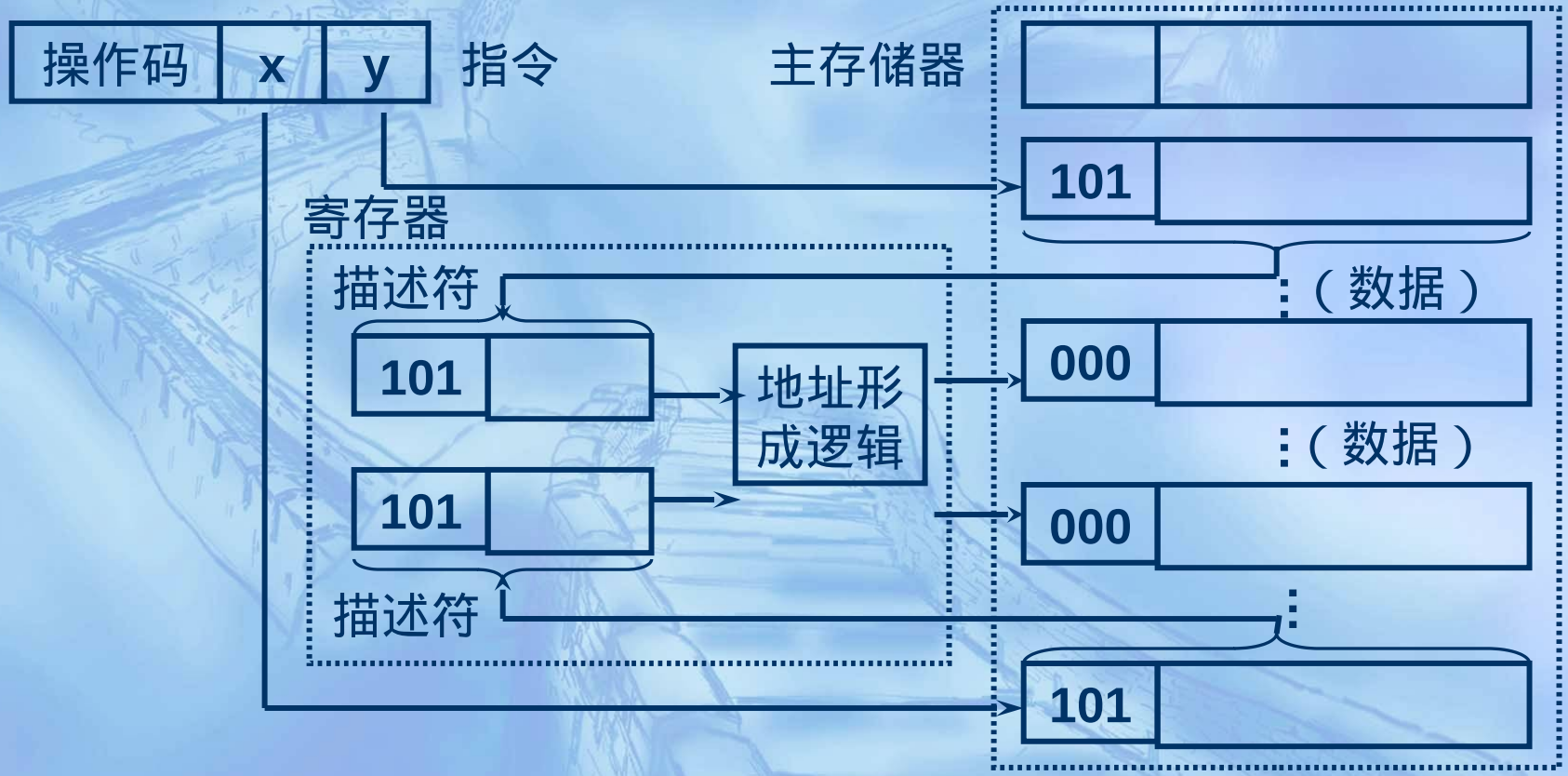
- 数据描述符格式

101	标志位	数据块长度	数据块起始地址

- 数据格式

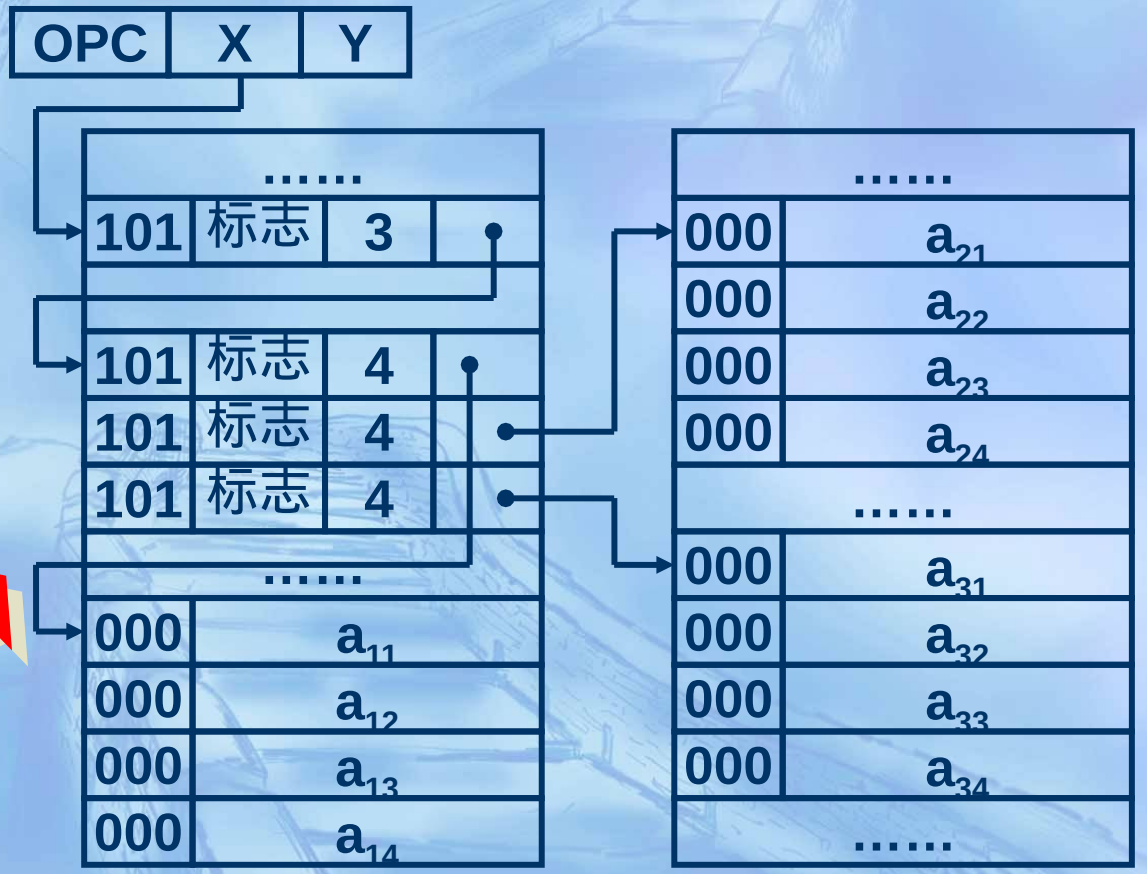
000	数值

取操作数的过程



描述二维阵列（按行存储）

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}$$



特 点

- 优点
  - 整块数据可一次性操作
  - 简化编译中的代码生成
- 比较

标志符是和每个数据相连，合存于一个存储单元中，描述单个数据的类型特征；描述符是和数据分开存放的，专门用来描述一组数据。

数据表示设计

确定哪些数据类型用数据表示来实现的原则主要有：

- 原则一
- 原则二

设计原则一

是否提高系统的效率，即：是否减少了实现时间和所需的存储空间。

- 实现时间的衡量
  - 主存和处理机间所需传送的消息量有否减少
  - 高速运算部件是否节省了时间
  - 是否节省了大量的辅助操作（由硬件完成）
  - 是否节省了编译所需要的时间
- 例子

