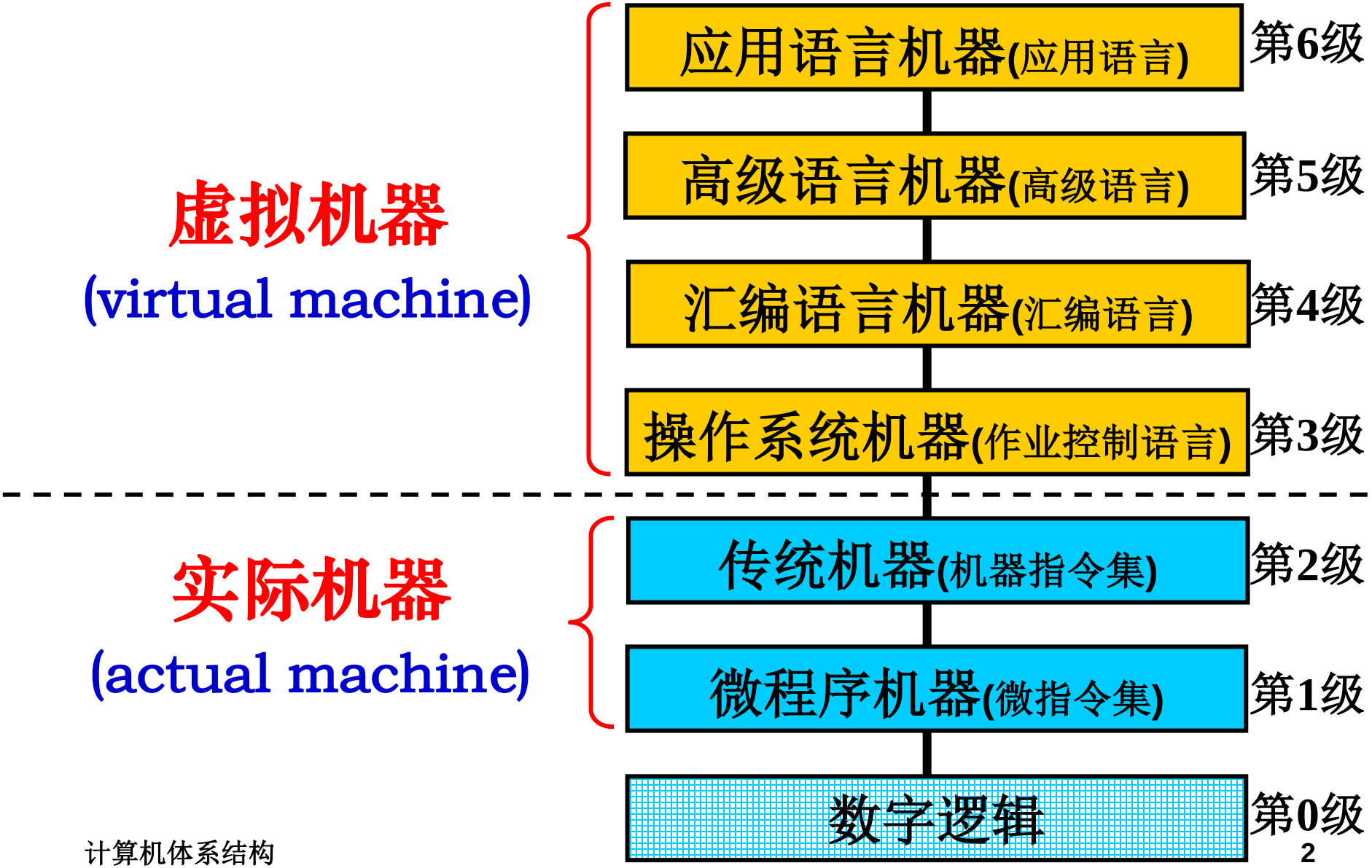
第2章
数据表示与指令系统

郑宏 副教授
计算机学院
北京理工大学

计算机系统的多级层次结构



计算机体系结构

- 在Amdahl提出的体系结构定义中：

  程序员指的是机器语言程序员或编译程序设计者，他们所看到的计算机属性是传统机器级的概念性结构和功能特性

Amdahl计算机体系结构指的是多级层次结构中传统机器级的体系结构

计算机体系结构≡Amdahl计算机体系结构

传统机器级的属性：功能特性

- **数据表示：** 硬件能够直接识别和处理的数据类型和格式；
- **寻址方式：** 最小寻址单位、寻址方式的种类和地址运算等；
- **指令集：** 机器指令的操作类型、格式，指令间的排序和控制机制等；
- **存储系统**；
- **输入输出系统**；
- ……

学习内容

- 2.1 数据表示
- 2.2 寻址方式
- 2.3 指令系统的设计和优化
- 2.4 指令系统的发展和改进
- 2.5 典型的RISC处理器
- 2.6 Intel嵌入式处理器

2.1 数据表示

- 数据：
  - 在计算机科学中，是指所有能输入到计算机中并被计算机程序处理的的符号的总称。可以是符号、数字、文字、图像、声音、代码等。
  - 数据本身没有意义，数据只有在对实体行为产生影响时才成为信息。数据是信息的表达，信息是数据的内涵。

2.1 数据表示

- 计算机中常用的数据典型有三类：
  - 用户定义的数据
  - 系统数据
  - 指令数据
- 数据有不同的数据类型。
- **数据类型：**

数据值的集合以及该集合的操作的集合。

2.1 数据表示

- **高级语言中的数据类型：两类**
  - **原子类型：例如实型、整型、布尔型、字符型等。**
  - **结构类型：例如数组、队列、链表、栈、向量等。**

2.1 数据表示

```
# define LIST_INIT_SIZE   100;
# define LIST_INC         10;
Typedef struct {
    ElemType        *elem;
    int             length;
    int             listsize;
}
```

```
class Student {
    private String IDNo;
    private String name;
    private String sex;
    private Date birthday;
}
```

2.1 数据表示

- 高级语言中的数据类型：两类
  - 原子类型：例如实型、整型、布尔型、字符型等。
  - 结构类型：例如数组、队列、链表、栈、向量等。
  - 常用十进制表示数值，Typedef 或 Class 定义或描述结构型数据类型。
  - 抽象数据类型：数学模型及其一组操作，与计算机内部如何表示和实现无关。

2.1 数据表示

- 传统机器中的数据类型：
  - 均采用二进制表示和存储。
  - 定点数：原码、反码、补码、移码。定点数使用补码进行运算。
  - 浮点数：尾数、基数和阶码。IEEE754 标准。
  - 字符与字符串表示方法：常用ASCII码。
  - 汉字：输入码，内码，子模码。
- 与高级语言中的数据表类型及表示是不同的。

2.1 数据表示

- 计算机系统结构研究的首要问题是：

  在所有的数据类型中，哪些用硬件实现，哪些用软件实现，并研究它们的实现方法。

- 数据表示
  - 可以被硬件直接识别和指令系统直接调用的数据类型。
  - 例如：定点数据表示。
- 思考：数据表示 = 数据的表示？

2.1.1 数据表示与数据结构

- **数据结构**
  - 结构化数据的组织方式，反映了应用中各种数据元素或信息单元之间的结构关系。
  - 例如：串、队列、堆栈、链表、树、向量、数组等。
  - 必须通过软件映像，变换成机器中所具有的各种数据表示来实现。

2.1.1 数据表示与数据结构

- **数据结构（续）**
  - 研究的是面向系统软件、面向应用领域所需要处理的各种数据类型；
  - 研究的是这些数据类型的逻辑结构和物理结构之间的关系，并给出相应的算法；
  - **数据表示以外的数据类型，一般都是数据结构要研究的内容。**

2.1.1 数据表示与数据结构

- 数据表示与数据结构的关系
  - **数据结构**是通过软件映像成机器所具有的各种数据表示实现的；
  - **数据表示**是数据结构的组成元素；
  - 数据表示为数据结构提供不同程度的支持，反映在效率和方便程度的不同；

2.1.1 数据表示与数据结构

- 数据结构和数据表示都是数据类型的子集。
- 哪些数据类型用数据表示实现，哪些用数据结构实现，实质上是一个软、硬件取舍的问题。
- 目前，定点数、浮点数、逻辑数、十进制数、字符串等基本数据表示和变址操作是不可缺少的。但还需要为数据结构提供更进一步的支持，减少软件负担。

2.1.1 数据表示与数据结构

- 例如：实现矩阵加法：C＝A×B，其中A和B均为200×200的矩阵。分析向量数据表示的作用。

```
int a[200][200], b[200][200], c[200][200];
for(i=0; i<200; i++)  {
    for(k=0; k<200; k++)  {
        for(j=0; j<200; j++)   {
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
        }
    }
}
```

2.1.1 数据表示与数据结构

- 分析：如果在没有向量数据表示的计算机系统上实现，一般需要6条指令，其中有4条指令要循环4万次。因此，CPU与主存储器之间的通信量：
  - 取指令 2＋4×40,000 条，
  - 读或写数据 3×40,000 个，
  - 共要访问主存储器 7×40,000 次以上

2.1.1 数据表示与数据结构

- 分析：如果有向量数据表示，只需要一条指令：

向量乘	A向量参数	B向量参数	C向量参数

  - 减少访问主存（取指令）次数：4×40,000
  - 缩短程序执行时间一倍以上

- 可见，引入一些高级数据表示，有时比在指令系统中增设一条技巧性的指令的意义要大

2.1.2 高级数据表示

- 有以下几种高级数据表示：
  - 自定义数据表示
  - 向量数组数据表示
  - 堆栈数据表示

1. 自定义数据表示

- 由数据本身来表明数据类型，使计算机内的数据具有自定义能力。
- 目的：缩短机器语言与高级语言对数据属性的说明之间的语义差距。
- 分为两类：
  - 带标志符的数据表示；
  - 数据描述符；

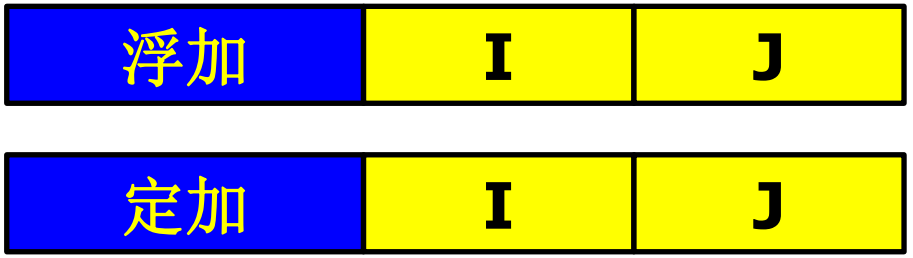
(1). 带标志符的数据表示

- 从处理运算符和数据类型的关系上看，高级语言和机器语言的差距很大。
- 高级语言：用数据类型说明语句说明数据，运算符通用。
  - 例如：在FORTRAN中，浮点数相加

```
REAL  I, J
I=I+J          （‘+’通用）
```

(1). 带标志符的数据表示

- **机器语言：操作码指明操作数**
  - 同一种操作指令通常需要很多条，IBM370系列机，仅加法指令就有8条。
  - 例如：
    - 浮点加法指令
    - 定点加法指令
    - ……



(1). 带标志符的数据表示

- **编译程序：**将高级语言中的数据类型说明语句和运算符变换成机器语言中的不同类型指令的操作码，并验证操作数与运算符是否一致。
- **编译程序负担很重。**