向量数据表示的引入（例如：两个 **200×200** 的定点数矩阵相加）。

设计原则二

通用性和利用率是否高。

- **堆栈机**

  尽管堆栈操作速度很快，但矩阵运算效率却降低了。

- **阵列机**

  还需要解决通用性问题、如何高效地实现不同的数据结构、如何确定阵列型数据表示的规模等。

- **树型结构机器**

  堆栈、向量、链表等结构的实现低效。

寻址技术

概念

寻址技术就是寻找操作数及其他信息的地址的技术，它是软件和硬件的一个主要分界面。

对象

寄存器、主存储器、堆栈和输入输出设备。

内容

- 编址方式
- 寻址方式
- 定位方式

方法

分析各种寻址技术的优缺点，如何选择和确定寻址技术。

编址方式

- 编址单位
- 零地址空间个数
- I/O 设备的编址技术
- 并行存储器的编址技术

编址单位

- 字编址
- 字节编址
- 位编址

字编址

- 方法

每个编址单位与设备的访问单位相一致。即，每个编址单位所包含的信息量（例：二进制位数）与访问一次设备（指读或写）所获得的信息量是相同的。

- 优点

实现很简单，地址信息、存储器容量等没有任何浪费。

- 缺点

没有对非数值计算的应用（要求按字节编址，因为它的基本寻址单位是字节）提供支持。

字节编址

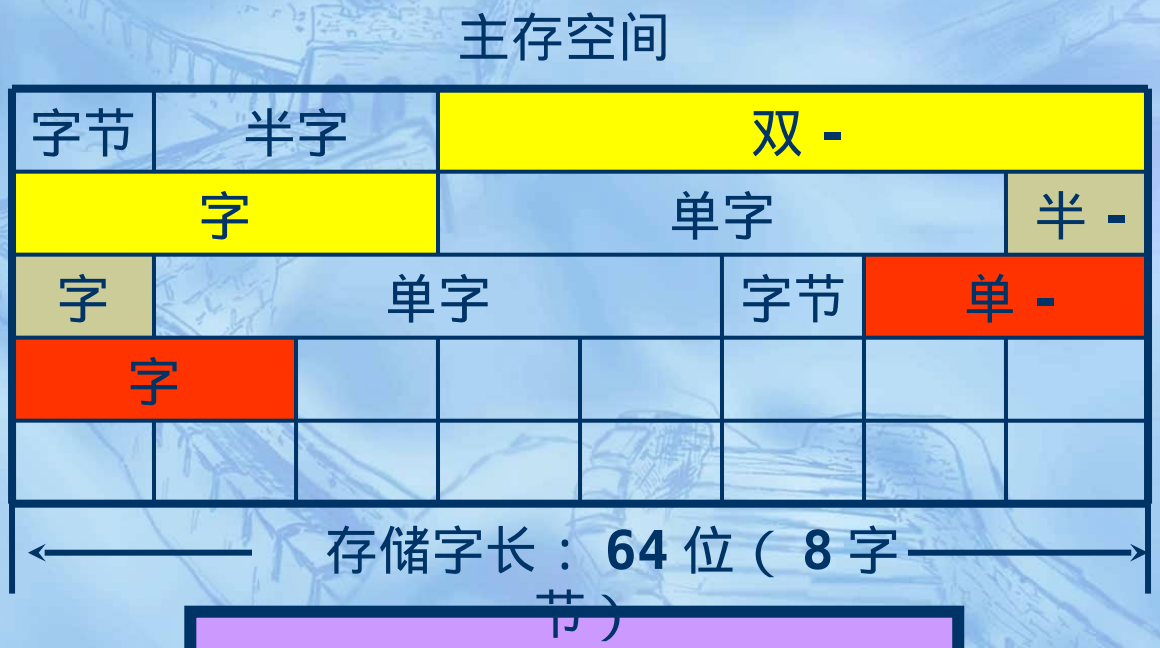
- **问题**

因为每个编址单位所包含的信息量（一个字节）与访问一次设备所获得的信息量（通常是一个字：字节的 4 倍以上）不相同，从而就产生了数据如何在存储器里存放的问题。

- **解决**

有三种：（**1**、**2**、**3**）

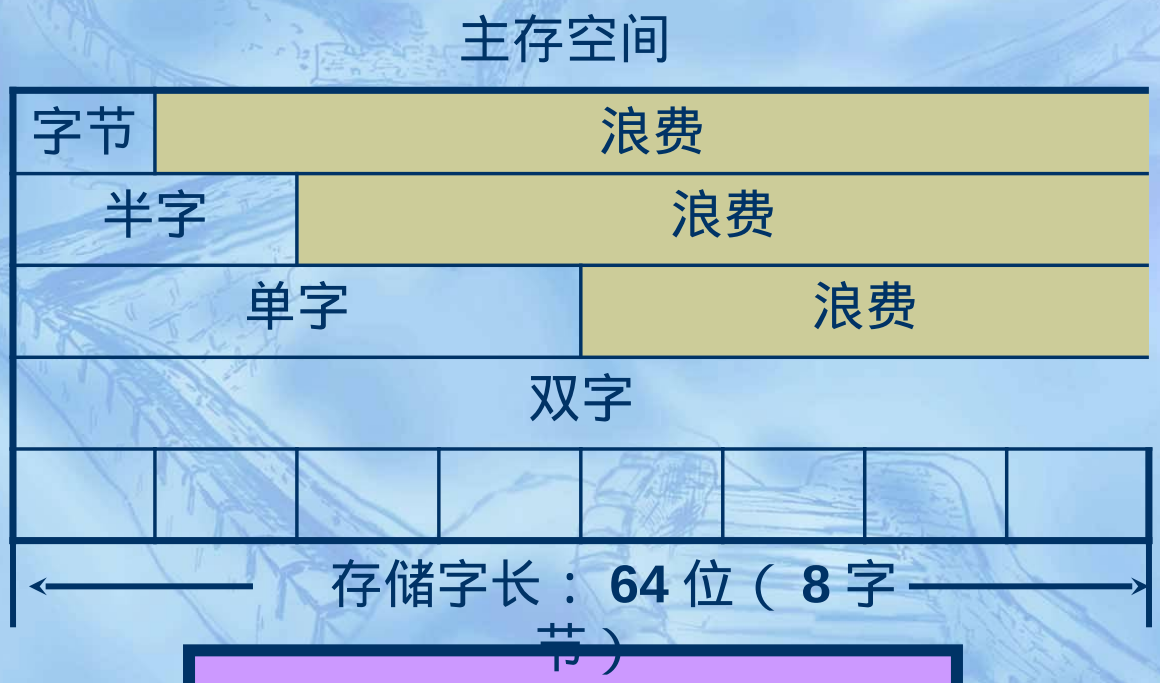
从任意位置开始存储



- 优点
  - 不浪费存储器资源。

- 缺点
  - 增加读取时间（可能需要两个存储周期）
  - 存储器的读写控制比较复杂

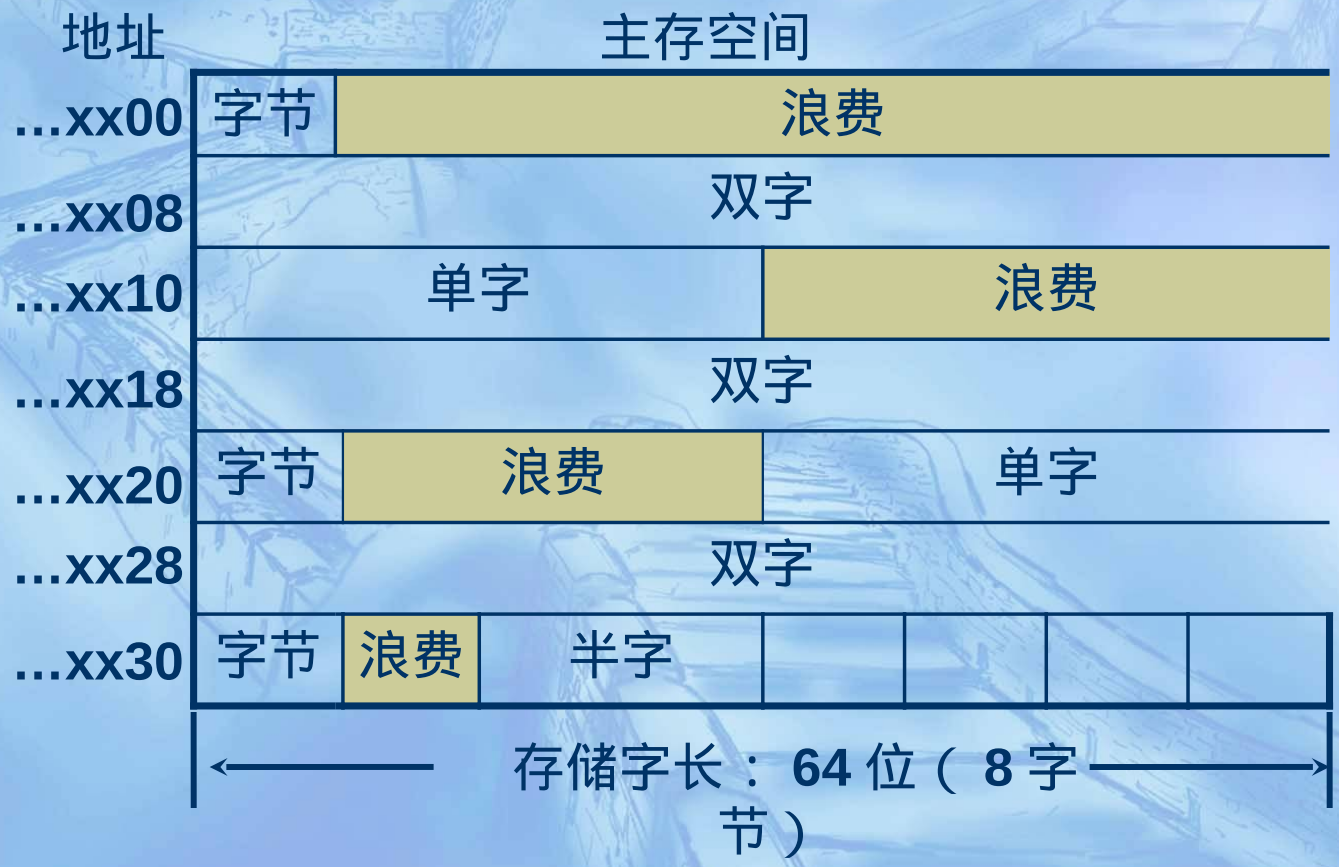
从存储字起始位置开始存储



- 优点
  - 读取都可以在一个存储周期内完成
  - 存储器的读写控制比较简单

- 缺点
  - 浪费存储器资源。

从地址整数倍位置开始存储



性能是前两种方法的折中

位编址

- **方法**

  每个编址单位是位。

- **特点**

  同字节编址，只不过地址信息的浪费更大。

早期的计算机大多采用字编址，目前使用最普遍的是字节编址，而位编址是一种很有应用前景的编址方式。

几种编址设备的性能比较

存储设备	容量	工作速度	寻址种类	编址方式
通用寄存器	小	很快	单一直接寻址	单字或双字
主存储器	大	较慢	多种寻址方式	字、字节或位
输入输出设备	较小	很慢	单一直接寻址	多种编址
堆栈	较小	较慢	隐含寻址	不编址

零地址空间个数

- 三个零地址空间
- 两个零地址空间
- 一个零地址空间
- 没有零地址空间

三个零地址空间

- 通用寄存器、主存储器和输入输出设备均独立编址。
- 优点：能够满足不同存储设备的需要
- 缺点：指令系统中通常要有三类操作指令，例如：三类的数据传送指令、运算指令、移位指令等
  - **RISC 计算机**：为简化指令系统，一般规定所有的运算、移位和测试等操作只能在通用寄存器中进行，只有访问操作在三类存储设备中都能进行
  - **CISC 计算机**：通常所有操作都能在三类存储设备上进行

两个零地址空间

- 通用寄存器单独编址，主存储器与输入输出设备统一编址。
- 优点：能够简化指令系统，因为无需另外设置 I/O 指令
- 缺点：指令执行过程复杂，执行时间要加长；因为所有访问主存储器的操作都需判断是否访问 I/O 设备
  - RISC 计算机：尤为不利，因为具体操作都需涉及 Load/Store 指令。
  - CISC 计算机：对于 I/O 设备许多复杂的运算指令根本用不上

一个/没有零地址空间

- 一个零地址空间

所有存储设备统一编址，最低端是通用寄存器，最高端是输入输出设备，中间为主存储器。

此类计算机中通常有一个存储容量比较大的高速存储器，有多个进程或线程存放在高速存储器中，并且可以通过硬件进行进程或线程的切换。

- 没有零地址空间（隐含编址方式）

所有存储设备都无需编址，例如：堆栈、 **Cache** 等。

I/O 设备的编址技术

- 一台设备一个地址
  - 对 I/O 设备本身进行编址，没有对寄存器进行编址。
  - 通过指令中的操作码来识别该 I/O 设备上的有关寄存器：通常要设置对数据寄存器进行读写的指令和设置对控制寄存器或状态寄存器进行操作的指令等。
  - 应用：早期广泛应用，现在许多微型、小型机普遍使用。
  - 优点： I/O 设备的地址很规整，地址码的长度很短
  - 缺点：指令系统比较复杂