(1). 带标志符的数据表示

- 为缩短这种语义差距，在机器中设置带标志符的数据表示，让每个数据都带有类型标志位：

指明数据值部分的类型，如为二进制整数，还是十进制整数，浮点数，字符串或地址字。

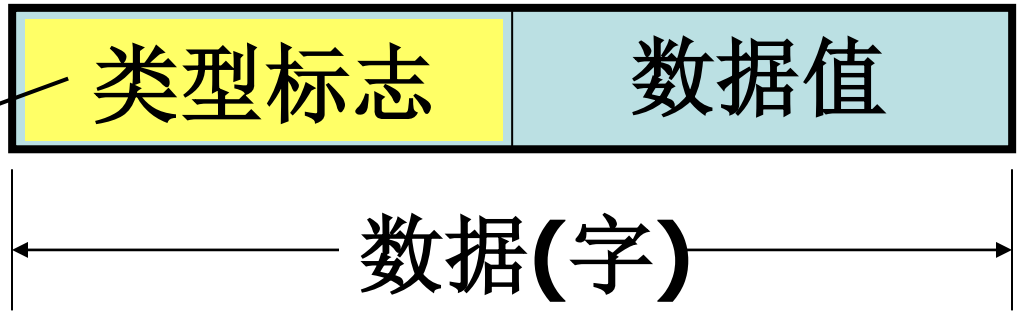


图 带标志符的数据表示

(1). 带标志符的数据表示

- 目的：

  将数据类型与数据本身直接联系在一起，使机器语言中的操作码与高级语言中的运算符一致。

- 标志符由编译器或其它系统软件建立，对程序员透明。

(1). 带标志符的数据表示

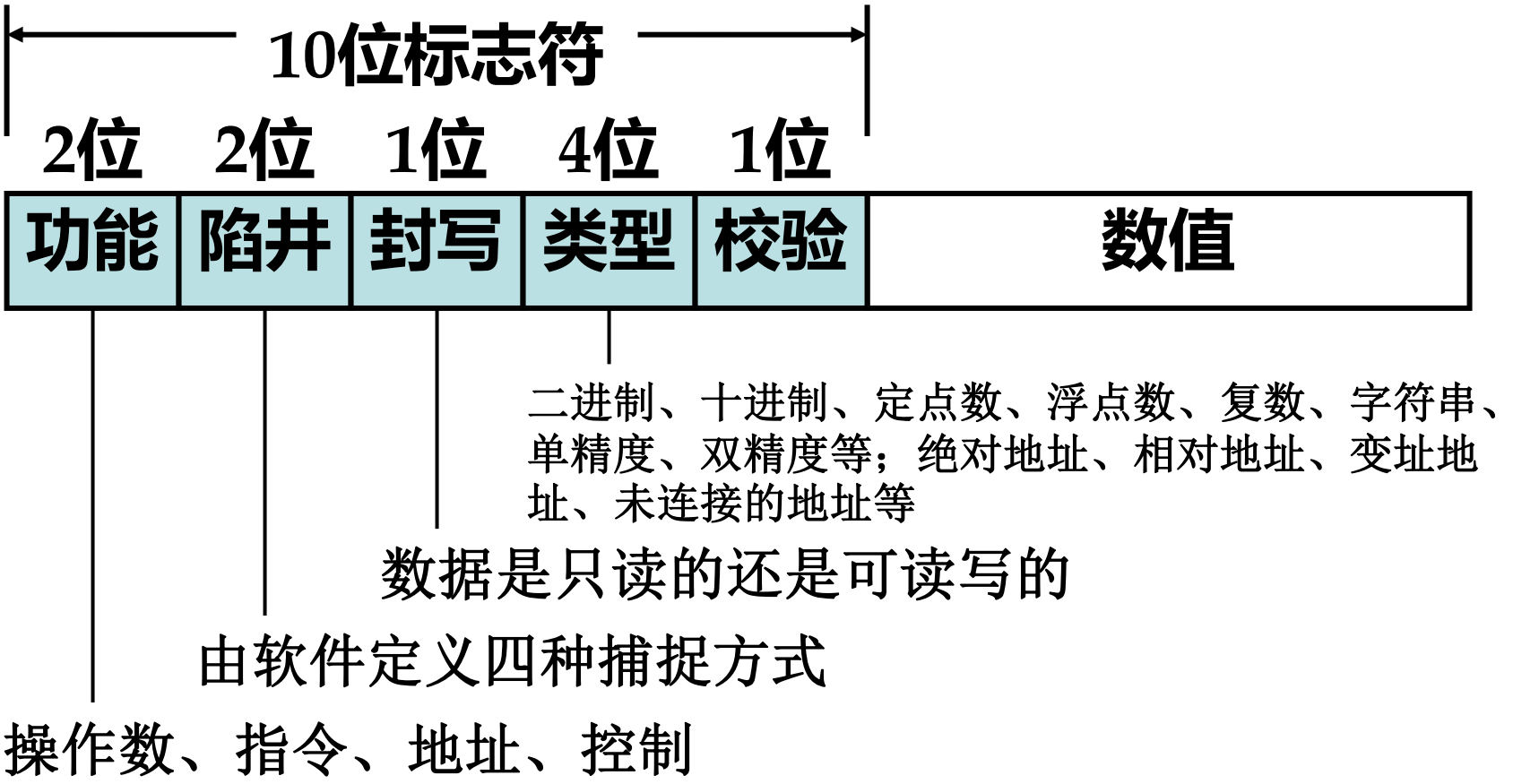


图 在R2巨型机中带标志符的数据表示方法

(1). 带标志符的数据表示

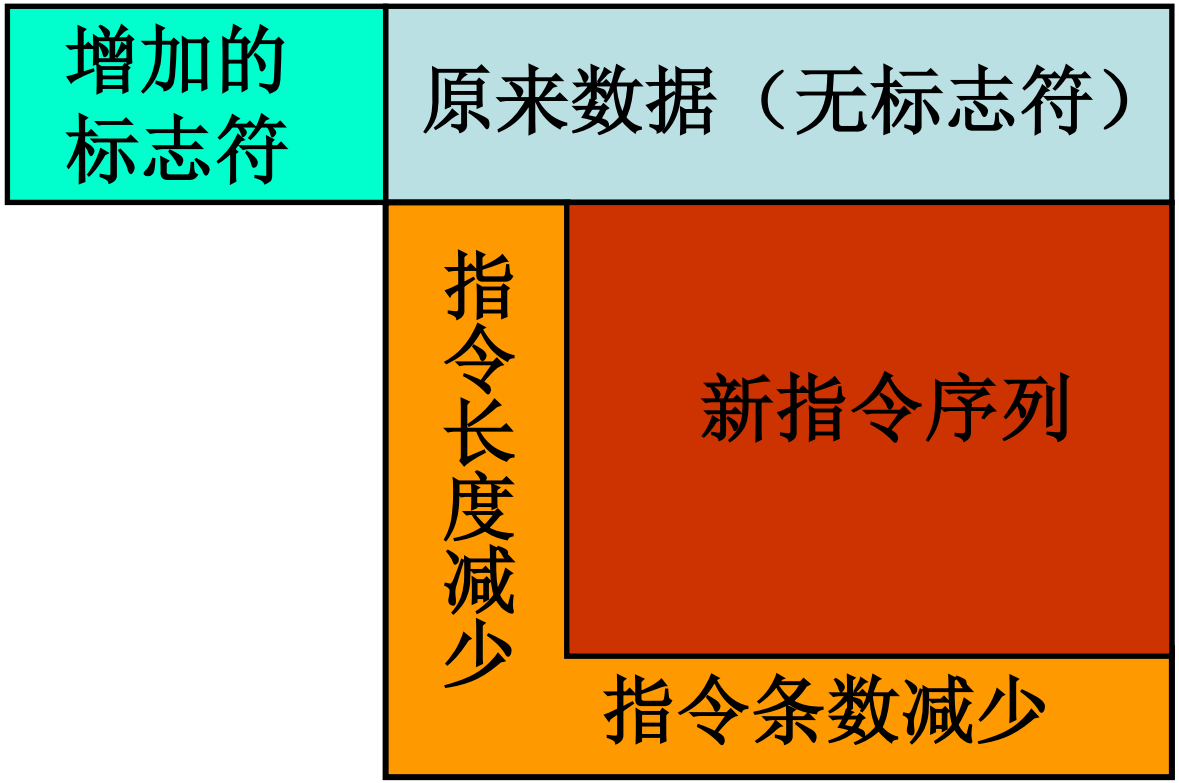
- 优点：

1. 提高了操作码的通用性，简化了指令系统和程序设计；
2. 简化的编译程序；
3. 便于实现一致性检查，可由硬件直接快速地检测出多种程序错误；
4. 能够由硬件自动完成数据类型转换；
5. 支持了数据库系统的实现与数据类型无关的要求；
6. 为软件调试和开发提供了支持。

(1). 带标志符的数据表示

- 缺点：
  1. 可能会使程序占用的空间增加。需要合理设计和使用；
  2. 单条指令的执行速度可能会降低。但编制、调试的时间缩短，总时间缩短。
- 优点明显，得到了广泛的应用。

(1). 带标志符的数据表示



当 面积大于 面积时，程序所占空间减少

(2). 数据描述符

- **目的：**支持向量、数组、记录等数据，减少标志符所占用的空间。
  - 因为这些数据的数据元素的属性都相同。

- 描述数据块的属性，与数据分开存放。

(2). 数据描述符

- 与带标志符的数据表示的区别：
  - 标志符要与每个数据相连，两者合存在一个存储器单元中；而描述符则和数据分开放；
  - 要访问数据集中的元素时，必须先访问描述符，这就至少要增加一级寻址；
  - 描述符可看成是程序一部分，而不是数据一部分，因为它是专门来描述要访问的数据的特性。