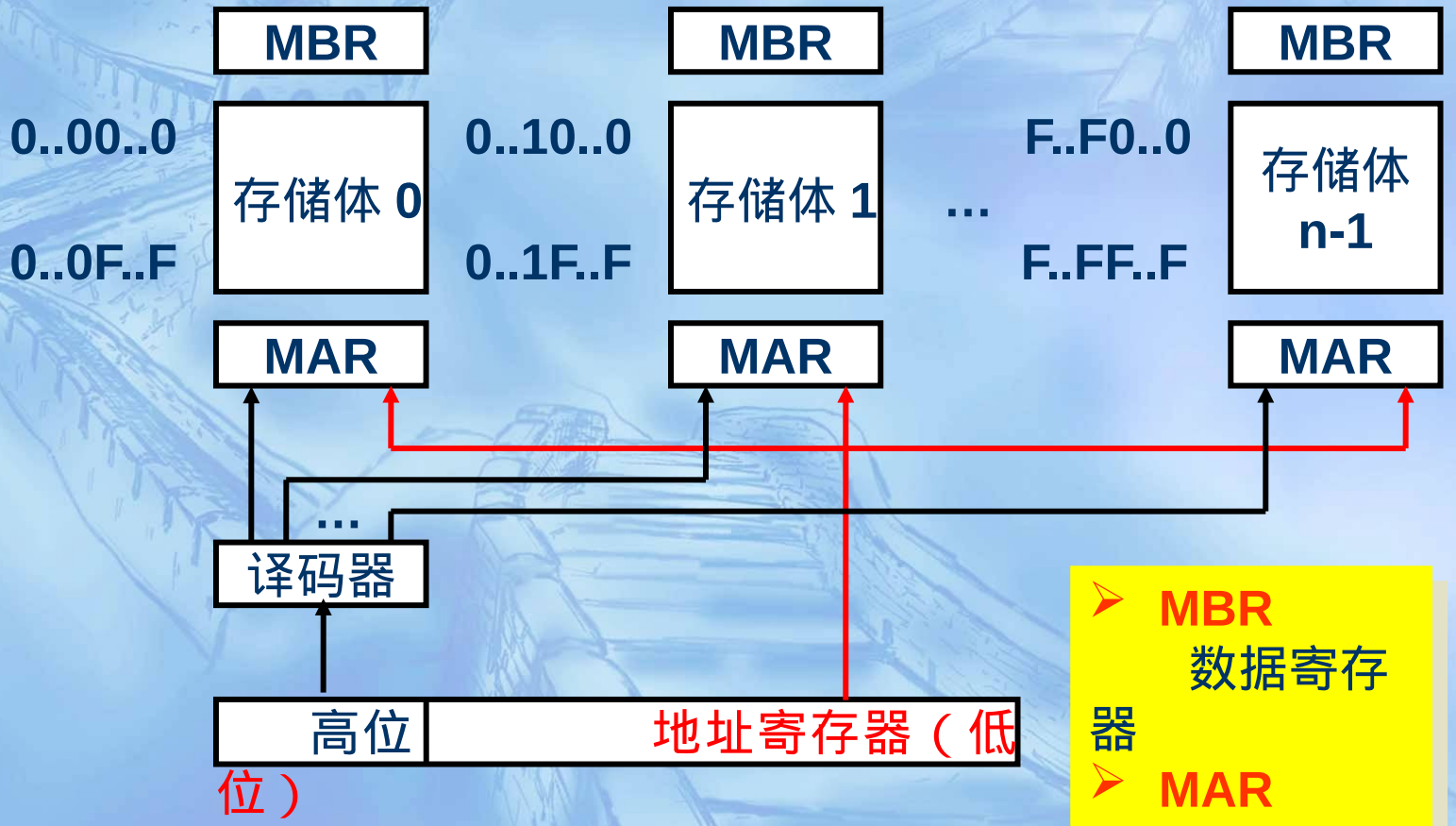
I/O 设备的编址技术

- 一台设备两个地址
  - 一个地址是数据寄存器，另一个是状态或控制寄存器。
  - 应用：适用于大多数 I/O 设备，因为绝大多数 I/O 设备的接口上只有这两个要编址的寄存器。
  - 特点：虽然设备地址扩大了一倍，但是用于 I/O 的指令系统简单了。
- 一台设备多个地址
  - 根据 I/O 设备的不同需要为它分配不同数据的地址。
  - 应用：适用于主存与 I/O 设备合用一个零地址空间的计算机系统中。

并行存储器的编址技术

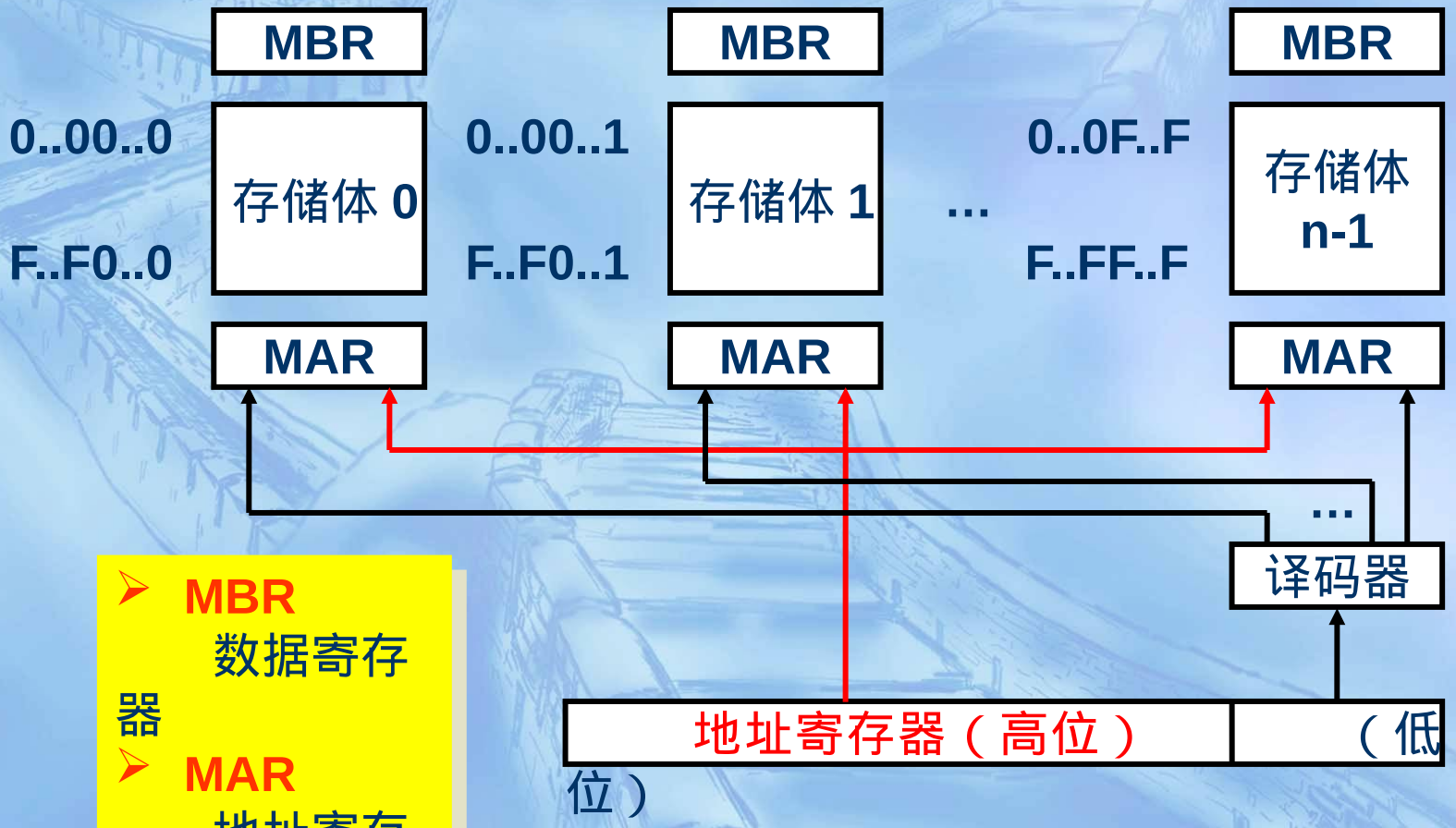
- 高位交叉编址
  - 目的：用来扩大存储器容量。
  - 优点：模块化结构好，用户可以很方便地扩充自己的主存储器。
  - 缺点：控制部件的数量增加了。
- 低位交叉编址
  - 目的：提高存储器速度，存储器容量也相应增加了。
  - 在一个存储周期内，n 个存储体必须分时启动，采用流水线工作
  - 缺点：存在访问冲突问题

高位交叉编址



- MBR 数据寄存器
- MAR 地址寄存器

低位交叉编址



- MBR 数据寄存器
- MAR 地址寄存器

寻址方式

- 常用寻址方式
- 寻址方式选择
- 寻址方式参数大小选择

常用寻址方式的含义和使用

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	Regs[R4]←Regs[R4] + Regs[R3]	When a value is in a register.
Immediate	Add R4,#3	Regs[R4]←Regs[R4]+3	For constants.
Displacement	Add R4,100(R1)	Regs[R4]←Regs[R4] + Mem[100+Regs[R1]]	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	Add R4,(R1)	Regs[R4]←Regs[R4] + Mem[Regs[R1]]	Accessing using a pointer or a computed address.
Indexed	Add R3,(R1 + R2)	Regs[R3]←Regs[R3] +Mem[Regs[R1]+Regs[R2]]	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1,(1001)	Regs[R1]←Regs[R1] + Mem[1001]	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1,@(R3)	Regs[R1]←Regs[R1] + Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer p, then mode yields $*p$.
Autoincrement	Add R1,(R2)+	Regs[R1]←Regs[R1] + Mem[Regs[R2]] Regs[R2]←Regs[R2]+d	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d.
Autodecrement	Add R1,-(R2)	Regs[R2]←Regs[R2]-d Regs[R1]←Regs[R1] + Mem[Regs[R2]]	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	Regs[R1]← Regs[R1]+ Mem[100+Regs[R2] + Regs[R3]*d]	Used to index arrays. May be applied to any indexed addressing mode in some computers.