(2). 数据描述符

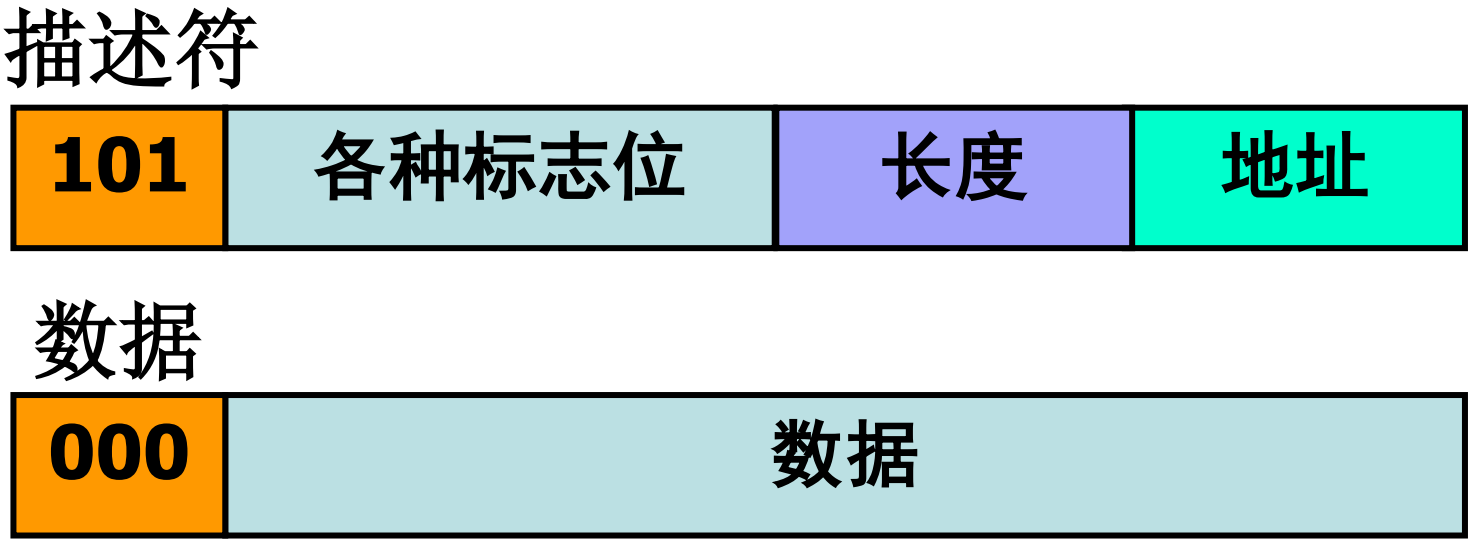


图 **B6700** 数据描述符表示方式

(2). 数据描述符

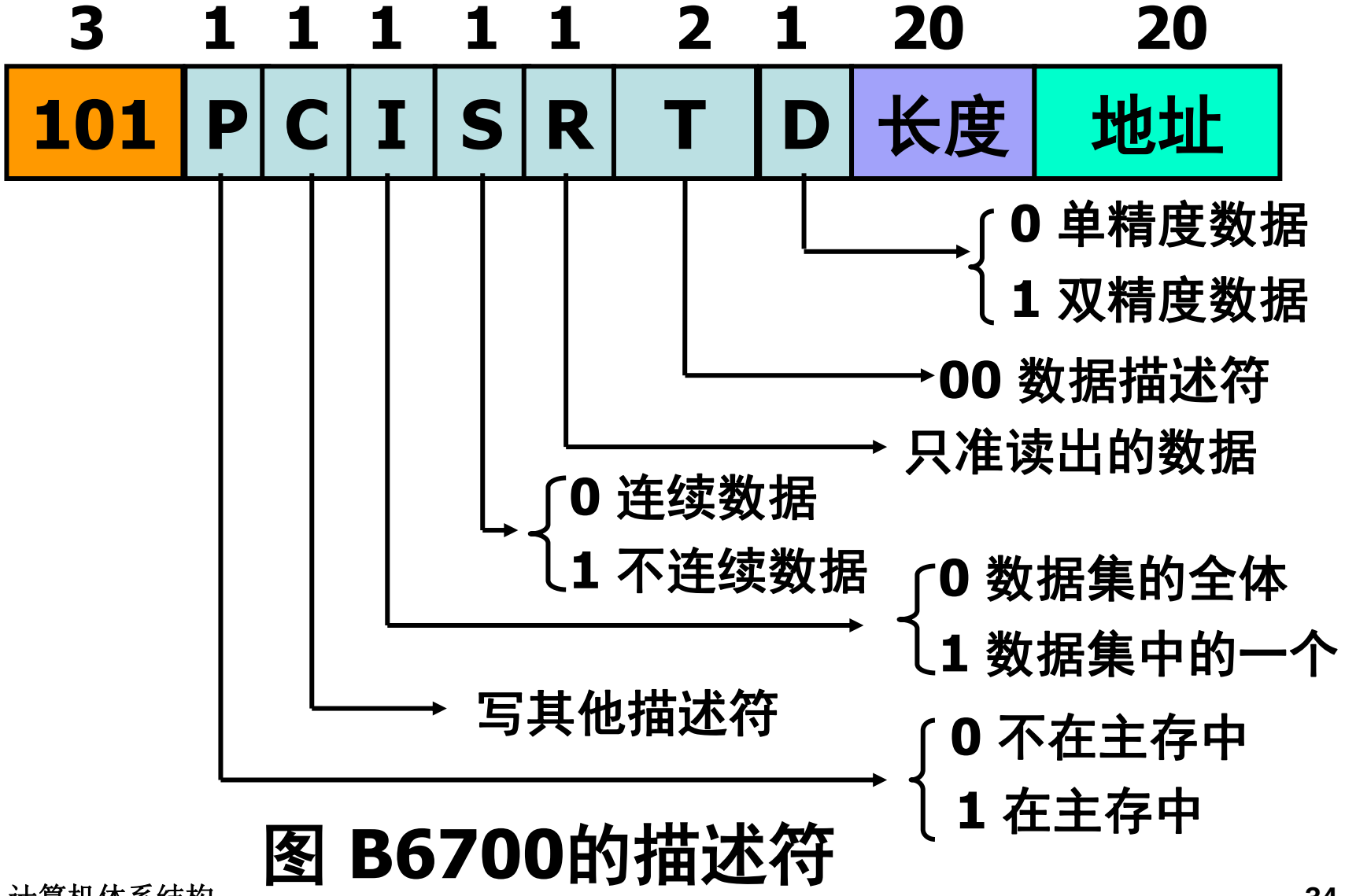


图 B6700的描述符

(2). 数据描述符

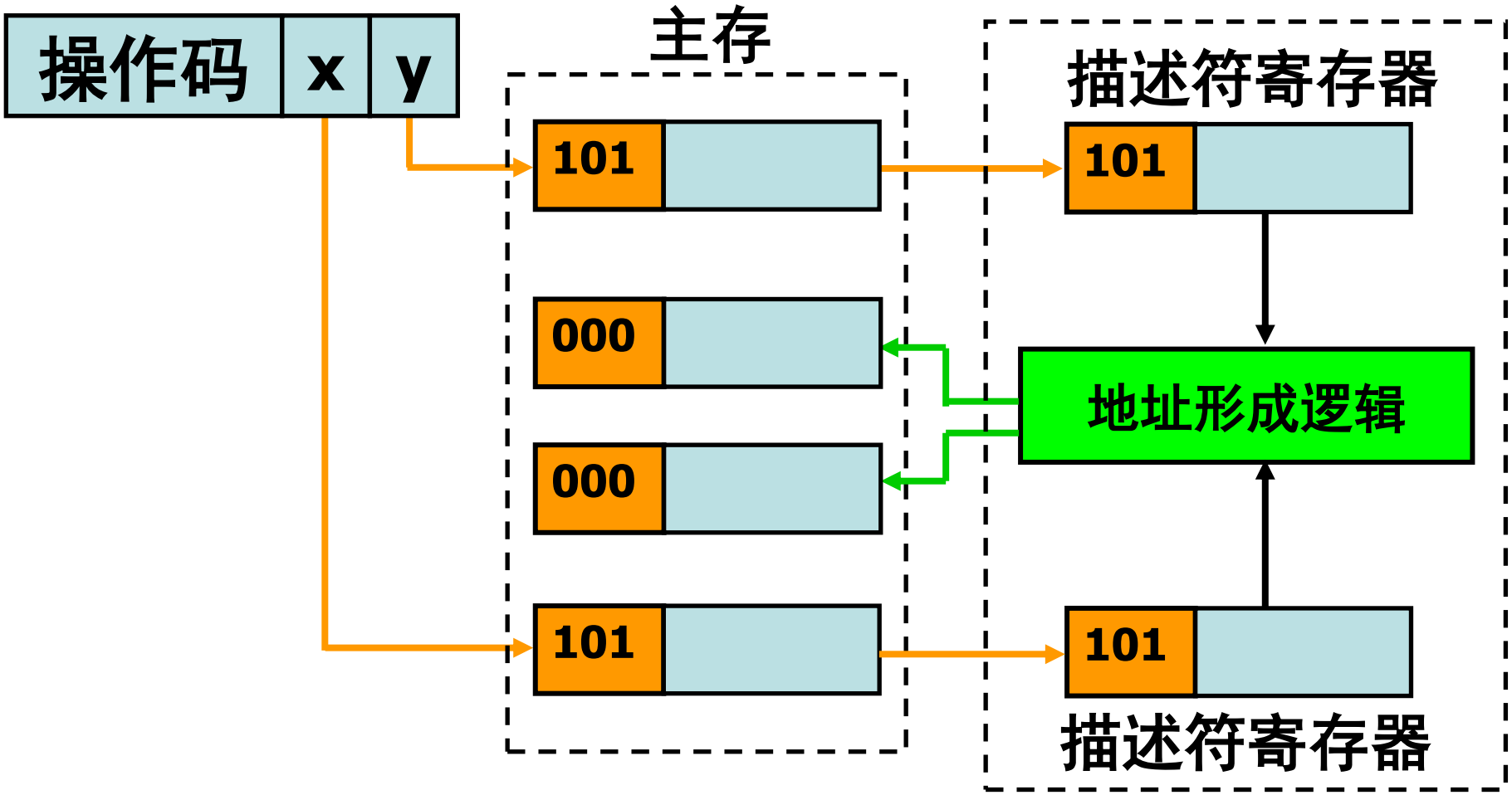
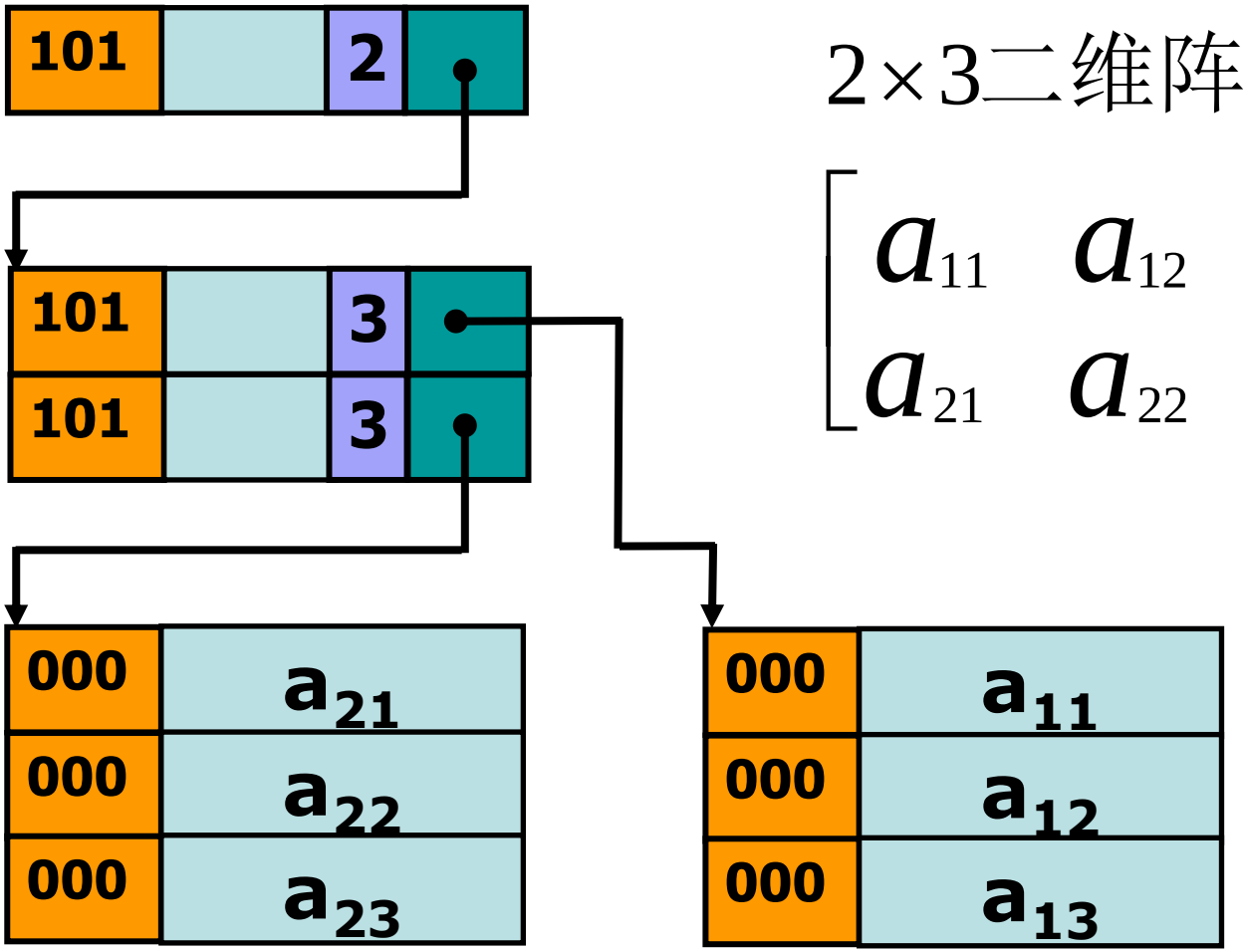


图 经描述符访问存储器，取得操作数

(2). 数据描述符



2×3二维阵列

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{a3} \end{bmatrix}$$

图 将描述符按树形连接可以描述多维阵列（数组）

(2). 数据描述符

- 优点
  - 将描述符按树形连接，可以描述复杂和多维数据结构；
  - 为向量、数组等数据结构的实现提供了一定的支持，有利于简化编译，可以比变址法更快地形成元素地址。

2. 向量数组数据表示

- 向量：指具有 n 个数据的数组。
- 特点：
  - 各个数据称为数组的元素；
  - 每个数据具有相同的数据类型，(如实数或逻辑数)；相同的数据表示（如字长、字的格式相同）；进行相同的操作；
  - 各数据之间是独立无关的。

2. 向量数组数据表示

- 20世纪70年代开始。
- 在需要对大量为向量、数组等数据进行高速运算的计算机中设置向量、数组数据表示，组成向量处理机。
- 在向量处理机上，在硬件上设置有丰富的向量或阵列运算指令，配置有以流水或阵列方式处理的高速运算器。

2. 向量数组数据表示

- 例如：要计算

$$c_i = a_{i+5} + b_i \qquad i = 10, 11, \ldots, 1000$$

- 用FORTRAN语言写成的有关DO循环部分为

```
      DO  40  I = 10, 1000
40    C(I) = A(I+5) + B(I)
```

2. 向量数组数据表示

- 例如（续）：在具有向量、数组数据表示的向量处理机上，表现出在硬件上设置有丰富的向量或阵列运算指令，配置有以流水或阵列方式处理的高速运算器，只需用一条如下的**向量加法**指令：

向量加	A向量参数	B向量参数	C向量参数

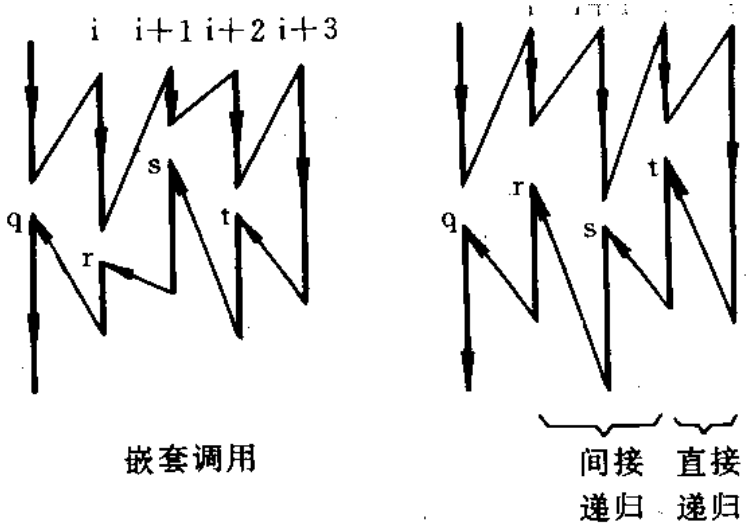
2. 向量数组数据表示

- 优点：提高了向量、数组的运算速度。
  - 快速形成元素地址；
  - 实现数据块的预取；
  - 用一条向量、数组指令，借助流水和并行运算等处理方式，可以同时实现对整个向量、数组的高速处理；
  - 硬件处理稀疏向量和交叉阵列，节省存储空间，节省处理时间；

3. 堆栈数据表示

- 堆栈数据结构在编译和子程序调用中经常用到。
- 为了能高效实现堆栈数据结构，不少大型机乃至微型机都设有堆栈数据表示。
- 具有堆栈数据表示的机器称为堆栈机器。

3. 堆栈数据表示



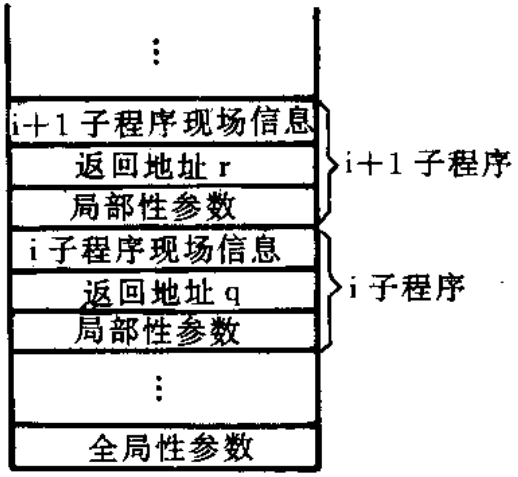


图 用堆栈实现子程序的嵌套和递归调用

3. 堆栈数据表示

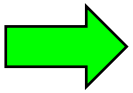
- 通用寄存器型机器对堆栈数据结构的支持很差，表现为：
  - 用于堆栈操作的机器指令少，功能单一；
  - 堆栈置于存储器内，访问堆栈的速度慢；
  - 通常只用来保存子程序调用时的返回地址，少量用堆栈来实现程序之间的参数传递。

3. 堆栈数据表示

- **堆栈机器则不同，表现为：**
  - 有由若干高速寄存器组成的**硬件堆栈，**通过附加控制电路，让它与主存中的堆栈区在逻辑上组成一个整体，使堆栈的**访问速度**是寄存器的，但**容量**是主存的；
  - 有丰富的堆栈操作类指令，功能很强，可以直接对堆栈中的数据进行处理；
  - 有力地支持高级语言的编译，简化了编译，缩小了高级语言与机器语言的语义差距；

算术赋值语句

F=A*B+C/(D-E)



逆波兰表达式

AB*CDE-/+

作为编译时的中间语言，可以直接生成堆栈机器指令程序。

3. 堆栈数据表示

- 堆栈机器则不同，表现为（续）：
  - 有力地支持子程序的嵌套调用和递归调用；
  - 堆栈机器的存储效率高，因为：
    - 及时释放不用单元；
    - 在访问堆栈时较多地使用零地址指令；
    - 在访问主存时一般采用相对寻址，使访存的地址码位数减少，从而使堆栈机器上程序的总位数，以及执行所要用到的存储单元数减少。

2.1.3 引入数据表示的原则

- 主要原则：
  - 缩短程序的运行时间；
  - 减少CPU与主存储器之间的通信量；
  - 这种数据表示的通用性和利用率是否很高。
    - 通用性：例如是否能高效支持多种数据结构，所花费的硬件代价是否带来性能上的提高。
    - 利用率：例如硬件设备量是否过多等。

2.1.3 引入数据表示的原则

- 从70年代开始，经过近10年的实践和探索，目前除了基本数据表示外，已根据使用环境分别引入了较复杂的堆栈数据表示、自定义数据表示和向量数据表示。
- 数据表示在不断发展，如：矩阵、树、图、表及自定义数据表示等。
- 现有的基本数据表示也还存在着一些问题。关于这个问题的讨论，请同学们阅读相关资料。

2.2 寻址方式

- **寻址技术：寻找操作数及其他信息的地址的技术。**
- **它是软件与硬件的一个主要分界面，是计算机系统结构的重要组成部分。**

2.2 寻址方式

- **研究内容：** 编址方式、寻址方式和定位方式。
- **研究对象：** 寄存器、主存储器、堆栈和输入输出设备。
- **在组成原理和汇编语言中，** 已经学习了常见的寻址方式的基本原理和寻址技术的实现方法。

体系结构研究的重点：分析各种寻址技术的优缺点，选择和确定寻址技术。

2.2.1 寻址方式分析

- **编址方式：** 指对各种存储设备进行编码的方法。
- 能被指令访问到的**存储部件**通常有：
  - 主存
  - 通用寄存器
  - 堆栈
  - 控制寄存器
  - 设备寄存器（I/O设备）
  - 某些专门寄存器等

1. 编址单位

- 常用的编址单位：
  - 字编址
  - 字节编址
  - 位编址
  - 块编址等
- 编址单位 = 访问单位
  - 每个编址单位所包含的二进制位数与读/写一次寄存器、主存的位数是相同。

1. 编址单位

- 字编址
  - 早期的大多数机器都采用这种编址方式。
    - 编址单位 = 访问单位
      - 每个编址单位所包含的二进制位数与读/写一次寄存器、主存的位数是相同。
  - 最简单。
  - 为进行字节或位操作，需要设置字节操作指令、位操作指令，在指令中指出操作数的字节编号或位编号。

1. 编址单位

- 字节编址
  - 是为了适应非数值计算的需要；
  - 字节编址方式使编址单位与信息的基本单位（一个字节）相一致；
  - 通常主存的访问单位是编址单位的若干倍（否则主存频带就太窄了）；
  - 编址单位 < 访问单位。

1. 编址单位

- 编址单位与访问单位（字长）
  - 一般：字节编址，字访问；
  - 部分机器：位编址，字访问。
  - 辅助存储器：块编址。

2. 编址方式

- 主要有三种：
  - 分类编址
  - 统一编址
  - 隐含编址

2. 编址方式

- 分类编址
  - 将部件适当分类，每类各自从“0”开始单独编址。
  - 通用寄存器、主存储器和输入输出设备均独立编址，形成三个零地址空间。
  - 通用寄存器独立编址，主存储器与输入输出设备统一编址，形成两个零地址空间。

2. 编址方式

- 分类编址
  - 优点：
    - 指令字长较短
    - 地址形成简单
    - 主存的编址空间较大
  - 缺点：
    - 指令中应有区分每类部件的标志或约定

2. 编址方式

- 统一编址
  - 把各种部件统一编成一个从“0”开始的一维线性地址空间（一个零地址空间）。
  - 对不同部件的访问反映在对这个空间不同地址范围的访问。

2. 编址方式

- 统一编址
  - 优点：
    - 可简化指令系统。
      - 例如若将I/O设备与主存统一编址，就不必设置单独的I/O指令。
  - 缺点：
    - 在一定程度上使地址形成复杂化。