寻址方式选择

- 问题

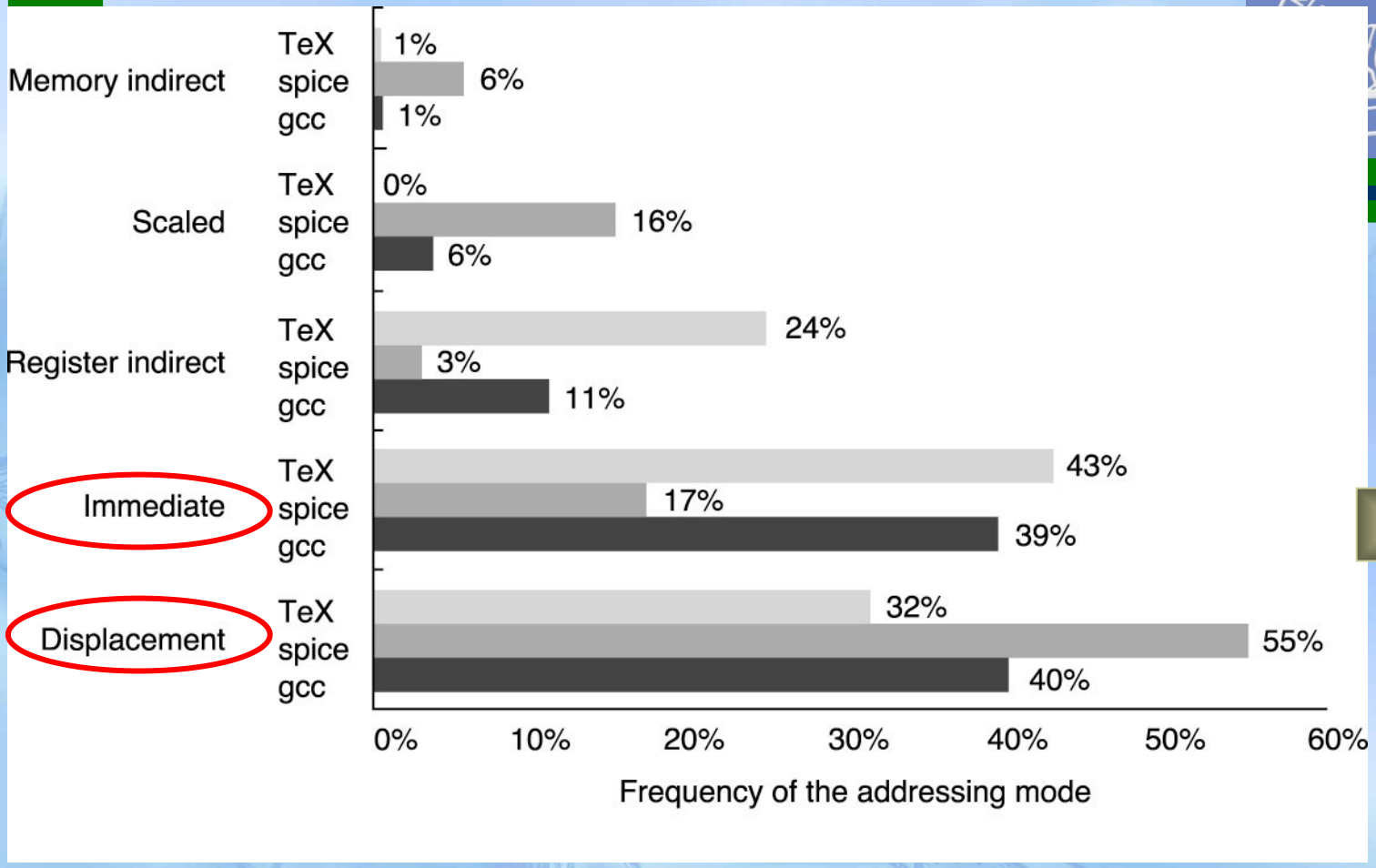
采用多种寻址方式可以显著减少程序的指令条数，但会增加计算机的实现复杂度以及指令的 **CPI** 。