2. 编址方式

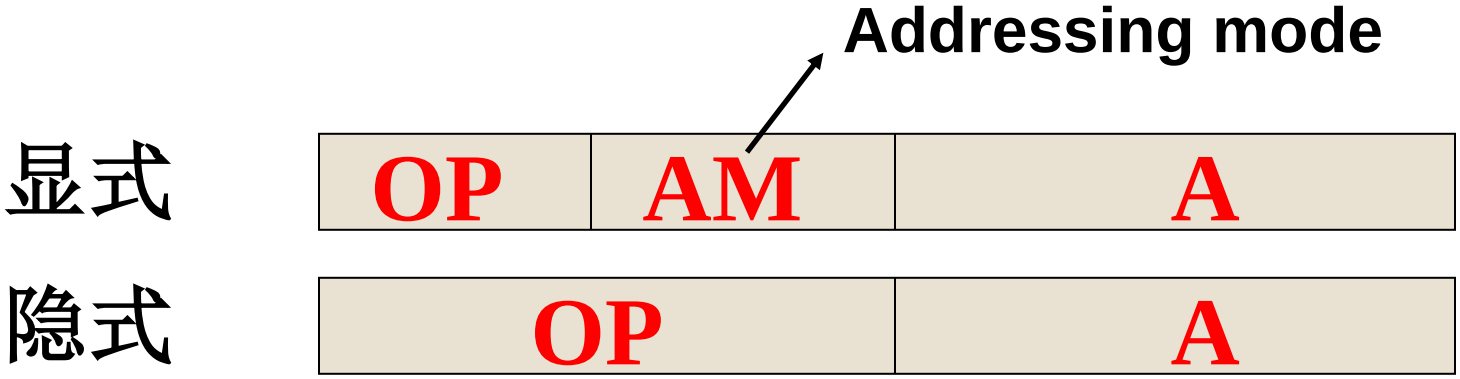
- 隐含编址
  - 采用事先约定的编址方式隐含寻址。
  - 例如堆栈、某些专用寄存器、Cache等可以采用这种编址方式。
  - 无零地址空间。
  - 优点：
    - 不必进行地址计算，访问速度比较快。
  - 缺点：
    - 有时会使指令的设计不规范。

2.2.1 寻址方式分析

- **寻址方式：**指令寻找（或访问）到所需要的数据的方法。
- **主要内容：**寻址方式的设计思想和设计方法。
- 寻址的多样性、灵活性、寻址空间范围大小、地址变换速度等都有了很大的发展。

2.2.1 寻址方式分析

- 寻址方式的指明：
  - **显式：** 在指令中设置专门的寻址方式字段，用二进制代码来表明寻址方式类型。
  - **隐式：** 由指令的操作码字段说明指令格式并隐含约定寻址方式。



2.2.1 寻址方式分析

- 大多数计算机采用分类编址方式。
- 通常将通用寄存器、主存和堆栈分类编址。
- 因此有三类寻址方式：
  - 面向寄存器的寻址方式
  - 面向主存的寻址方式
  - 面向堆栈的寻址方式

2.2.1 寻址方式分析

- 面向寄存器的寻址方式
  - 指令形式：

    opcode　reg

    opcode　reg1，reg2

    opcode　reg1，reg2，reg3

    opcode　reg，mem

2.2.1 寻址方式分析

- 面向寄存器的寻址方式
  - 操作数可以取自寄存器或主存。
  - 结果大多数送寄存器，少数送主存。
  - **优点：** 指令字长短；执行速度快；支持向量、矩阵等运算。
  - **缺点：** 不利于优化编译；现场切换困难；硬件复杂。

2.2.1 寻址方式分析

- 面向主存的寻址方式
  - 指令形式：

    opcode　mem

    opcode　mem1，mem2

    opcode　mem1，mem2，mem3

    opcode　reg，mem
  - 主要访问主存，少量访问寄存器。

2.2.1 寻址方式分析

- 面向堆栈的寻址方式
  - 指令形式：

    opcode

    opcode　mem

    opcode　reg
  - 主要访问堆栈，少量访问主存或寄存器。

2.2.1 寻址方式分析

- 面向堆栈的寻址方式
  - **主要优点：** 支持高级语言，有利于编译程序；节省存储空间；支持程序的嵌套和递归调用，支持中断处理。
  - **主要缺点：** 运算速度比较低。

2.2.1 寻址方式分析

- 哪一类寻址方式最好？
  - 很难说。
  - 对不同的程序和不同的工作阶段会得到不同的效果。

2.2.1 寻址方式分析

- 由于：
  - 用户程序的多样性；
  - 以及高级语言程序从编译到运行各阶段的工作特点不同；
- 所以三类寻址方式不应当互相排斥
- 在同一系统结构中都应当采用，并以某一类寻址方式为主，其它方式为辅。

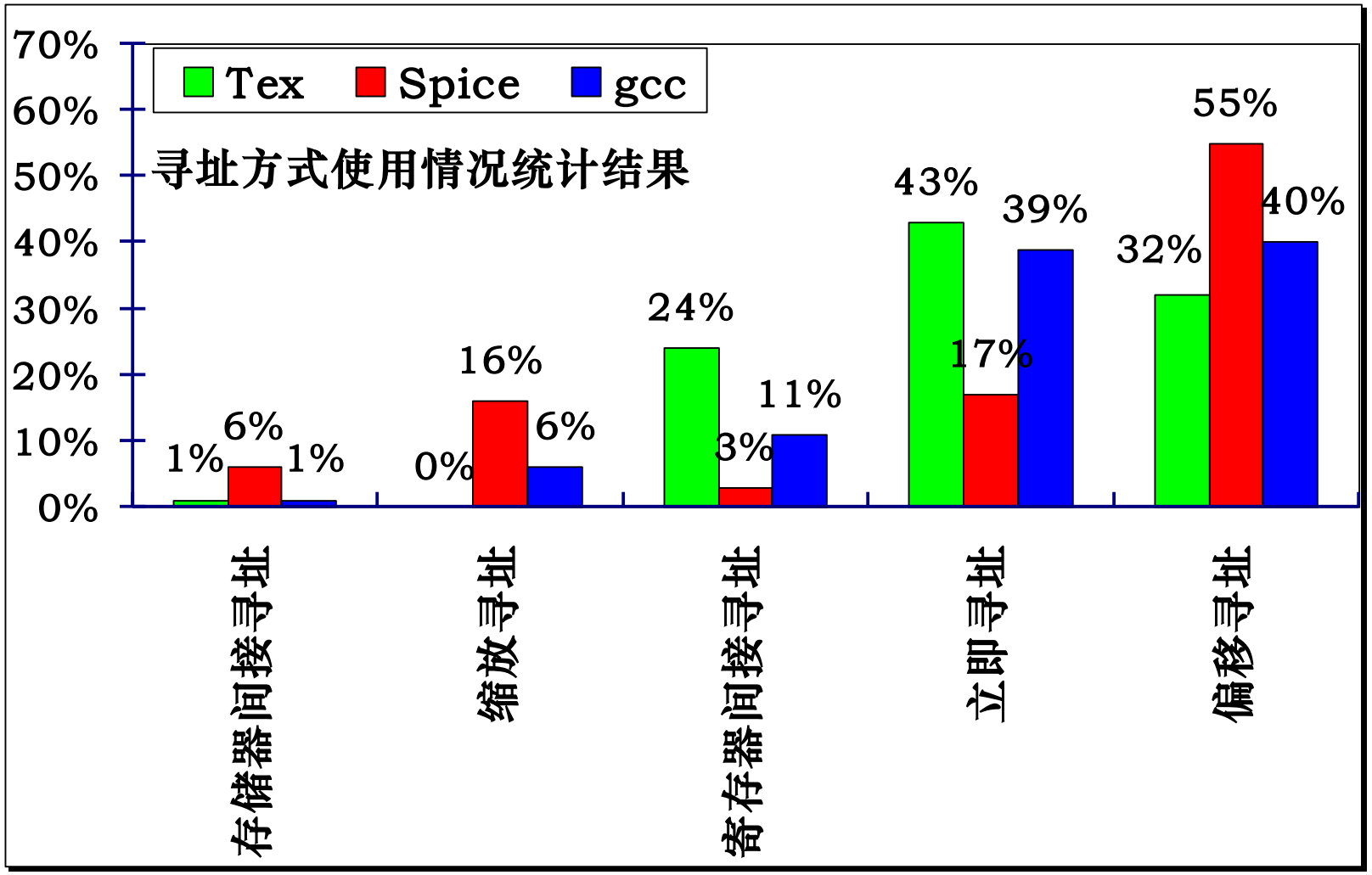
2.2.1 寻址方式分析

- 具体的寻址方式：很多，如：
  - 立即寻址
  - 直接寻址
  - 间接寻址
  - 相对寻址
  - 变址寻址等

2.2.1 寻址方式分析

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]]
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]

常用的一些操作数寻址方式



2.2.1 寻址方式分析

- 如何在指令中指明采用哪种寻址方式？
- 通常有两种方式：
  - 在操作码中指明（隐式）
    - 由操作码中的某些位来指明.
  - 在地址码中指明（显式）
    - 专门设置寻址方式位指明。
    - 优点：寻址灵活，操作码短。
    - 缺点：需要设置专门的寻址方式位，指令长度相对变长。

2.2.2 间接寻址方式与变址寻址方式的比较

- 目的相同：
  - 都是为了解决操作数地址的修改问题；
  - 都能做到不改变程序而修改操作数地址。
- 原则上，一种处理机中只需设置间接址寻址方式与变址寻址方式中的任何一种即可，有些处理机两种寻址方式都设置。
- 如何选取间址寻址方式与变址寻址方式？

2.2.2 间接寻址方式与变址寻址方式的比较

- 例：一个由N个元素组成的数组，已经存放在起始地址为AS的主存连续单元中，现要把它搬到起始地址为AD的主存连续单元中。不必考虑可能出现的存储单元的重叠问题。为了编程简单，采用一般的两地址指令编写程序。

2.2.2 间接寻址方式与变址寻址方式的比较

解：采用间接寻址方式编写程序如下：

```
start: move asr,asi        ;保存源数组的起始地址
       move adr,adi        ;保存目标数组起始地址
       move num, cnt       ;保存数据的个数
loop:  move @asi,@adi      ;用间址寻址方式传送数据
       inc     asi         ;源数组的地址增量
       inc     adi         ;目标数组的地址增量
       dec     cnt         ;个数减1
       bgt     loop        ;测试n个数据是否传送完
       halt                ;停机
asr:           as          ;源数组的起始地址
adr:           ad          ;目标数组的起始地址
num:           n           ;需要传送的数据个数
asi:           0           ;当前正在传送的源数组地址
adi:           0           ;当前正在传送的目的数组地址
cnt:           0           ;剩余数据的个数
```

2.2.2 间接寻址方式与变址寻址方式的比较

解：采用变址寻址方式编写程序如下：

```
start: move as, x          ;源数组起址送变址寄存器
       move num, cnt       ;保存数据个数，保证再入性
loop: move (x), ad-as(x)   ;ad-as位地址偏移量,
                           ;在汇编时计算
       inc    x            ;增量变址寄存器
       dec    cnt          ;个数减1
       bgt    loop         ;测试n个数据是否传送完成
       halt                ;停机
num: n                     ;需要传送的数据个数
 cnt:  0                   ;剩余数据的个数
```

2.2.2 间接寻址方式与变址寻址方式的比较

- 采用变址寻址方式编写的程序简单、易读。
- 对于程序员，两种寻址方式的主要差别是：
  - 间址寻址方式：间接地址在主存储器中，没有偏移量；
  - 变址寻址方式：基地址在变址寄存器中，有偏移量；
- 主要优缺点比较：
  - 实现的难易程度：间址寻址方式容易
  - 指令的执行速度：间址寻址方式慢
  - 对数组运算的支持：变址寻址方式比较好

需要注意的问题：见教科书。

2.2.3 程序在主存中的定位技术

- 何时确定程序在主存中的物理地址？
- 以何种方法确定程序在主存中的物理地址？
- 根据程序的独立性要求和模块化设计思想，为了解决程序空间与主存空间大小不同的矛盾、以及多道程序共享主存空间等问题，引入了新的“地址”概念。

2.2.3 程序在主存中的定位技术

- **逻辑地址**
  - 程序员编写程序时所使用的地址。
- **主存物理地址**
  - 程序在主存中的实际地址。

2.2.3 程序在主存中的定位技术

- 早期，程序和数据都存放在实际主存中，其位置由程序员在编程时确定，即没有逻辑地址与主存物理地址之分，或者说，逻辑地址与主存物理地址是一致的。
- 随着汇编语言、编译程序和操作系统的出现，源程序已经不使用指令、数据在主存中的实际地址编程，而是使用符号、标号等编址。
- 这些符号构成了名字空间，通过汇编或编译程序翻译成逻辑地址空间。

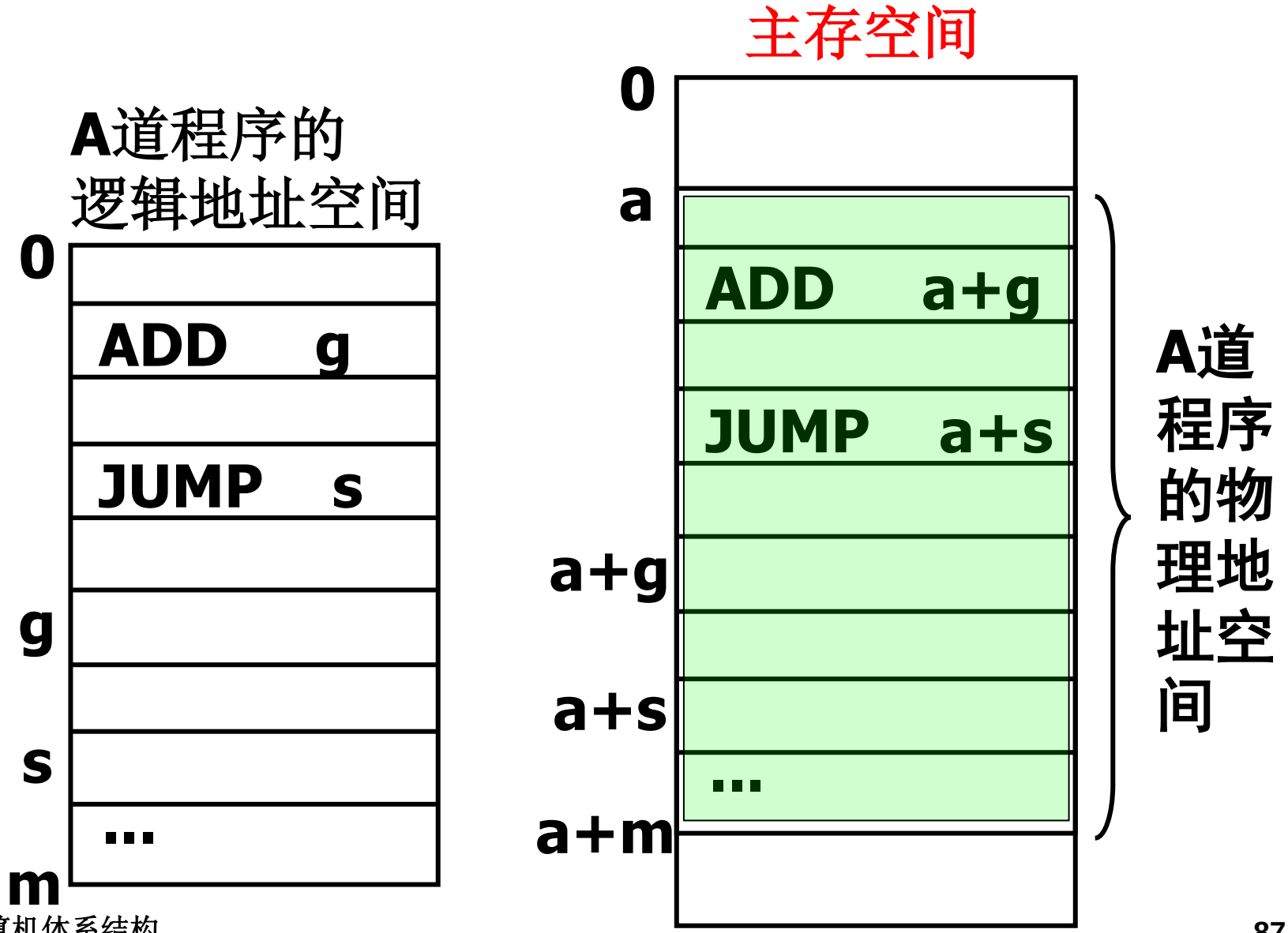
2.2.3 程序在主存中的定位技术

- 由于在主存中同时存放着多道程序，程序员事先无法知道其所编制的程序存放在主存的什么位置上，因此各道程序的逻辑地址只能都从零开始编址，而主存物理地址是从零开始的一维线性空间。
- 这样，逻辑地址空间和物理地址空间是不一致的。