- 选择方法

使用频度分析法根据指令系统风格和各种寻址方式的使用频率，选择高频率的寻址方式。

- 例子

在 **VAX** 指令集机器上运行 **gcc** 、**Spice** 和 **Tex** 基准程序，各种寻址方式的分布见图。



结论：偏移寻址和立即数使用频率很高，必须支持这两种方式；对其他寻址方式，则应根据软、硬取舍原则进行选择。

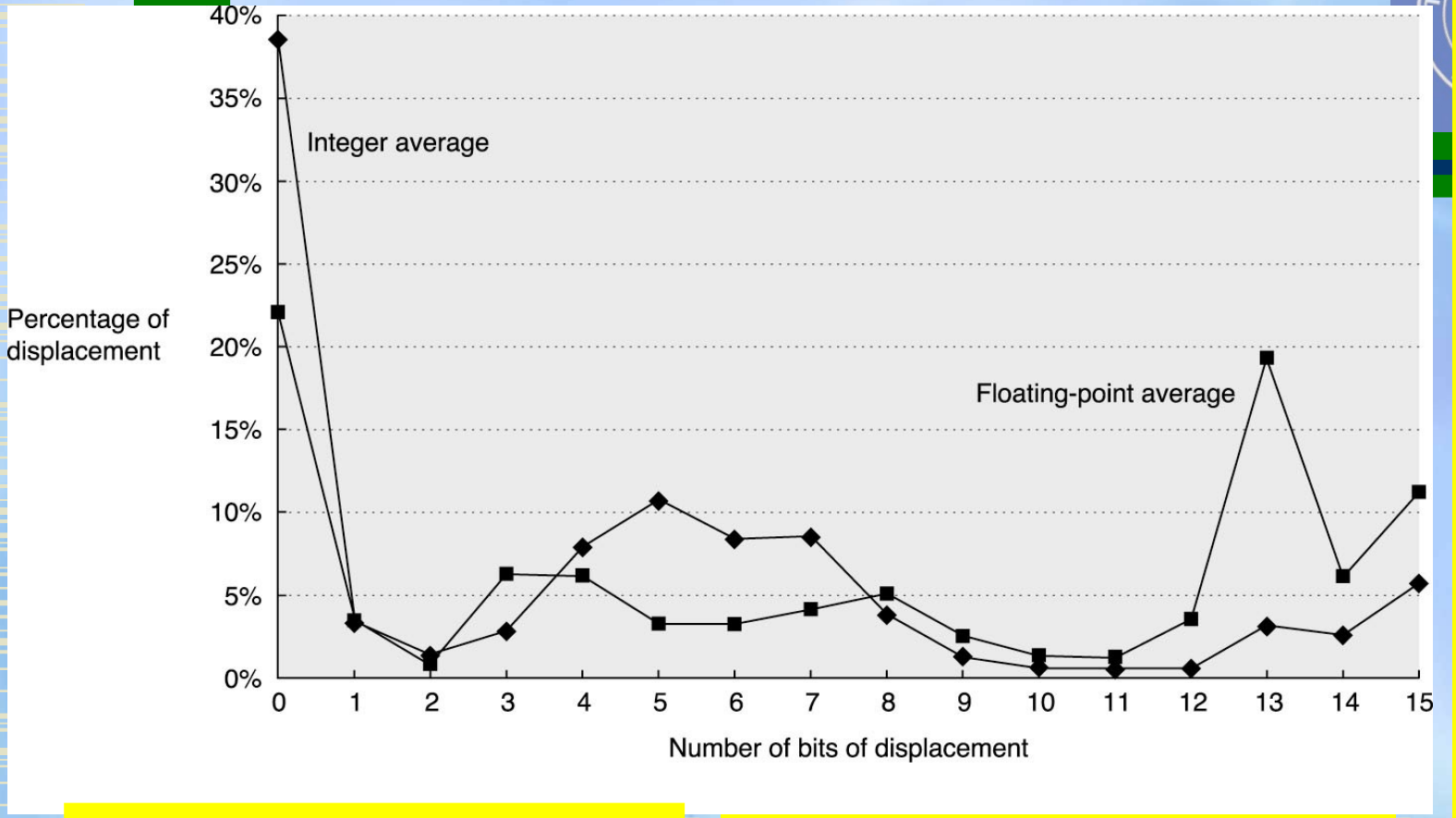
寻址方式参数大小选择

- 选择方法

  根据频度分析法进行分析和选择。

- 例子
  - 偏移寻址参数大小选择
  - 立即数参数大小选择

在 Alpha 体系结构计算机上使用 SPEC CPU2000 测试所得结果



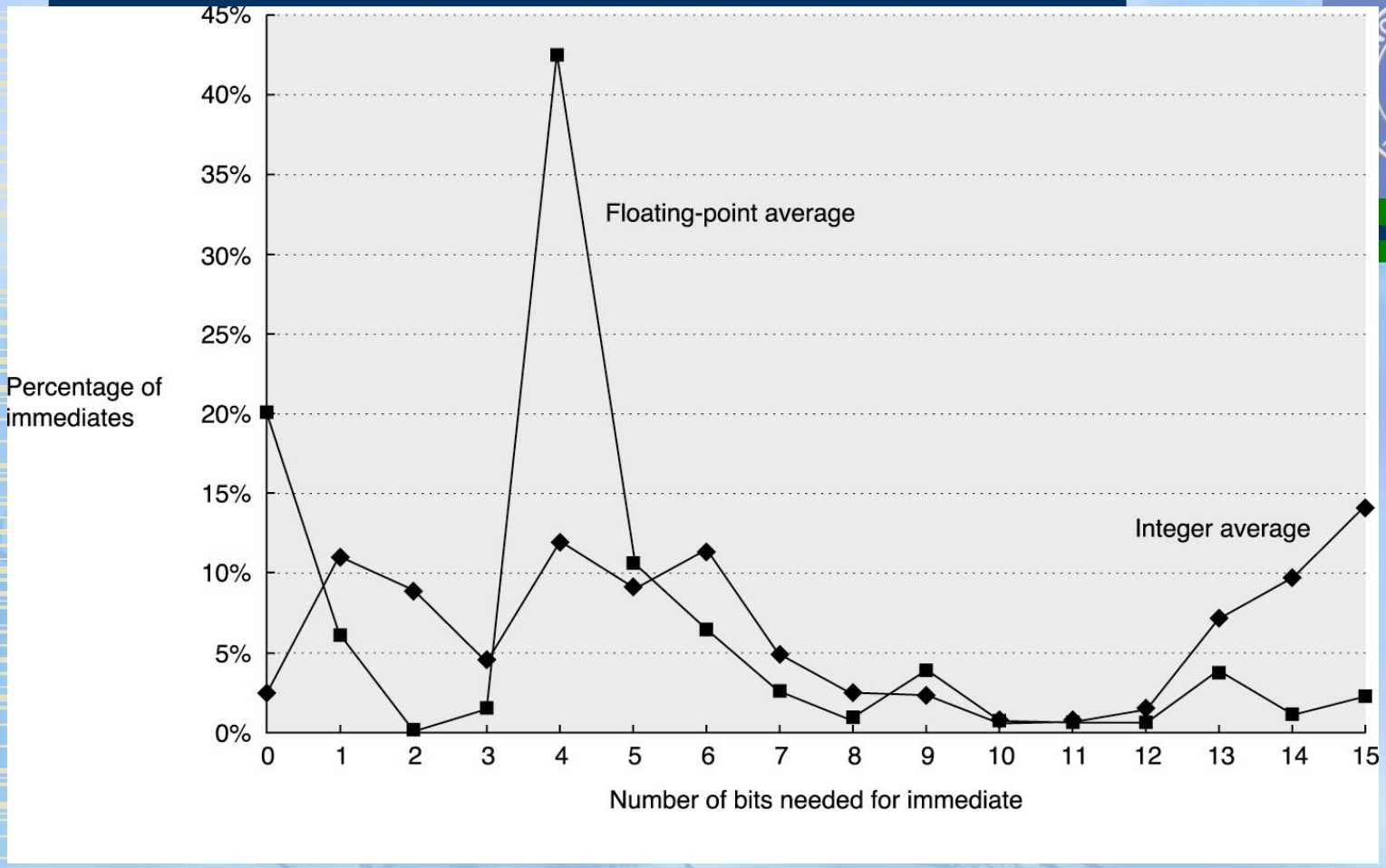
问题：

偏移的范围如何确定，因为其大小直接影响指令的长度。

结论：

偏移寻址方式中的地址长度至少为 **12 ~ 16bit（ 75% ~ 99 % ）**。

在 Alpha 体系结构计算机上使用 SPEC CPU2000 测试所得结果



问题：

立即数的取值范围如何确定，因为其大小直接影响指令的长度。

结论：

立即数寻址模式中的立即数字段长度至少为 **8 ~ 16bit（ 50% ~ 80% ）**。

定位方式

- **直接定位方式**

在程序装入主存储器之前，程序中的指令和数据的主存物理地址就已经确定了的称为直接定位方式。

- **静态定位方式**

在程序装入主存储器的过程中随即进行地址变换，确定指令和数据的主存物理地址的称为静态定位方式。

- **动态定位方式**

在程序执行过程中，当访问到相应的指令或数据时才进行地址变换，确定指令和数据的主存物理地址的称为动态定位方式。

指令格式的优化设计

- 指令的组成
- 操作码的优化设计
- 地址码的优化设计
- 指令字格式的优化设计

指令的组成

指令一般由两部分组成：操作码和地址码。

- 操作码

  由两部分组成：
  - 指令的操作种类
  - 所用操作数的类型

- 地址码

  由三部分组成：
  - 操作数地址
  - 地址的附加信息
  - 寻址方式

指令格式优化设计的目标

- 节省程序的存储空间
- 指令格式要尽量规整，以减少硬件译码的复杂度

操作码的优化设计

- 评价方法
- 固定长度操作码
- **Huffman** 编码
- 扩展编码法

评价方法

- 编码的平均码长 l

$$l = \sum_{i=1}^{n} p_i \cdot l_i$$

其中：

p_i: 表示第 i 种操作码在程序中出现的概率；

l_i: 表示第 i 种操作码的编码长度；

n: 操作码的总数。

- 编码的信息冗余量 R

$$R = 1 - \frac{H}{l}$$

其中：

H: 信息熵；（理论上的最短平均码长）。

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

例 子

为增加可比性，对下面介绍的编码方法采用同一例子：假设一台模型计算机，共有 7 种不同的指令，已知各种指令在程序中出现的概率如下表。

指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
概率	0.45	0.30	0.15	0.05	0.03	0.01	0.01

固定长度操作码

- 思想

所有指令的操作码长度都是相同的。如果需要编码的指令有 n 条，则固定长度操作码的位数至少需要 $\lceil \log_2 n \rceil$ 位。目前许多的 RISC 采用该思想。

- 例子

H=1.95 ； l=3 ； R=35%

- 特点

优点：非常规整，硬件译码也很简单。

缺点：浪费严重。

Huffman 编码

- 思想

概率高的用短位数表示，概率低的用长位数表示。

- 算法

利用 **Huffman** 树实现。

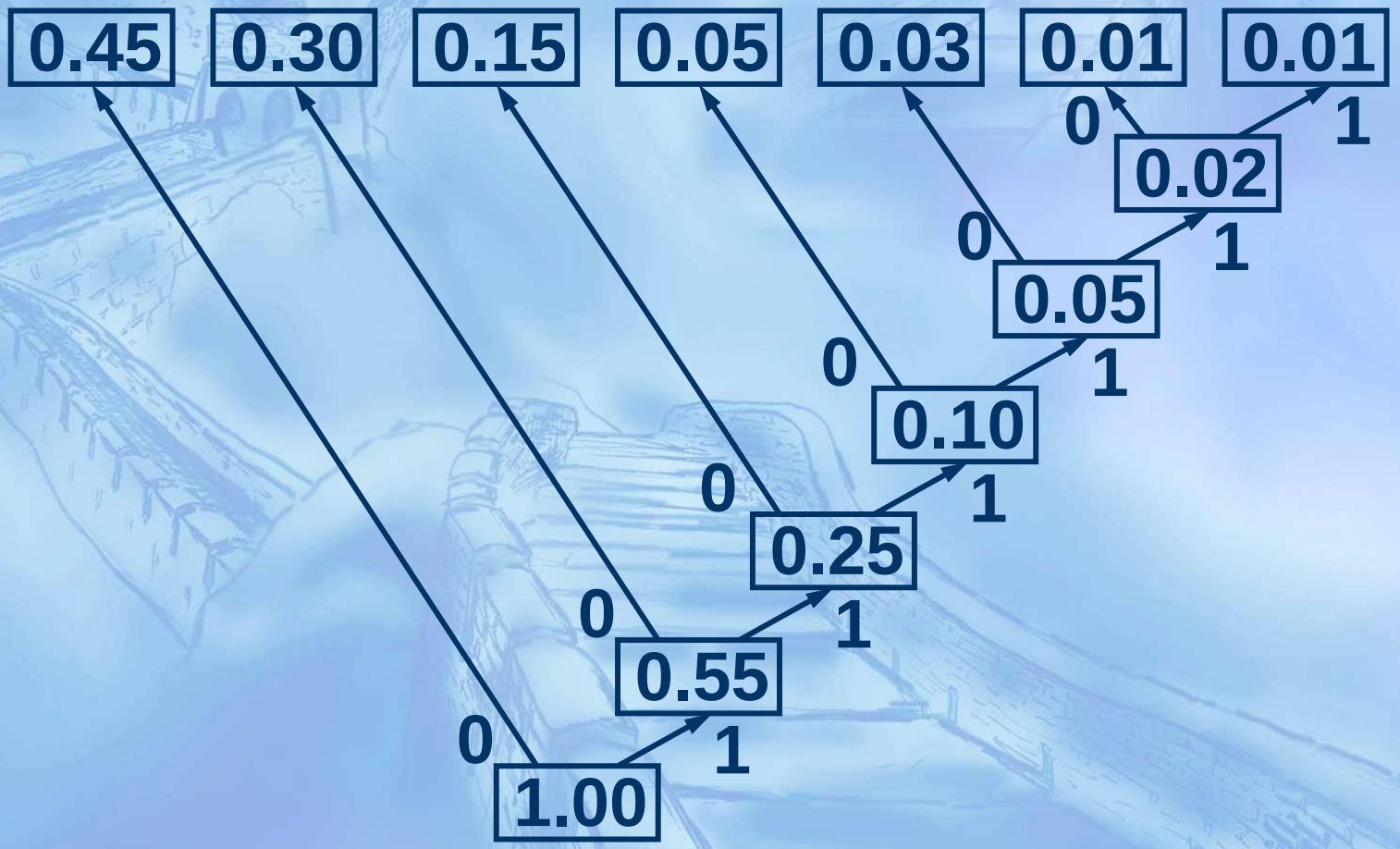
- 特点

是最优化的编码方式（平均码长最短，信息的冗余量最小），但操作码很不规整。

Huffman 编码算法

1. 把所有指令按照使用概率自左向右排列好。
2. 选取两个概率最小的结点合并成一个概率值是二者之和的新结点，并把这个新结点与其它还没有合并的结点一起形成新结点集合。
3. 在新结点集合中选取两个概率最小的结点进行合并，如此继续进行下去，直至全部结点合并完毕。
4. 最后得到的根结点的概率值为 1 。
5. 每个结点都有两个分支，分别用“ 0” 和“ 1” 表示。
6. 从根结点开始，沿尖头所指方向，到达属于该指令的概率结点，把沿线所经过的代码组合起来得到这条指令的操作码编码。

Huffman 编码举例



Huffman 编码举例

指令序号	概率	Huffman 编码法	操作码长度
I_1	0.45	0	1 位
I_2	0.30	10	2 位
I_3	0.15	110	3 位
I_4	0.05	1110	4 位
I_5	0.03	11110	5 位
I_6	0.01	111110	6 位
I_7	0.01	111111	6 位

H=1.95 ； l=1.97 ； R≈1%

扩展编码法

- 思想

是固定长度操作码和 **Huffman** 编码法相结合形成的。即：先根据指令使用频率的宏观分布，将指令分成有限几类；然后根据 **Huffman** 编码原理，对使用频率高的指令类采用短位数，使用频率低的指令类采用长位数；同一类指令内采用固定长度操作码。

- 问题

有多种扩展编码，等长扩展（例如：**4-8-12**、**3-6-9** 等）？不等长扩展（例如：**4-6-10** 等）？

- 解决

取决于具体指令的使用频度的分布。

扩展编码法举例

序号	概率	1-2-3-5 扩展编码	2-4 等长扩展编码
I_1	0.45	0	00
I_2	0.30	10	01
I_3	0.15	110	10
I_4	0.05	11100	1100
I_5	0.03	11101	1101
I_6	0.01	11110	1110
I_7	0.01	11111	1111
平均码长		2.0	2.2
信息冗余量		2.5%	11.4%

等长 (4-8-12)15/15/15 扩展编码法

操作码编码	说明
0000 0001 …… 1110	4 位长度的操作码 共 15 种
1111 0000 1111 0001 …… 1111 1110	8 位长度的操作码 共 15 种
1111 1111 0000 1111 1111 0001 …… 1111 1111 1110	12 位长度的操作码 共 15 种

等长 (4-8-12)8/64/512 扩展编码法

操作码编码	说明
0000 0001 …… 0111	4 位长度的操作码 共 8 种
1000 0000 1000 0001 …… 1111 0111	8 位长度的操作码 共 64 种
1000 1000 0000 1000 1000 0001 …… 1111 1111 0111	12 位长度的操作码 共 512 种

不等长 (4-6-10) 扩展编码法

编码方法	各种不同长度操作码的指令种类			总的指令种类
	4 位操作码	6 位操作码	10 位操作码	
15/3/16	15	3	16	34
8/31/16	8	31	16	55
8/30/32	8	30	32	70
8/16/256	8	16	256	280
4/32/256	4	32	256	292

地址码的优化设计

- 地址个数的选择
- 优化单个地址码

地址个数的选择

地址数目	指令条数	程序存储量	程序执行速度	适用场合
三地址	少	最大	一般	向量、矩阵运算为主
二地址	一般	很大	很低	一般不宜采用
一地址	较多	较大	较快	连续运算，硬件结构简单
零地址	最多	最小	最低	嵌套、递归、变量较多
二地址 R 型	一般	最小	最快	多累加器、数据传送较多

采用各种不同地址数指令编写的程序的特点和适用范围

优化单个地址码

- 目的

要用一个较短的地址码表示一个较大的逻辑地址空间，同时还要有较灵活有效的寻址方式。

- 常用方法
  - 用间址寻址方式缩短地址码长度
    - 在主存储器的低端（地址码长度可以很短）开辟出一个专门用来存放地址的区域，用于间址寻址
    - 例：在一个按字节编址、存储字长度为 **64bit** 的主存储器最低端的 **1KB** 之内有一个用来存放地址码的区域，可以实现在指令中只要用 **7bit** 就能访问一个 **64bit** 长的逻辑地址。

优化单个地址码

- 目的

  要用一个较短的地址码表示一个较大的逻辑地址空间，同时还要有较灵活有效的寻址方式。

- 常用方法
  - 用变址寻址方式缩短地址码长度
    - 基地址 + 地址偏移量
    - 由于程序的局限性，地址偏移量可以比较短；地址比较长的基地址（例如：**64bit**）通常放在变址寄存器中，在指令的地址码中只需给出比较短的地址偏移量，通常只需十多位或二十位左右即可