2.2.3 程序在主存中的定位技术

- 因此，当把程序装入主存时，需要进行逻辑地址空间到物理地址空间的**变换**，即进行**程序的定位。**

2.2.3 程序在主存中的定位技术



2.2.3 程序在主存中的定位技术

- 显然，当程序转入主存后，为保证运行正确，指令中的地址码应根据不同的寻址方式做相应的变换。
- 主要有三种变换（定位）技术：
  - 直接定位
  - 静态再定位
  - 动态再定位

1. 直接定位

- 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。**

2. 静态再定位

- 20世纪60年代初
- 利用Von Neumann型机器指令可以修改的特点，在目标程序装入主存时，通过调用装入程序，用软件的方法把目的程序的逻辑地址变换成物理地址。
- 在程序执行过程中，物理地址不再改变。

2. 静态再定位

- 问题：
  - 一道程序的地址修改错误会导致其它程序被破坏；
  - 妨碍了程序可再（重）入性要求；
  - 故障的诊断、排错、调试等变得很困难；
  - 给重叠、流水等技术的采用带来困难。
- 所以60年代后提出程序中的指令不允许修改。

3. 动态再定位

- 20世纪60年代后期
- 在执行指令时，才形成访问主存的物理地址；

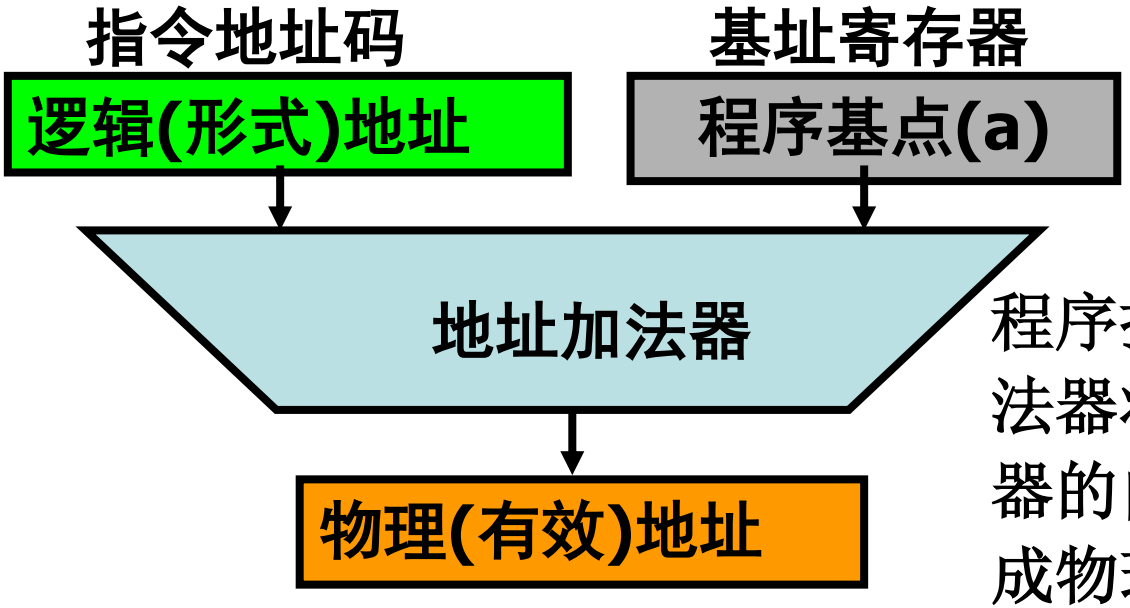


图　基址寻址工作原理

程序执行时，通过例如地址加法器将逻辑地址加上基址寄存器的内容(程序基点地址)，形成物理地址，然后访存，地址变换速度快。

3. 动态再定位

- 20世纪70年代初，出现了虚拟存贮器，增加了映像表硬件，使程序空间可以超过实际的主存空间，进一步改经和发展了动态再定位技术。

2.2.4 信息分布

- 寻址方式中，信息在物理地址空间中的分布是一个值得注意的问题。
- 通常，一台机器中可以同时存放宽度不同的多种信息。
- 如何存放这些信息，以使访问速度尽可能地快？➔ 减少访问主存的次数。

2.2.4 信息分布

- 物理地址空间通常采用按字节编址，即可以寻址到字节，各种宽度的信息按该信息首字节的地址来访问。
- 若主存宽度为64位（即8个字节），则在一个存储周期内可以访问8个字节。

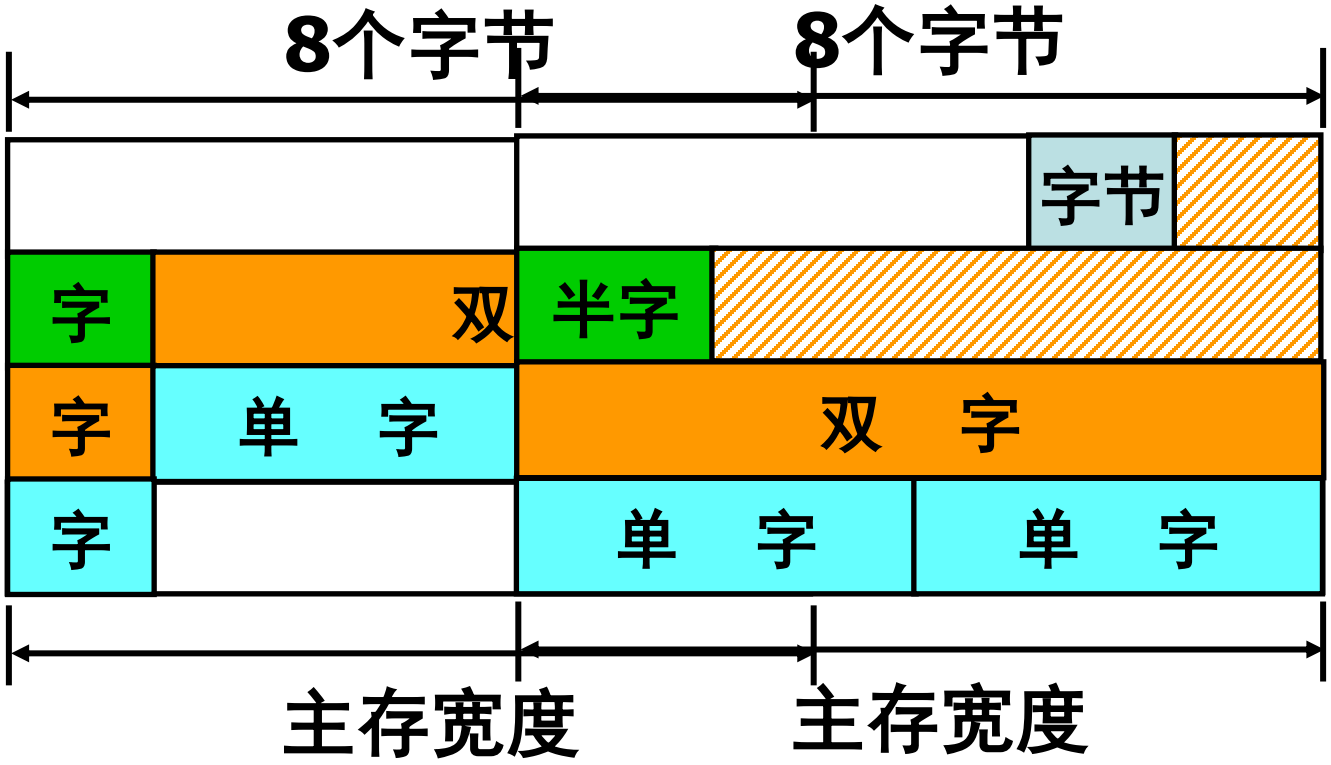
2.2.4 信息分布

- 信息的存储有两种方式：
  - 任意存储
  - 按字节数或位数的整数倍存储
- 以IBM 370为例：
  - 四种信息宽度：字节(8位)、半字(2字节)、单字(4字节)、双字(8字节)
  - 主存宽度：64位(8字节)

1. 任意存储方式

- 各种宽度的信息可以任意存储。
- 问题：可能会出现信息跨主存字边界存贮的情况。
  - 虽然信息宽度小于或等于主存宽度，但却需要花费两个存贮周期才能访问到，使访问速度显著降低。

2.2.4 信息分布

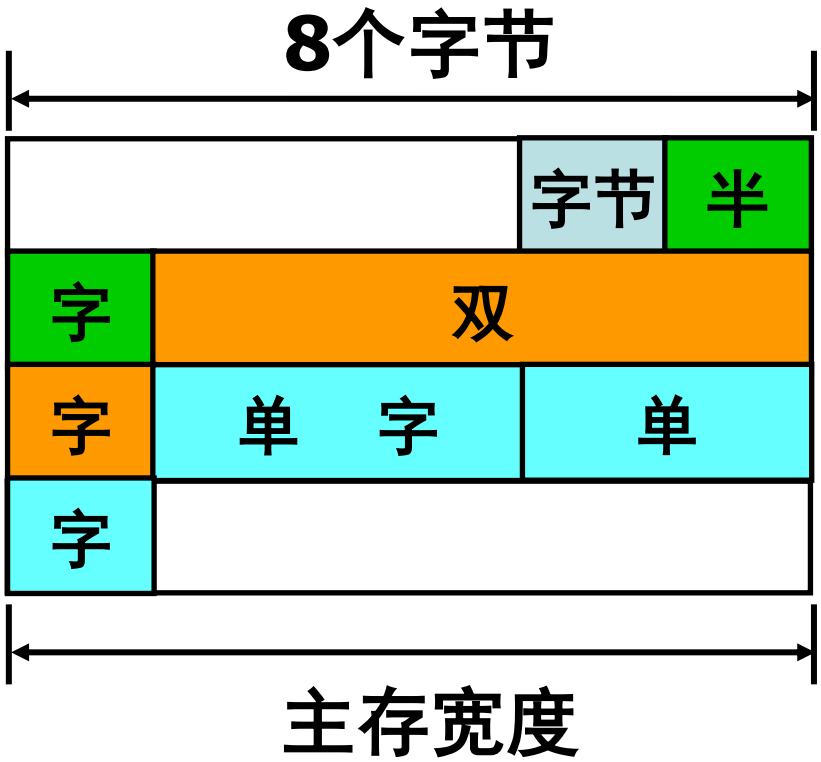


(a)任意存储方式　　(b)按字节数的整数倍存储方式

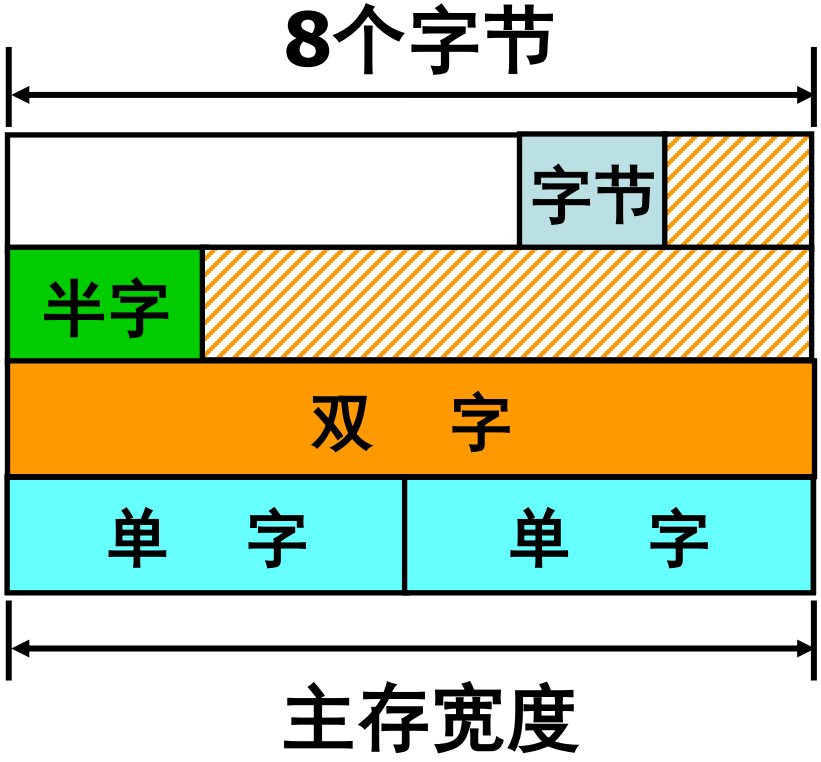
2. 按字节数的整数倍存储

- 信息在主存的存贮地址必须是字节数的整数倍。
  - 这就是信息在存贮器中按整数边界存贮的概念。
- 信息在主存的存贮地址必须是：
  - 字节信息：X…XXXX
  - 半字信息：X…XXX0
  - 单字信息：X…XX00
  - 双字信息：X…X000