优化单个地址码

- 目的

  要用一个较短的地址码表示一个较大的逻辑地址空间，同时还要有较灵活有效的寻址方式。

- 常用方法
  - 用寄存器间接地址方式缩短地址码长度（是最有效的方法）
    - 寄存器中存放的是操作数地址
    - 例：有 **8** 个用于间接寻址的寄存器，每个寄存器的长度为 **64bit** ，这样用一个 **3bit** 的地址码就能访问一个 **64bit** 的逻辑地址

指令字格式的优化设计

- 问题

操作码和地址码的优化，会造成指令字的不定长，无法同时满足速度快和空间省。

- 解决

合理结合，形成定长或多种长度的指令字，例如：长操作码配短地址码等。

- 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

- 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。
  - VAX 和 80x86 采用了这种编码格式

操作码	地址描述符 1	地址码 1	…	地址描述符 n	地址码 n

寻址方式通过设置专门的地址描述符给出，因为指令有多个操作数，无法在操作码中进行编码。

定长编码格式

- 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（如 **Alpha**、**MIPS**、**SPARC** 等）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

- 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。
  - 例如： **IBM 360/370**

指令系统的功能设计

- 指令系统设计的基本原则
- 基本指令系统
- 指令系统优化设计
  - 复杂指令系统计算机（ **CISC** ）
  - 精简指令系统计算机（ **RISC** ）

指令系统设计的基本原则

- 完整性

  是指作为通用计算机所应该具备的基本指令种类。

- 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

  是指指令的执行速度要快，使用频度要高。例如：在RISC 体系结构中，大多数指令都能在一个节拍内完成，而且只设置那些使用频度高的指令。

- 兼容性

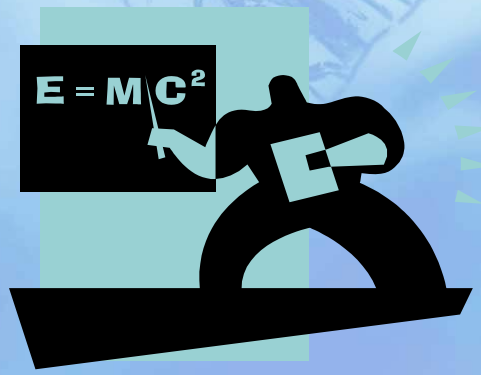
  是计算机系统的生命力之所在。

指令系统设计的基本原则

- 规整性

是硬件设计（如： VLSI 技术）和软件设计（如：编译程序）的需要。

规整性主要包括：对称性和均匀性。



对于规整性的要求必须有所选择。

指令系统设计的基本原则

- 对称性
  - 是指各种与指令系统有关的数据存储设备的使用、操作码的设置等都要对称。
  - 例 1 ：所有的通用寄存器要同等对待
  - 例 2 ：如果设置 A-B 指令，则也应该设置 B-A 指令
- 均匀性
  - 是指对于各种不同的数据类型、字长、操作种类和数据存储设备，指令的设置要同等对待。
  - 例如：某机器有 5 种数据表示、 4 种字长、 8 种数据存储设备，则在设计加法指令时，指令种类应该是： 5×4×8 = 160 种两地址加法指令（不可能也不太现实）。

基本指令系统

在设计通用计算机时，指令系统的完整性是必须要考虑的：前 4 类是基本指令，后 4 类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

复杂指令系统计算机（CISC）

- 思想

增强原有指令的功能以及设置更为复杂的新指令取代原先由软件子程序完成的功能，实现软件功能的硬化。

- 途径
  - 从面向目标程序的优化实现来改进指令系统
  - 从面向高级语言的优化实现来改进指令系统
  - 从面向操作系统的优化实现来改进指令系统
- 特点

指令集极其复杂，硬件难以实现。最具代表性的指令集： **x86** 指令集。

从面向目标程序的优化实现来改进

- 优化目的
  - 缩短目标程序的长度，即减少程序的空间开销
  - 缩短目标程序的执行时间，即减少程序的时间开销
- 优化思路

对大量已有的目标程序及其执行情况进行统计，按统计出的各种指令和指令串的使用频度来分析改进。

指令的使用频度

- 指令静态使用频度

是指对多种典型程序源代码中所用的指令和指令串进行统计所得出的百分比（着眼于减少目标程序所占用的存储空间）。

- 指令动态使用频度

是指对多种典型程序在运行过程中所执行的指令和指令串进行统计所得出的百分比（着眼于减少目标程序的执行时间）。

统计结果表明：指令动态使用频度≈指令静态使用频度

优化方法

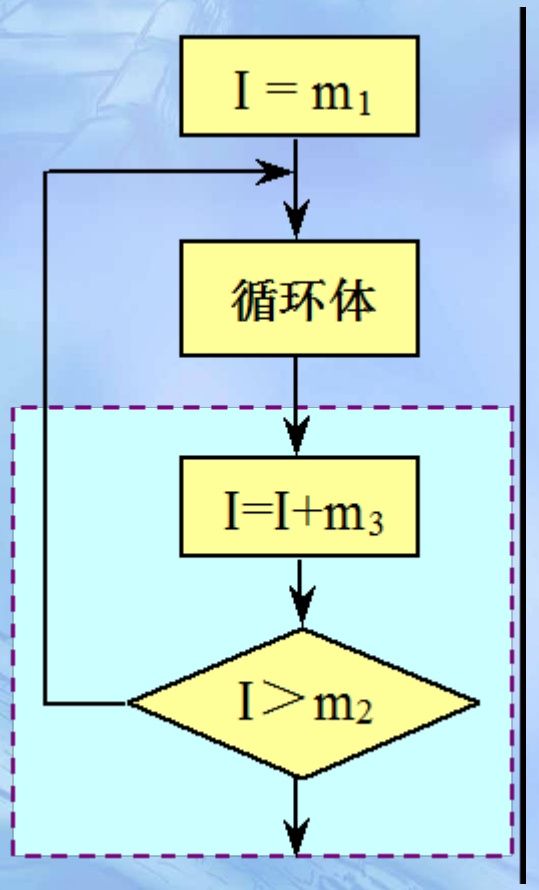
- 对使用频率高的指令，增强其功能、加快其执行速度并缩短字长；
- 对使用频率高的指令串，增设新指令来替代它；
- 对使用频率低的指令，取消或合并，但要考虑到指令系统的兼容性问题。

- 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 子

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（例： IBM 370 中专门设置了“大于转移”指令）。



从面向高级语言的优化实现来改进

- 优化目的

尽可能减小高级语言和机器语言之间的语义差距，以利于支持高级语言编译器（缩短编译器的代码长度及编译时间）的构造。

- 优化方法
  - 增强对高级语言支持的指令的功能
  - 高级语言计算机

增强对高级语言支持的指令的功能

- 思想

  对源程序中各种高级语言语句的静态 / 动态使用频度进行统计分析来改进。

- 例子
  - 一元赋值语句在高级语言中使用最频繁（ FORTRAN 中达 31% ），可优化数据传送指令来支持；
  - 条件转移（ IF ）和无条件转移（ GOTO ）语句所占比例较高（ ≥ 20% ），可增强转移指令的功能，增加转移指令的种类；
  - 增强系统结构的规整性，减少系统结构中的各种例外情况，是对编译程序的有力支持。