2.2.4 信息分布



(a)任意存储方式



(b)按字节数的整数倍存储方式

2. 按字节数的整数倍存储

- **优点：** 访问速度快。任何信息都只需用一个存贮周期就可以访问到。
- **缺点：** 会造成存贮空间的某些浪费。

2.3 指令系统的设计和改进

- 指令系统是程序设计者看到的机器的主要属性，是软、硬件的主要界面，在很大程度上决定了机器的所具有的基本功能。

2.3 指令系统的设计和优化

- 设计内容：
  - 指令功能设计
  - 指令格式设计
- 设计原则：
  - 有利于优化机器的性能价格比
  - 有利于指令系统的发展和改进

2.3.1 指令操作码的优化

- 指令的组成：操作码 + 地址码(可能多个)

图　指令的组成

2.3.1 指令操作码的优化

- **操作码**主要包括两部分：
  - **操作种类：**加、减、乘、除、数据传送、移位、转移、输入输出等。
  - **操作数描述：**
    - **数据类型：**定点数、浮点数、复数、字符、字符串、逻辑数、向量；
    - **进位制：**2进制、10进制、16进制；
    - **数据字长：**字、半字、双字、字节。

2.3.1 指令操作码的优化

- **地址码**通常包括三部分：
  - **地址：**直接地址、间接地址、立即数、寄存器编号、变址寄存器编号等；
  - **地址附加信息：**偏移量、块长度、跳距；
  - **寻址方式：**直接寻址、间接寻址、立即数寻址、变址寻址、相对寻址、寄存器寻址等。

2.3.1 指令操作码的优化

- 指令操作码优化目的：

在足够表达全部指令的前提下，使操作码字段所占用的位数最少，从而缩短指令字长，减少程序所占存储空间。

2.3.1 指令操作码的优化

- 操作码的三种编码方法：
  - 固定长度编码
  - Huffman编码
  - 扩展编码

- 为进行评价，先介绍几个概念和公式。

2.3.1 指令操作码的优化

- **信息源熵（entropy）**
  - 信息源包含的平均信息量。
  - 对操作码而言就是操作码的**最短平均码长（理想情况）**。
  - 计算公式：

$$H = -\sum p_i \log_2 p_i$$

其中：H—信息源熵

p_i—第i个操作码出现的概率

2.3.1 指令操作码的优化

- **信息冗余量**

$$R = 1 - \frac{H}{\text{实际平均码长}}$$

- **实际平均码长**

$$L = \sum p_i l_i$$

其中：p_i —第i个操作码出现的概率

l_i —第i个操作码的长度

1. 固定长度编码

- 传统方法。
- 定长：所有指令的操作码长度相同。
- 优点：规整，便于译码。
- 缺点：信息冗余量大，指令长，程序大。

1. 固定长度编码

- 例：一台模型机，有七条不同的指令，各指令的使用频度如表所示：

指令	使用频度(P_i)
I_1	0.40
I_2	0.30
I_3	0.15
I_4	0.05
I_5	0.04
I_6	0.03
I_7	0.03

1. 固定长度编码

- 解：
  - 实际平均码长：3位二进制数
  - 信息源熵：

$$H = -\sum p_i \log_2 p_i = 2.17(\text{位})$$

  - 信息冗余量：

$$R = 1 - \frac{H}{\text{实际平均码长}} = 1 - \frac{2.17}{3} = 28\%$$

2. Huffman编码

- 基于哈夫曼压缩概念。 1952年由Huffman首先提出。
- 哈夫曼压缩概念
  - **基本思想：** 当各种事件发生的概率不均等时，使用概率高的事件用短代码表示，使用概率低的事件用长代码表示，就会使平均位数缩短。
  - **前提条件：** 必须事先知道事件发生的概率。

2. Huffman编码

- 又称为最小概率合并法。
- 编码方法：

  (1)把所有指令按照操作码在程序中出现的概率，自左向右从小到大排列好。

  (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。

2. Huffman编码

- 又称为最小概率合并法。
- 编码方法（续）：

  (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，分别用一位代码“0”和“1”表示。

  (6)从根结点开始，沿箭头所指方向，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。

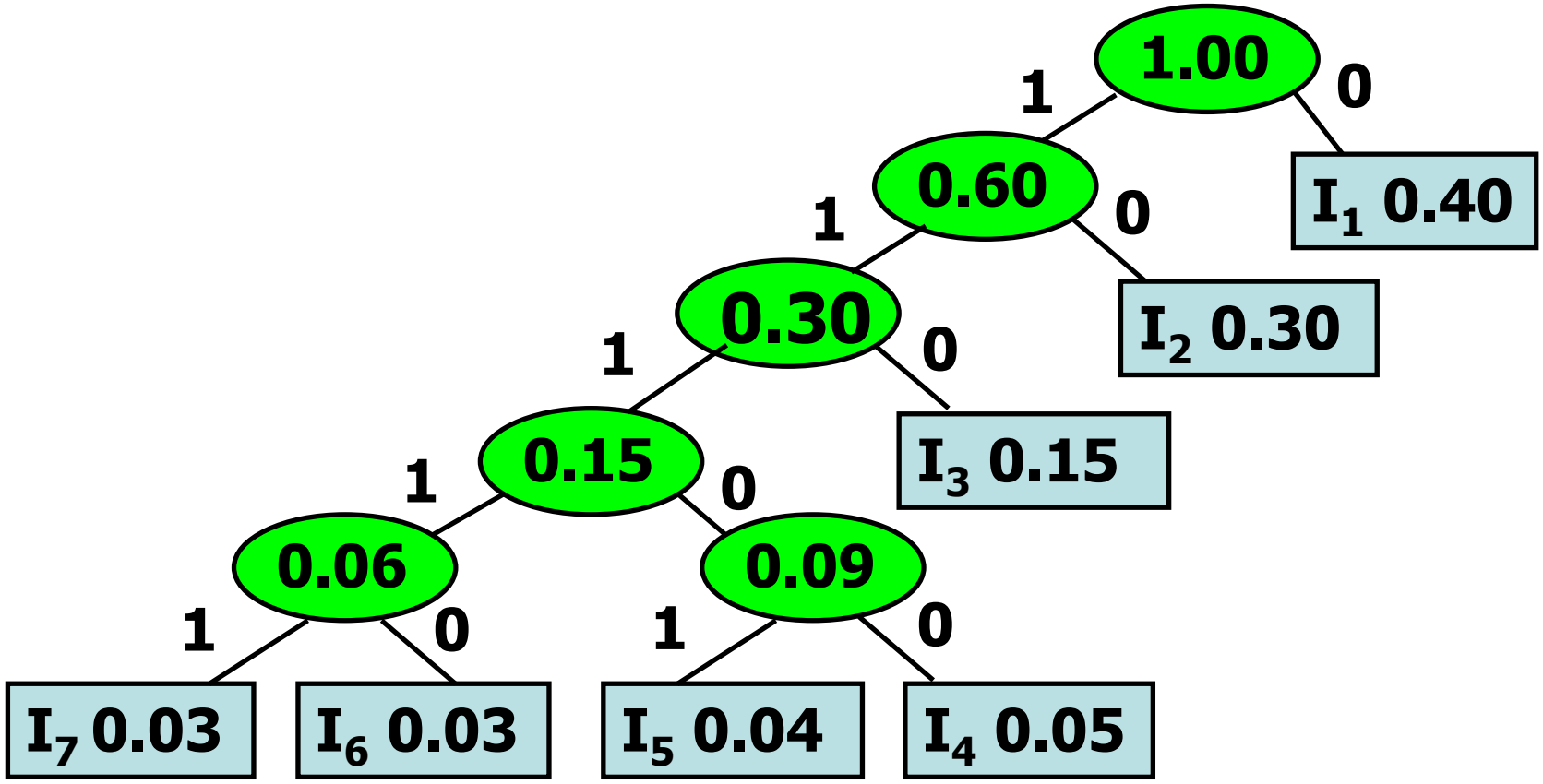
2. Huffman编码

- Huffman编码
  - 利用哈夫曼算法，构造哈夫曼树（二叉树），分别用“0”或“1”表示出两个分支，从而得到每条指令的哈夫曼编码。
- 举例

指令	使用频度(P_i)
I_1	0.40
I_2	0.30
I_3	0.15
I_4	0.05
I_5	0.04
I_6	0.03
I_7	0.03

2. Huffman编码

- 解：构造哈夫曼树



2. Huffman编码

- 解：每条指令的长度

指令	频度(P_i)	操作码使用的哈夫曼编码
I_1	0.40	0
I_2	0.30	1 0
I_3	0.15	1 1 0
I_4	0.05	1 1 1 0 0
I_5	0.04	1 1 1 0 1
I_6	0.03	1 1 1 1 0
I_7	0.03	1 1 1 1 1

2. Huffman编码

- 解：
  - 实际平均码长：

$$L=\sum p_i l_i = 2.20(\text{位})$$

  - 信息源熵：

$$H=-\sum p_i log_2 p_i = 2.17(\text{位})$$

  - 信息冗余量：

$$R=1-\frac{H}{\text{实际平均码长}}=1-\frac{2.17}{2.20}\approx 1.36\%$$

2. Huffman编码

- **特点：** 哈夫曼编码并不唯一，但平均码长唯一。
- **优点：** 实际平均码长最短。
- **主要缺点：**
  - 操作码长度很不规整，硬件译码困难。
  - 与地址码共同组成固定长的指令比较困难。

3. 扩展编码

扩展编码

- 一种介于定长编码和全哈夫曼编码之间的一种编码方法。
- 操作码长度不固定，但只有有限的几种长度。
- 使用频度高的指令用短操作码表示，使用频度低的指令用长操作码表示(哈夫曼压缩思想)。

3. 扩展编码

- 扩展编码举例：
  - 使用2－4 扩展编码（2位和4位码长）。
  - 使用频度高的指令I_1、I_2、I_3用2位编码表示。
  - 使用频度低的指令I_4、I_5、I_6、I_7用4位编码表示。

指令	使用频度(P_i)
I_1	0.40
I_2	0.30
I_3	0.15
I_4	0.05
I_5	0.04
I_6	0.03
I_7	0.03

3. 扩展编码

- 解：

指令	频度(P_i)	2—4 扩展操作码编码
I_1	0.40	0 0
I_2	0.30	0 1
I_3	0.15	1 0
I_4	0.05	1 1 0 0
I_5	0.04	1 1 0 1
I_6	0.03	1 1 1 0
I_7	0.03	1 1 1 1

3. 扩展编码

- 解：
  - 实际平均码长：

$$L=\sum p_i l_i = 2.30(\text{位})$$

  - 信息源熵：

$$H=-\sum p_i log_2 p_i = 2.17(\text{位})$$

  - 信息冗余量：

$$R=1-\frac{H}{\text{实际平均码长}}=1-\frac{2.17}{2.30}\approx 5.65\%$$

3. 扩展编码

- 一般采用等长扩展，以便于译码。
- 等长扩展编码：每次扩展的位数相等，便于实现分级译码。
- 有多种不同的等长扩展方法，如：
  - 4-8-12 （位数）
  - 15/15/15 （条数）
  - 8/64/512 （条数）等

3. 扩展编码

等长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

3. 扩展编码

等长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

3. 扩展编码

- 随着硬件、器件技术的发展，已经不再强调非要等长扩展了。

不等长操作码(4-6-10)扩展编码法

编码方法	指令操作码的长度			指令种类
	4位	6位	10位	
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

3. 扩展编码

- 具体采用哪种方法取决于所设计系统中的指令使用频度p_i的分布。
- 衡量的标准是看哪种编码方法的平均码长最短，码长种类最少，便于优化实现。

2.3.1 指令操作码的优化

- **注意：**

  无论是Huffman还是扩展操作码的编码方法，都不允许短码是长码的前缀，即短码不能与长码的开始部分的代码相同，否则将无法保证解码的唯一性和实时性。

例如：

若短码为：	则长码中的前4位中不能出现：
0000	0000
0001	0001
……	……
0111	0111

2.3.2 指令字格式优化

- 指令格式优化的目的：
  - 用最短的位数来表示指令的操作信息和地址信息，使程序中指令的平均字长最短；
  - 增加指令字所能表示的操作信息和地址信息，如寄存器数目和寻址方式类型；
  - 能够在具体实现中容易处理。

2.3.2 指令字格式优化

- 只对操作码进行优化，而没有对地址码和寻址方式采取相应的措施，程序总位数还是难以减少。
- **地址码优化：**
  - **一方面，**希望寻址范围越大越好。例如采用虚拟存储器可以使用比物理地址空间更大的逻辑地址空间。
  - **另一方面，**通过采取各种方法，在满足很大寻址范围的前提下，地址码的长度不一定非要那么宽。

2.3.2 指令字格式优化

- **地址码优化（续）：**
  - 可以用一个比较短的地址表示一个比较大的虚拟地址空间。如：
    - 用间址寻址方式
    - 用变址寻址方式
    - 用寄存器间址寻址方式
  - 可见，操作数的地址码可以在很宽的范围内变化，因此可以和可变长操作码配合。

2.3.2 指令字格式优化

- 操作码长度缩短，指令字中的空白增加。

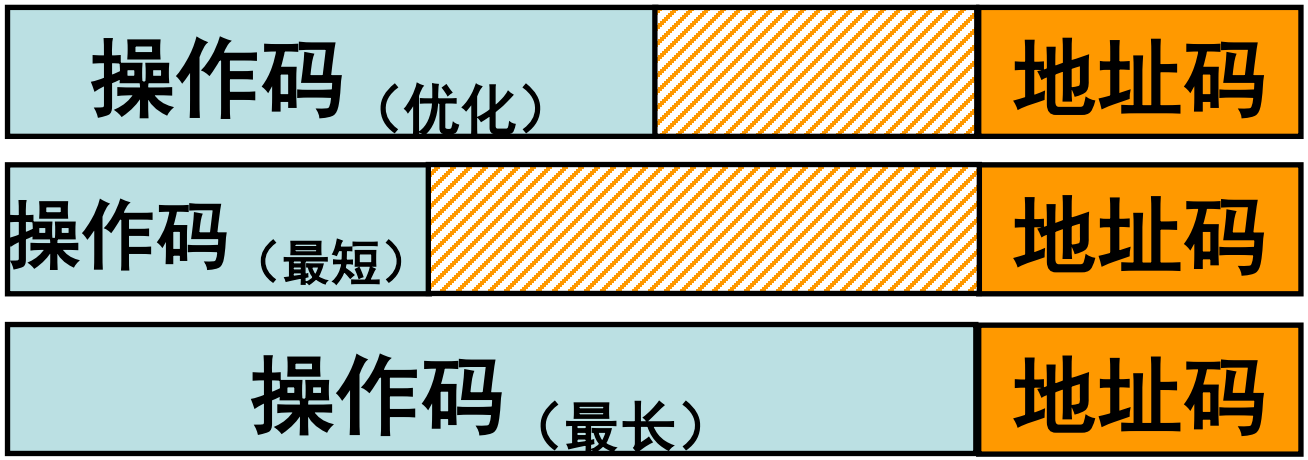


图　　等长地址码发挥不出操作码优化的作用

2.3.2 指令字格式优化

- 显然，只有地址码也是可变长的，才能用得上空白部分。
- **基本思想：** 可变长操作码与可变长地址码配合。
- **方法：** 通常采用长操作码配短地址码。