问题及解决

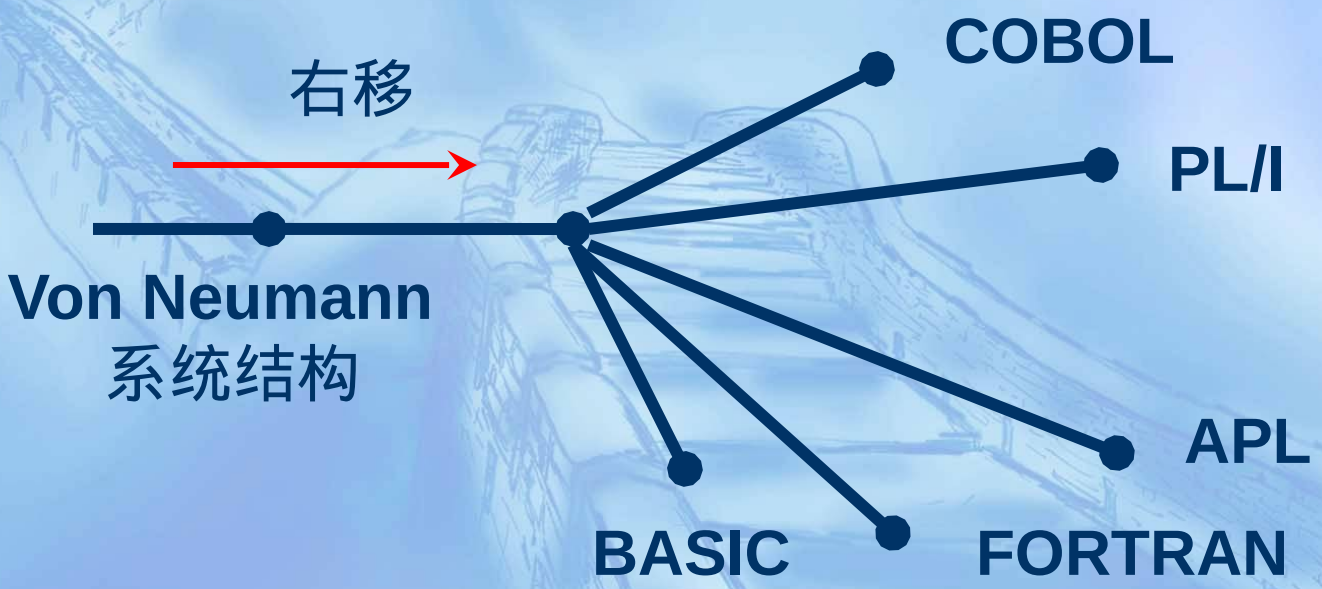
- 问题

如果计算机系统结构过分优化于一种高级语言的实现，就会显著减低与其语义结构相差较大的其它高级语言的实现效率。

- 解决
  - 同时面向各种高级语言的优化实现来改进
  - 动态自适应指令系统

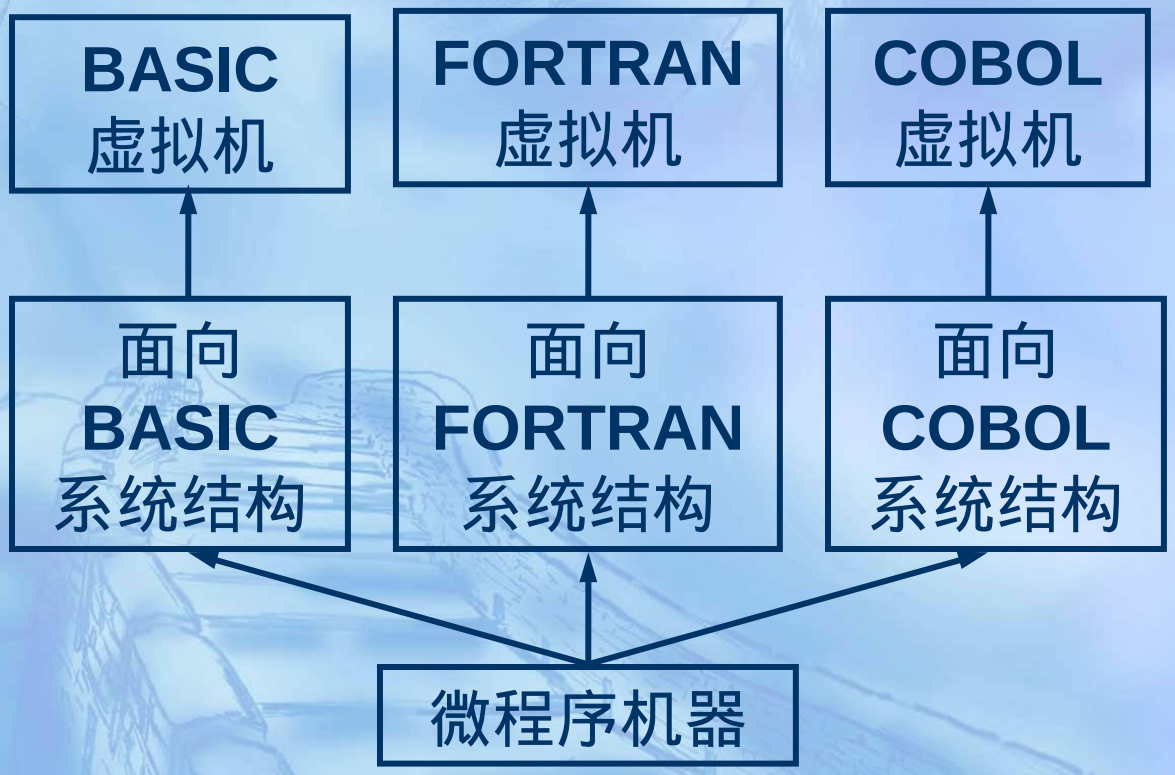
同时面向各种高级语言的优化实现来改进

可以设法改进指令系统，使它与各种高级语言的语义差距都有共同的缩小。



动态自适应指令系统

让机器具有分别面向各种高级语言的多种指令系统、多种系统结构，并能动态地切换。



高级语言计算机

- 思想

  让高级语言和机器语言之间无语义差距（增加微程序解释的工作，减少编译器翻译的工作）。

- 方法
  - 间接执行高级语言机器

    高级语言 = 汇编语言
  - 直接执行高级语言机器

    高级语言 = 机器语言

操作系统与计算机系统结构的关系

- 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。

从面向操作系统的优化实现来改进

- 优化目的
  - 减少运行操作系统所需的辅助操作时间；
  - 节省操作系统所占用的存贮空间。
- 优化方法
  - 对操作系统中常用的指令和指令串的使用频率进行统计和分析来改造；
  - 增设专用于操作系统的新指令；
  - 将操作系统中由软件子程序实现的某些功能改用硬件或固件实现。

精简指令系统计算机（RISC）

- CISC → RISC
- RISC 的定义
- RISC 的思想
- RISC 的关键技术
- RISC 的特点

CISC→RISC

RISC 思想的提出主要是因为 **CISC** 存在这样的一些问题：

- 指令系统复杂，不利于 **VLSI** 的设计，也不利于设计自动化技术的采用；
- 许多复杂指令的执行速度很低；
- 编译程序选择目标指令的范围大，很难优化；
- **20%** 和 **80%** 的规律： **80%** 的指令只在 **20%** 的运行时间里用到。

RISC 的定义

- Carnegie -Mellon 大学的定义
- IEEE 的 Michael Slater 的定义

至今没有一个确切的定义

Carnegie -Mellon 大学的定义

RISC 应有如下的特点：

- 大多数指令在单周期内完成；
- **LOAD/STORE** 结构；
- 硬布线控制逻辑；
- 减少指令和寻址方式的种类；
- 固定的指令格式；
- 注重编译优化技术。

IEEE 的 Michael Slater 的定义

RISC 处理器所设计的指令系统应使流水线处理能高效率执行，并使优化编译器能生成优化代码。

为实现前者应具有：

- 简单而统一格式的指令译码；
- 大部分指令可以单周期执行完成；
- 仅 **LOAD** 和 **STORE** 指令可以访问存储器；
- 简单的寻址方式；
- 采用延迟转移技术；
- 采用 **LOAD** 延迟技术。

为实现后者应具有：

- 三地址指令格式；
- 较多的寄存器；
- 对称的指令格式。

RISC 的思想

减少指令总数和简化指令的功能来降低硬件设计的复杂度，提高指令的执行速度。

特点：指令集简单，硬件实现容易

应用： **MIPS ·SPARC ·ARM ·Power PC ·RISC-V** 等

减少指令平均执行周期数（ **CPI** ）是 **RISC** 思想的精华！

RISC 的关键技术

- 硬件为主固件为辅
- 在 **CPU** 中设置数量较大的寄存器组

  采用重叠寄存器窗口技术提高使用效率。
- 指令的执行采用流水

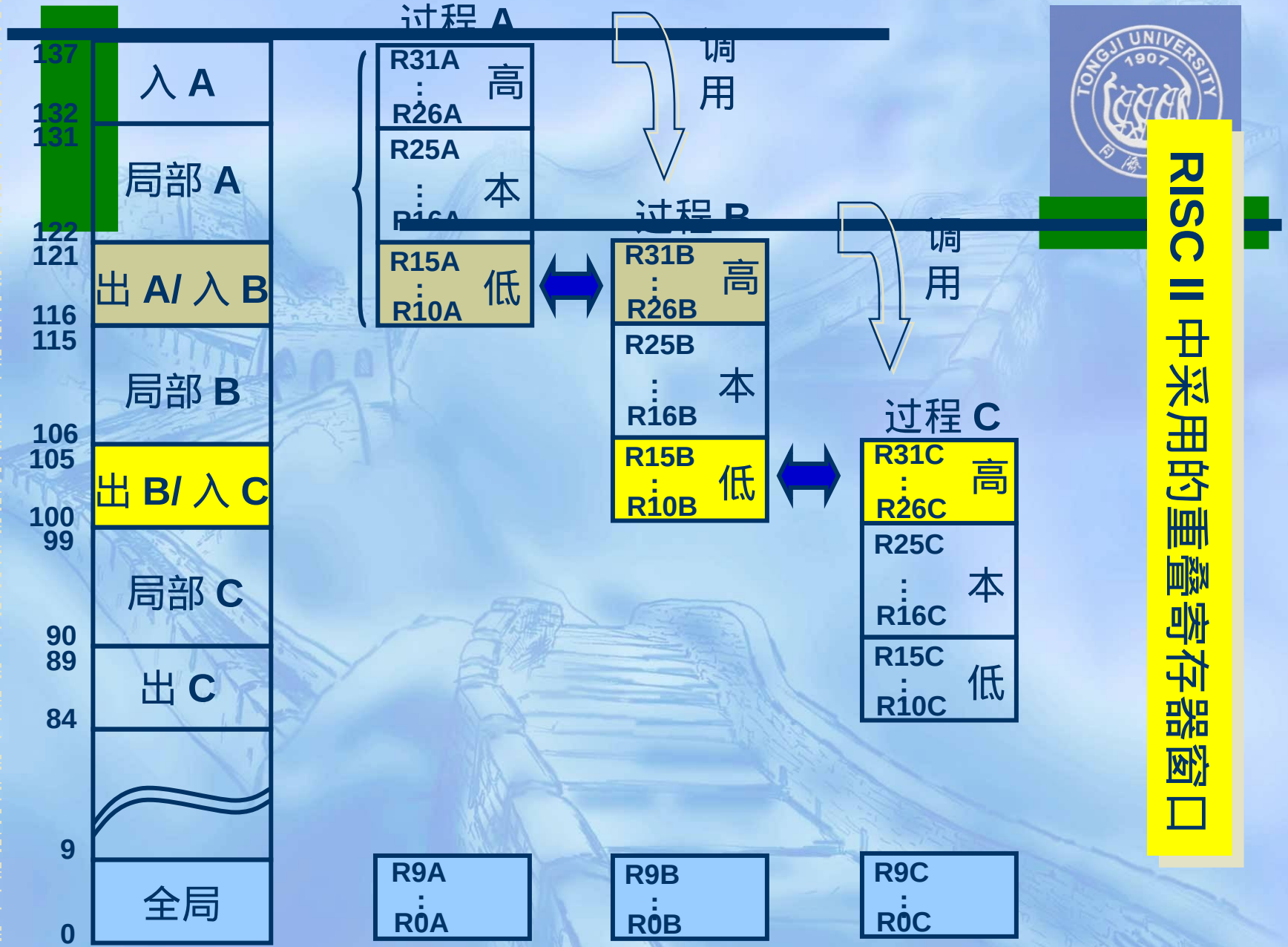
  采用延时转移技术和指令取消技术来降低（条件）转移指令对流水的影响。
- 采用认真设计和优化编译系统设计的技术

  例如：指令流调整技术。

重叠寄存器窗口技术

- 目标：缩短 CALL ，RETURN 操作时间。
- 方法：将设置的大量的寄存器，分成多个组和一个全局区；每个组中分高、本、低三个区；相邻组的高、低区重叠；不同的过程使用不同的组和共享全局区，这样可以加速参数与结果的传递。
- 例子： RISC II 中采用的重叠寄存器窗口，见后图。共有 138 个寄存器，分成 17 个窗口。
  - 1 个全局窗口： 10 个寄存器组成，能被所有过程访问；
  - 8 个局部窗口：各 10 个寄存器，作为 8 个过程的局部寄存器；
  - 8 个重叠窗口：各有 6 个寄存器，是相邻两个过程公用的；
  - 每个过程均可以访问 32 个寄存器。

RISC II 中采用的重叠寄存器窗口



寄存器窗口技术的效果

程序名称	Quick Sort	Puzzle
调用次数	111K	43K
最大调用深度	10	20
RISC II 溢出次数	64	124
RISC II 访存次数	4K	8K
VAX-11 访存次数	696K	444K

- Quick sort 程序的调用次数多，深度不大，而 Puzzle 程序正好相反。
- VAX-11、PDP-11、MC68000 为 CISC 计算机，RISC II 为 RISC 计算机

过程调用所需开销的比较

机器类型	执行指令条数	执行时间 (μs)	访问内存次数
VAX-11	5	26	10
PDP-11	19	22	15
MC68000	9	19	12
RISC II	6	2	0.2