2.3.2 指令字格式优化

- **指令的长度：** 固定长度/可变长度。
- **操作码长度：** 固定长度/可变长度。
- **地址码长度：** 可变长度。
- 将三者有机结合起来。
- 为了不降低访问主存以取得指令的速度，**有必要维持指令字按整数倍边界存储的方式。**

1. 指令字长度固定

- **采用多种地址制：**由于各种指令所需要的操作数的个数会有不同，因此指令系统可以采用多种地址制，例如：
  - 三地址制
  - 二地址制
  - 一地址制
  - 零地址

1. 指令字长度固定

- 让最常用的指令操作码最短，其地址码字段的个数越多，指令的功能就越能增强，就越能从宏观上减少所需要的指令条数。

图　定长指令字中采用多种地址制

1. 指令字长度固定

- 即使为同一种地址制，还可以采用多种地址形式和长度，可以利用空白处存放直接操作数或常数等，例如：
  - 寄存器-寄存器型
  - 寄存器-存储器型
  - 带直接操作数等

1. 指令字长度固定

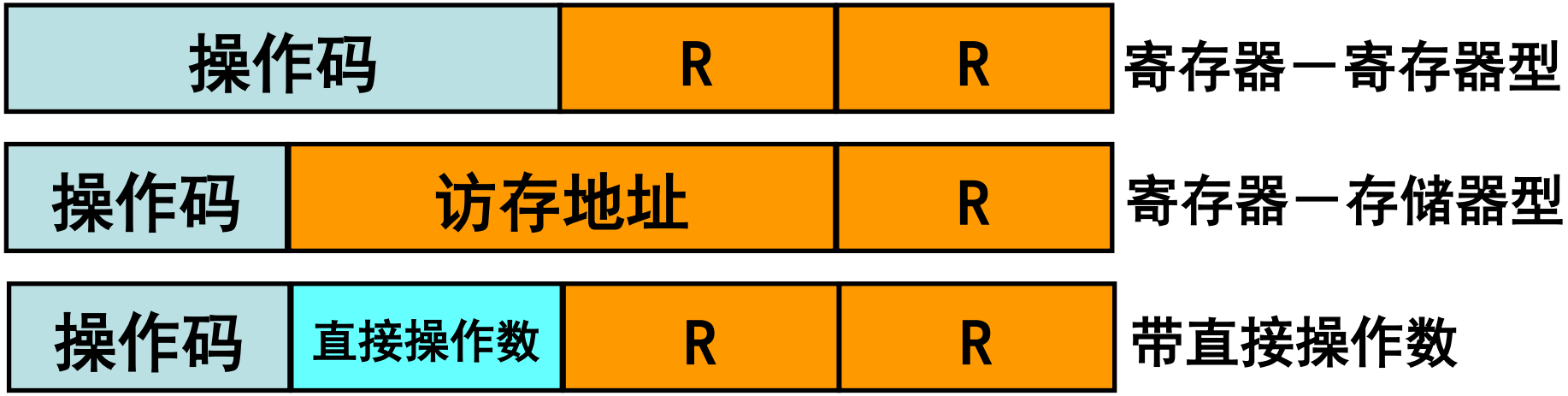


图 定长指令字中同种地址制下的
采用多种地址形式和长度

2. 指令字长度可变

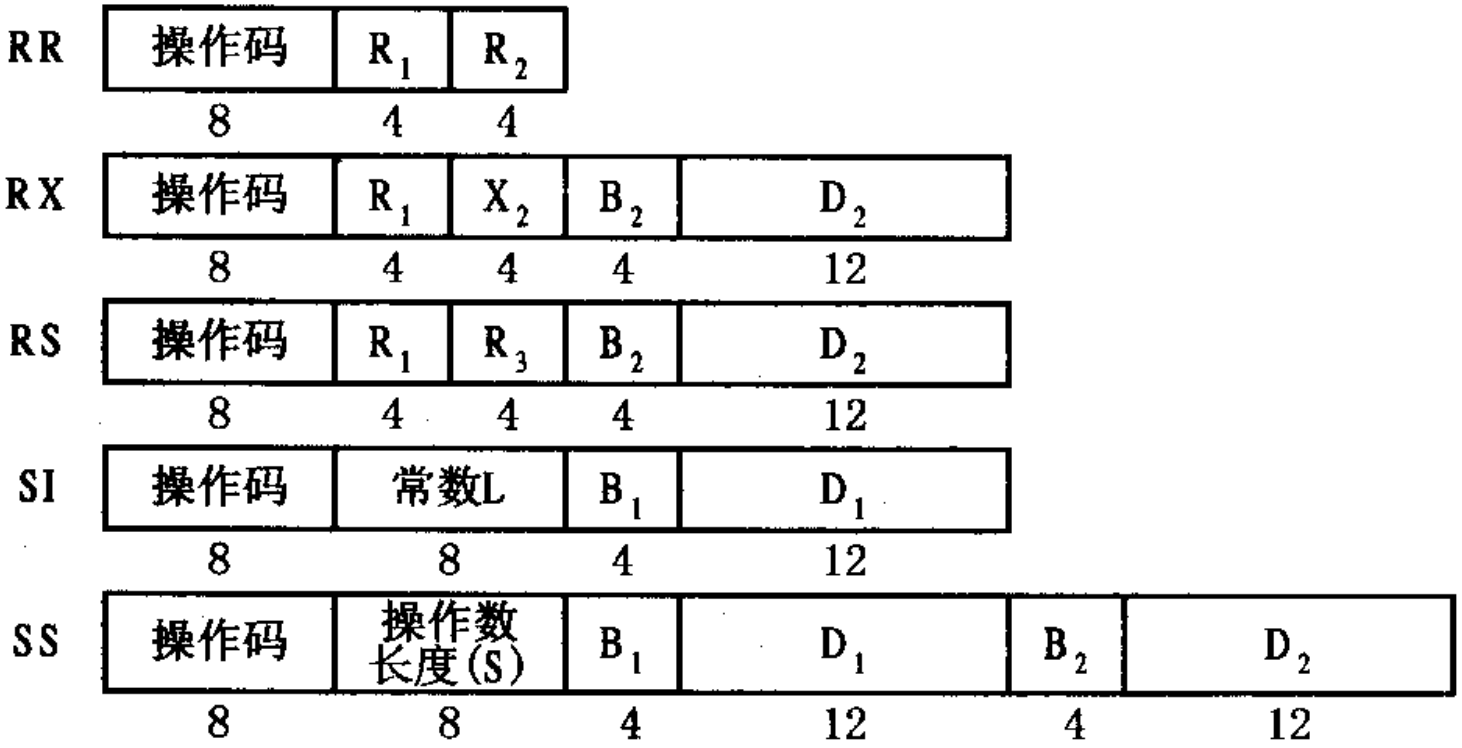
- 采用具有多种指令字长度的指令系统。
- 能有效地减少指令集的平均指令长度，降低目标代码的长度。
- 使得各条指令的字长和执行时间大不一样。
- 多数CISC计算机的指令集均采用这种格式。

2.3.2 指令字格式优化

- 总之，通过采用多种不同的寻址方式、地址制、地址形式和地址码长度，以及多种指令字长，并将它们与可变长操作码的优化表示相结合，就可以构成冗余度很小的指令字。
- 目前，指令字格式的优化技术已经被广泛应用。

2.3.2 指令字格式优化

- **IBM 370：大部分都采用 8 位定长操作码，采用 8 位操作码的主要指令格式有 5 种，如图所示。**



2.3.2 指令字格式优化

- IBM 370：其中R_1，R_2，R_3表示操作数寄存器和结果寄存器号，B_1，B_2表示基址寄存器号，D_1，D_2为相对位移量。它们分别由操作码的最高二位状态来指明其长度和格式。规则为：
  - 00 为RR格式，指令字长 16 位；
  - 01 为RX格式，指令字长 32 位；
  - 10 为RS或SI格式，指令字长 32 位；
  - 11 为SS格式，指令字长 48 位。

2.4 指令系统的发展与改进

- 指令系统的功能在很大程度上决定了机器的所具有的基本功能。
- 指令系统功能的设计和改进十分重要。
- 指令系统的设计、发展和改进有两种不同的途径和方向：
  - 增强指令系统的功能
  - 简化指令系统

2.4.1 CISC 和 RISC

- **增强指令系统的功能**
  - 进一步增强原有指令的功能。
  - 设置更为复杂、但功能更强的新指令以取代原先由软件子程序完成的功能，**实现软件功能的硬化。**
  - 按这种途径和方向发展，会使机器的指令系统越来越庞大和复杂，因此称采用这种途径设计而成的CPU的计算机为**复杂指令集计算机 (CISC － Complex Instruction Set Computer)**。
  - 如IBM 370，Intel i486，MC68040等。

2.4.1 CISC 和 RISC

- 简化指令系统
  - 减少指令总数，简化指令功能，以降低硬件设计的复杂度，提高指令的执行速度。
  - 按这种途径和方向发展，会使机器的指令系统精炼简单，因此称采用这种途径设计而成的CPU的计算机为精简指令集计算机(RISC － Reduced Instruction Set Computer)。
  - 20世纪70年代兴起，影响深远，是当前指令系统设计的主要趋势。
  - 典型系统或处理器有：Sun SPARC，Intel i860，HP3000的930和950、PowerPC等。

2.4.2 按CISC方向发展与改进指令系统

- **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。

time/program = (instructions/program) *x* (cycles/instruction) *x* (time/cycle)

- **从以下几个方面考虑：**
  - 面向目标程序的优化实现
  - 面向高级语言的优化实现
  - 面向操作系统的优化实现

1. 面向目标程序的优化实现

- 目的：
  - 减少目标程序占用的存贮空间；
  - 提高程序的运行速度 (减少访存次数，缩短指令执行时间)；
  - 更容易实现；

1. 面向目标程序的优化实现

- **改进方法1：利用哈夫曼思想改进指令**
  - 对于使用频度高的指令，可以：
    - 增强其功能
    - 加快其执行速度
    - 缩短其指令长度
    - 增设新指令来替代
  - 对于使用频度低的指令，可以：
    - 将其功能和并到某些高频指令中去
    - 在以后设计新系列机时，将其取消

1. 面向目标程序的优化实现

- **改进方法1（续）：**
  - **静态使用频度：** 对程序中出现的各种指令及指令串进行统计得出的百分比。
    - 按静态使用频度改进指令系统是着眼于减少目标程序所占用的**存储空间**。
  - **动态使用频度：** 在目标程序执行过程中对出现的各种指令及指令串进行统计得出的百分比。
    - 按动态使用频度改进指令系统是着眼于减少目标程序的**执行时间**。

1. 面向目标程序的优化实现

表　IBM 360 指令的动、静态使用频度

指令类型	静态使用频度%	动态使用频度%
L(取)	28.6	27.3
ST(存)	15.0	9.8
BC(条件转移)	10.0	13.7
LA(取地址)	7.0	6.1
SR(减)	5.8	4.5
A(加)		3.7
C(比较)		6.2
SLL(逻辑左移)	3.6	
BAL(转移与链接)	5.3	
IC(插入字符)	3.2	4.1

1. 面向目标程序的优化实现

- **改进方法2：增设强功能复合指令**
  - 增设少量新的、强功能复合指令，以取代原先由宏指令或子程序实现的功能；
  - 提高真正执行数据变换的加、减、乘、除等功能型指令的比例；
  - 优点：
    - 可以大大提高运算速度
    - 减少程序调用开销
    - 减少子程序所占用的主存空间

1. 面向目标程序的优化实现

- **改进方法2（续）：**
  - 最常见的指令是存、取和条件转移。
  - IBM 370 机上增设了用单条指令来完成多个数据传送的功能。 如“成组取”指令形式为：

成组取	R_1	R_3	B_2	D_2

  - “成组传送”指令可以实现字符行(即字节向量)的传送， 其形式为：

成组传送	L	B_1	D_1	B_2	D_2

1. 面向目标程序的优化实现

- 上述两种改进方法的**共同特点**是：
  - 不删改原有的指令系统；
  - 增加少量强功能新指令替代常用指令串或子程序；
  - 满足**软件向后兼容**的要求；
- 因此，这种改进受到用户和厂家的接受和欢迎。

2. 面向高级语言的优化实现

- 目的：
  - 缩短高级语言与机器语言的语义差异，以利于支持高级语言编译系统；
  - 缩短编译程序的长度和编译所需要的时间。

2. 面向高级语言的优化实现

- **主要改进方法：**
  - 统计使用频度来改进指令；
  - 面向编译、优化代码生成来改进指令；
  - 改进指令系统，缩小与各种语言的语义差异；
  - 让机器具有多种指令系统，多种系统结构；
  - 发展高级语言计算机；

2. 面向高级语言的优化实现

- (1) 统计使用频度来改进指令
  - 统计分析高级语言语句的使用频度；
  - 对于使用频度高的语句，可以设置专门的指令或采取措施增加相应指令的功能，以提高其编译速度和执行速度。

2. 面向高级语言的优化实现

表 各种语句的静态使用频度(%)

语句 / 语言	一元赋值	多元赋值	IF	GOTO	I/O	DO	CALL	其他
FORTRAN	31	15	11.5	10.5	6.5	4.5	6	15.0
COBOL	42.1	7.5	19.1	19.1	8.46	0.17	0.17	3.4

表 各种算术运算的静态使用频度(%)

算术运算类型 / 语言	加法		减法	乘法	除法	已置入函数	乘幂
	加“1”	其他					
FORTRAN	14	21.6	20.4	24.5	9.4	7.9	2.2
COBOL	18.5	44.4	2.4	17.8	16.2	/	0.7

2. 面向高级语言的优化实现

- (2)面向编译、优化代码生成来改进指令
  - 增加指令系统的**规整性**，尽可能减少例外情况和特殊用法，让所有运算都对称、均匀地在存贮单元或寄存器之间进行；
  - RISC机器就进一步发展了这个思路；

规整性：包括对称性和均匀性。

- **对称性**：所有寄存器同等对待，操作码的设置等都要对称，如：A－B与B－A。
- **均匀性**：不同的数据类型、字长、存储设备、操作种类要设置相同的指令。

2. 面向高级语言的优化实现

- (3)缩小与各种语言的语义差异
  - 造成代码生成困难、效率不高的主要原因是高级语言（包括编译过程产生的中间语言）与机器语言之间存在着很大的语义差异：
    - 不同高级语言之间的差别很大；
    - 至今难以统一出一种或少数几种通用的高级语言；
    - 即使是同一种高级语言也在不断发展，不同时期、不同厂家的实现也不尽相同；