重叠寄存器窗口技术

- 设计问题

寄存器设置多少个组？每个组多少个寄存器？

- 解决方法

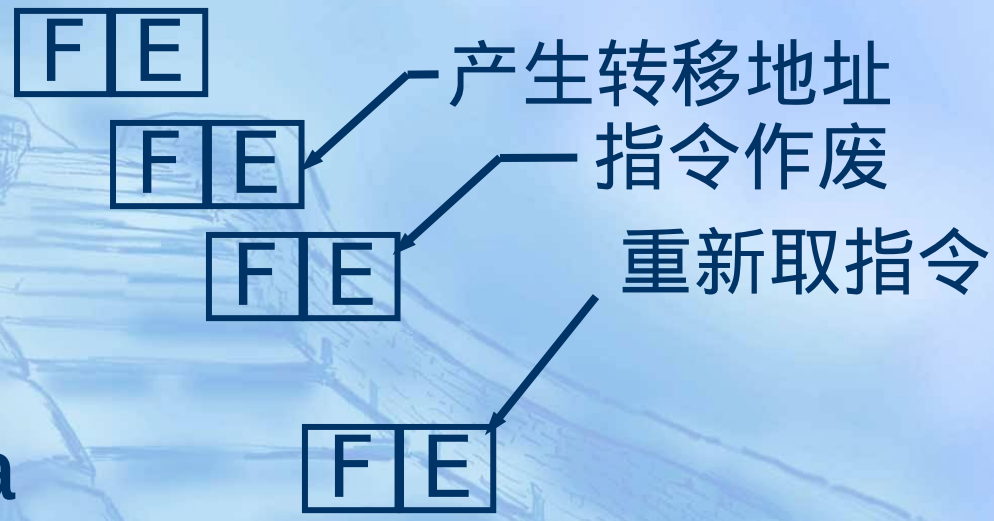
嵌套调用不超过 **8** 层，每层 **24** 个寄存器（每个区 **8** 个寄存器）可基本满足要求，不满足概率在 **1%** 左右（可使用主存缓冲）。

延时转移技术

- 问题

（条件）转移指令会引起流水线断流。

```
1:          add r1, r2
2:          jmp next2
3:  next1:  sub r3, r4
            ......
n:  next2:  move r4, a
```



延时转移技术

- 解决

  延时转移技术。

指令序列的调整由编译器自动进行

```
1:          jmp next2
2:          add r1, r2
3:  next1:  sub r3, r4
            ……
n:  next2:  move r4, a
```

F E ← 产生转移地址

F E ← 插入指令

F E ← 重新取指令

延时转移技术

- 思考：如何使用延时转移技术。

```
1:              move r1,r2
2:              cmp   r3,r4      ;(r3) 与 (r4) 比较
 3:             beq    next      ; 如果 (r3)=(r4), 则转移至 next
4:              add    r4,r5
                ……
n:  next:       move r4, a
```

```
1:              cmp   r3,r4      ;(r3) 与 (r4) 比较
 2:             beq    next      ; 如果 (r3)=(r4), 则转移至 next
3:              move r1,r2       ; 不能有数据相关，不能改变条件码
 4:             add    r4,r5
                ……
n:  next:       move r4, a
```

延时转移技术

- 分析

采用延迟转移技术的两个限制条件：

- 被移动指令在移动过程中与所经过的指令之间不能有数据相关；
- 被移动指令不破坏条件码，至少不影响后面的指令使用条件码。

如果找不到符合条件的指令，必须在（条件）转移指令后面插入空操作；如果指令的执行过程分为多个流水段，则要插入多条指令。

指令取消技术

- 问题

采用指令延时技术，遇到条件转移指令时，调整指令序列非常困难，在许多情况下找不到可以用来调整的指令。

- 解决

采用指令取消技术（为了提高程序的执行效率，应该尽量少取消指令）。

规则：如果是向后转移（转移目标地址小于当前程序计数器 PC 的值），则在转移不成功时取消下条指令，否则执行下条指令；如果是向前转移，则正好相反，在转移不成功时执行下条指令，否则取消下条指令。

例 1 ：向后转移（ loop 结构）

调整前	调整后
loop: X X X	X X X
Y Y Y	loop: Y Y Y
……	……
Z Z Z	Z Z Z
cmp r1, r2, loop	cmp r1, r2, loop
W W W	X X X
	W W W

分析：能够使指令流水线在绝大多数情况下不断流，因为绝大多数情况下，转移是成功的。

例 2 ：向前转移（ if-then-else 结构）

```
R R R
……
S S S
cmp r1, r2, thru ；若转移，则取消 TTT
T T T          ；若不转移，则执行 TTT
……
U U U
thru: VVV
```

分析：成功与不成功的概率通常各为 **50%** ，采用正常的指令取消技术即可。

指令流调整技术

- 目的

通过调整指令序列，变量重新命名等方法消除数据相关，提高流水线执行效率。

- 例子

调整后执行速度快一倍

调整前		调整后	
add	r1, r2, r3	add	r1, r2, r3
add	r3, r4, r5	mul	r6, r7, r0
mul	r6, r7, r3	add	r3, r4, r5
mul	r3, r8, r9	mul	r0, r8, r9

RISC 的特点

- 优点
  - 简化指令系统设计，适合 **VLSI** 实现
  - 提高执行速度和效率
  - 降低设计成本，提高了系统的可靠性
  - 可以提供直接支持高级语言的能力
- 缺点
  - 加重了汇编语言程序员的负担
  - 对浮点运算和虚拟存储器的支持不够理想
  - 相对来说， **RISC** 机器上的编译程序

综合实例 MIPS 指令集

- 概述
- MIPS 的寄存器
- MIPS 的数据表示
- MIPS 的寻址方式
- MIPS 指令格式
- MIPS 操作

概 述

MIPS 计算机是一个简单的 **64bit load-store** 系统结构的 **RISC** 计算机。 **MIPS** 强调：

- 简单的 **load-store** 指令集；
- 设计上重视流水线效率；
- 使得编译器更容易产生高效的目标代码。

MIPS 是一种适合学习和理解的系统结构模型

MIPS 的寄存器

- 整数寄存器

MIPS 设置了 **32** 个 **64bit** 的通用寄存器（ **GPR** ）用于整数操作，分别命名为： **R0** 、**R1** 、**…**、**R31** 。其中 R0 的值永远为 0 。

- 浮点寄存器

MIPS 设置了 **32** 个 **64bit** 的浮点数寄存器（ **FPR** ）用于浮点数数操作，分别命名为： **F0** 、**F1** 、**…**、**F31** 。

- 特殊寄存器

例如：浮点状态寄存器等。

MIPS 的数据表示

- 用于整数的数据类型

  有：字节（ 8bit ）、半字（ 16bit ）、字（ 32bit ）和双字（ 64bit ）。

- 用于浮点数的数据类型

  有：单精度（ 32bit ），双精度（ 64bit ）。

MIPS 的寻址方式

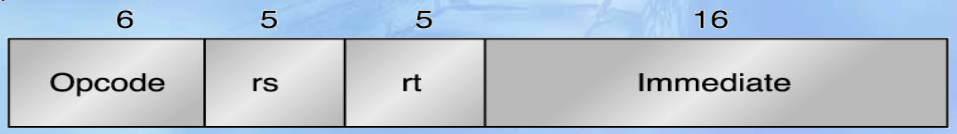
数据寻址方式只有两种：立即数和偏移寻址，字段长度都为 **16bit**。

寄存器间接寻址是通过将 **0** 放入 **16bit** 位移字段中得到的；**16bit** 绝对寻址是通过将寄存器 **R0** 作为基址寄存器得到的。

MIPS 指令格式

所有的指令都是 **32bit** ，其中 **6bit** 为操作码（包含寻址模式），具体格式见右图。

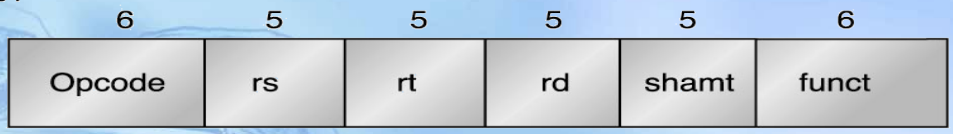
I-type instruction



Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

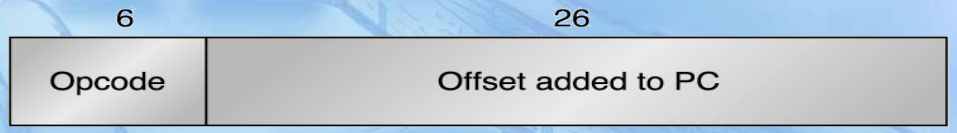
Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction



Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction



Jump and jump and link
Trap and return from exception

MIPS 操作

- **ALU** 操作
- 分支与跳转
- **load/store** 操作
- 浮点数操作

ALU 操作举例

Example instruction	Instruction name	Meaning
DADDU R1,R2,R3	Add unsigned	Regs[R1]←Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	Add immediate unsigned	Regs[R1]←Regs[R2]+3
LUI R1,#42	Load upper immediate	Regs[R1]←0^{32}##42##0^{16}
SLL R1,R2,#5	Shift left logical	Regs[R1]←Regs[R2]<<5
SLT R1,R2,R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1]←1 else Regs[R1]←0

上标：表示对字段进行复制。

##　：连接两个字段。

<<　：逻辑左移

分支与跳转举例

Example instruction	Instruction name	Meaning
J name	Jump	$PC_{36..63} \leftarrow$ name
JAL name	Jump and link	Regs[R31] $\leftarrow$ PC+4; $PC_{36..63} \leftarrow$ name; ((PC+4)-2^{27}) $\leq$ name $<$ ((PC+4)+2^{27})
JALR R2	Jump and link register	Regs[R31] $\leftarrow$ PC+4; PC $\leftarrow$ Regs[R2]
JR R3	Jump register	PC $\leftarrow$ Regs[R3]
BEQZ R4,name	Branch equal zero	if (Regs[R4]==0) PC $\leftarrow$ name; ((PC+4)-2^{17}) $\leq$ name $<$ ((PC+4)+2^{17})
BNE R3,R4,name	Branch not equal zero	if (Regs[R3] != Regs[R4]) PC $\leftarrow$ name; ((PC+4)-2^{17}) $\leq$ name $<$ ((PC+4)+2^{17})
MOVZ R1,R2,R3	Conditional move if zero	if (Regs[R3]==0) Regs[R1] $\leftarrow$ Regs[R2]

下标：标识字段中特定的位。

Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[30+Regs[R2]]
LD R1,1000(R0)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[1000+0]
LW R1,60(R2)	Load word	Regs[R1] $\leftarrow_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	Load byte	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ## Mem[40+Regs[R3]]##Mem[41+Regs[R3]]
L.S F0,50(R3)	Load FP single	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	Load FP double	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	Store double word	Mem[500+Regs[R4]] $\leftarrow_{64}$ Regs[R3]
SW R3,500(R4)	Store word	Mem[500+Regs[R4]] $\leftarrow_{32}$ Regs[R3]
S.S F0,40(R3)	Store FP single	Mem[40+Regs[R3]] $\leftarrow_{32}$ Regs[F0]$_{0..31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]] $\leftarrow_{64}$ Regs[F0]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]] $\leftarrow_{16}$ Regs[R3]$_{48..63}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]] $\leftarrow_{8}$ Regs[R2]$_{56..63}$

load/store 操作举例

浮点数操作

浮点操作包括加、减、乘、除，后缀 S 表示单精度浮点数，后缀 D 表示双精度浮点数，例如：

ADD.D ，ADD.S ，SUB.D ，SUB.S ，MUL.D ，MUL.S ，DIV.D ，DIV.S 等。