2. 面向高级语言的优化实现

- (3)缩小与各种语言的语义差异

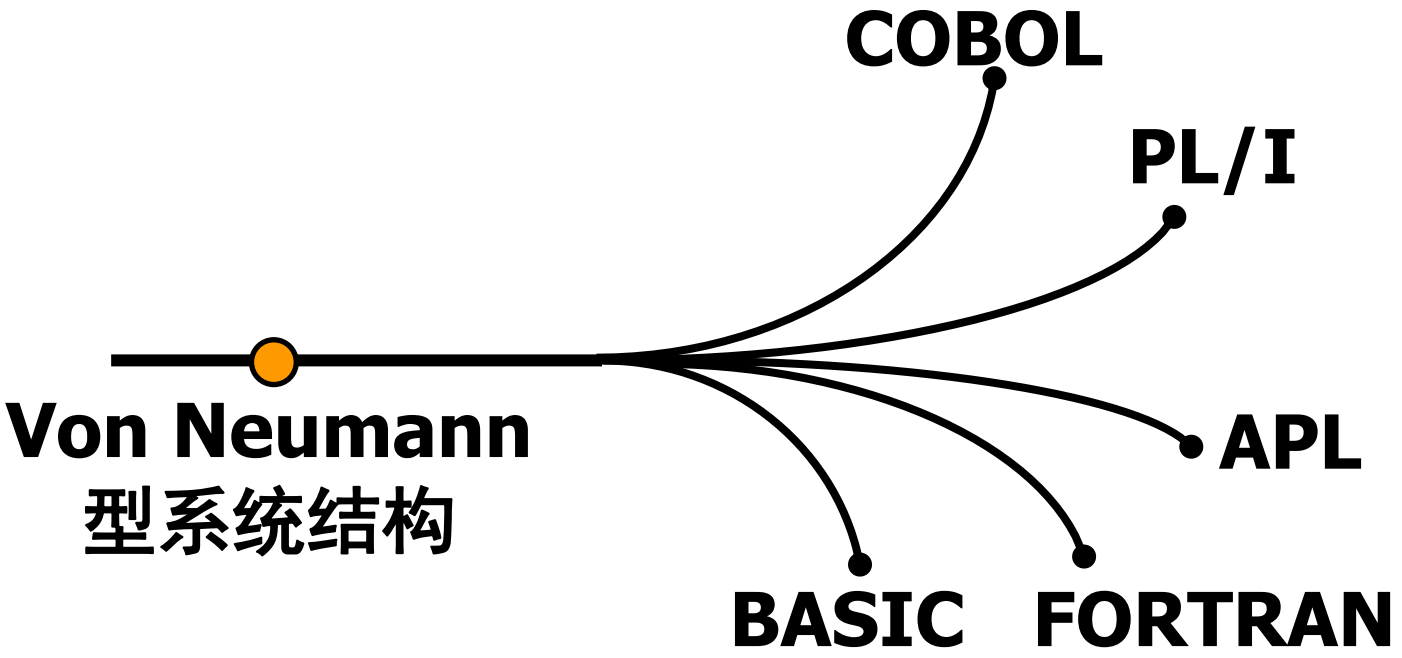


图 各种语言与传统机器指令系统的语义差异

2. 面向高级语言的优化实现

- (3)缩小与各种语言的语义差异
  - 解决方法1：
    - 把指令系统设计得较为通用和基本，即保持指令系统的通用性，使它对每种语言都远不是优化，只要通过编译程序能比较高效地实现即可。
  - 解决方法2：
    - 改进指令系统，使它与各种语言之间的语义差异有共同的缩小。在图上，就是把系统结构点向右移，使它与各种语言之间的路长都缩短。
  - 解决方法3：
    - 让机器具有多种指令系统，多种系统结构。

2. 面向高级语言的优化实现

- (3)缩小与各种语言的语义差异
  - 解决方法2：缩小语义差异

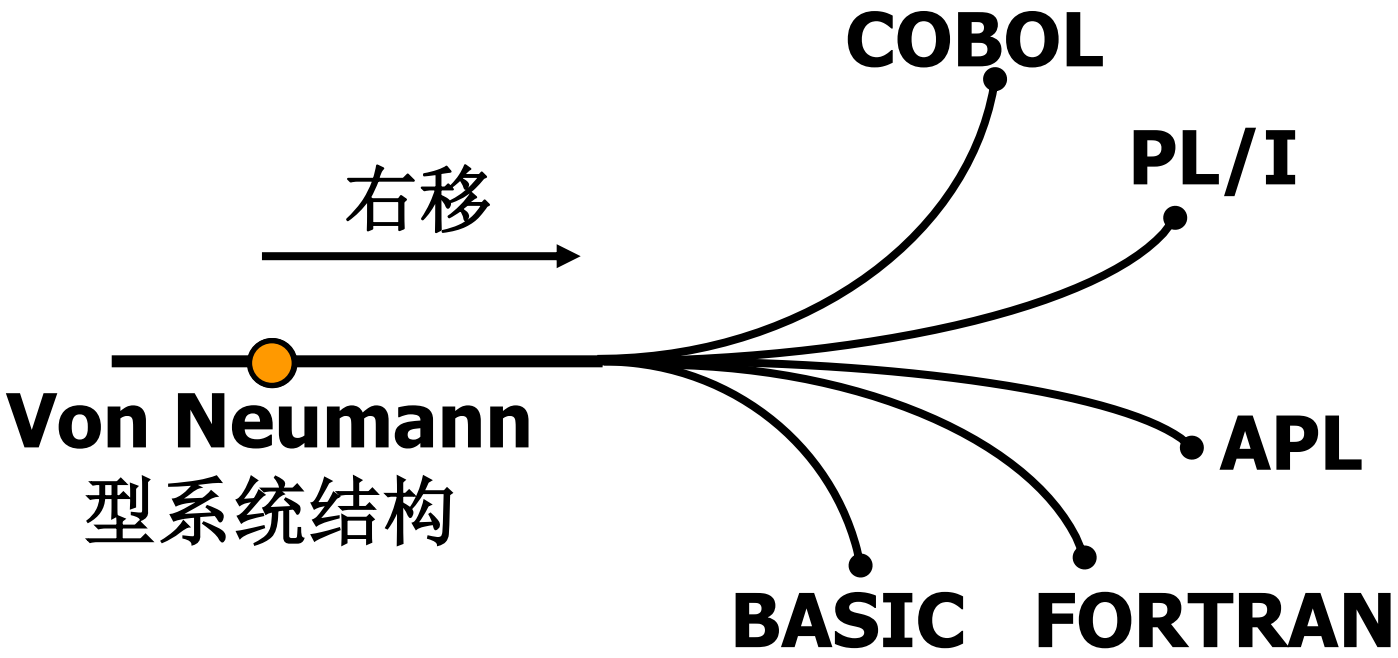


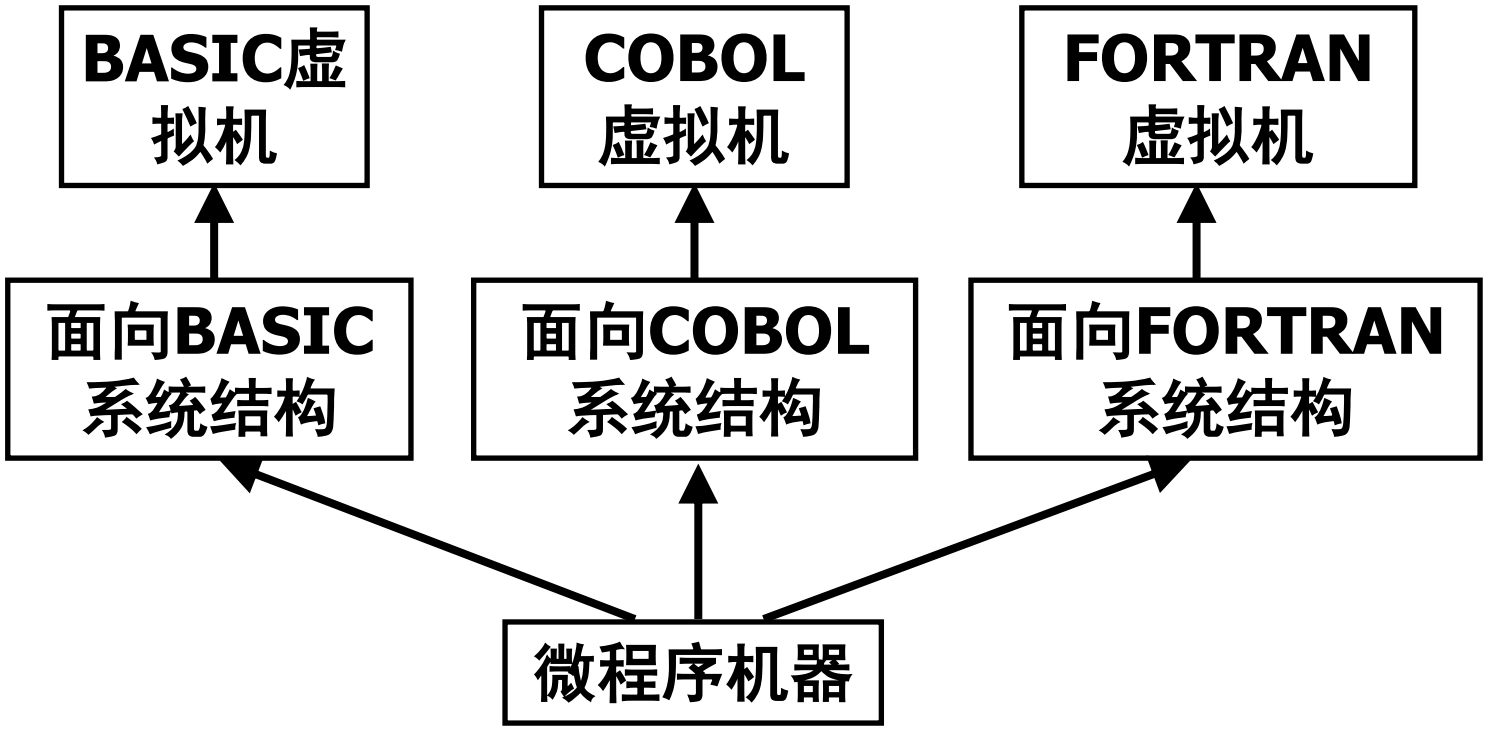
图 各种语言与传统机器指令系统的语义差异

2. 面向高级语言的优化实现

- (3)缩小与各种语言的语义差异
  - **解决方法3：**让机器具有多种指令系统，多种系统结构
    - 让机器具有多种指令系统，实现多种系统结构，并能动态切换；
    - 微程序的发展，特别是可读写控存的采用，给这种动态系统结构的实现提供了可能；
    - 1972年的B-1700及以后的B-1800，B-1900，70年代中期的IBM 5100等就是这样的思路；

2. 面向高级语言的优化实现

- **解决方法3（续）：**



2. 面向高级语言的优化实现

- (4)发展高级语言计算机
  - 翻译和解释是语言实现的两种基本技术。
    - 高级语言程序经过编译翻译为使用机器语言的目标程序；
    - 在微程序控制的机器上，机器语言程序用解释技术实现。
  - 缩小语义差异实际上意味着增大解释的比重，减少翻译的比重。

2. 面向高级语言的优化实现

(4)发展高级语言计算机（续）

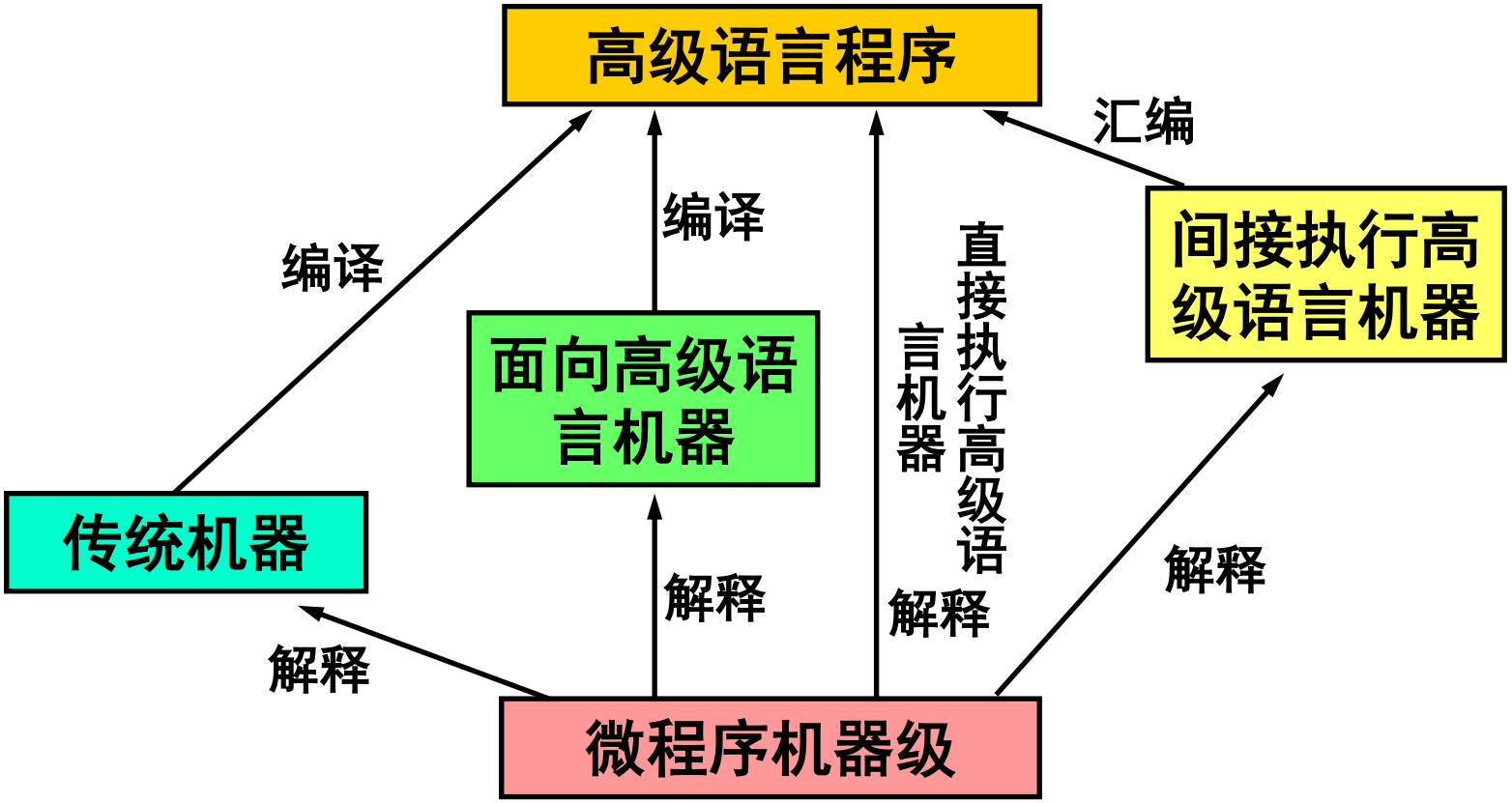


图　各种机器的语义差异

2. 面向高级语言的优化实现

- (4)发展高级语言计算机（续）
  - 高级语言计算机的基本特点：
    - 不需要编译
  - 两种发展形式：
    - 间接执行高级语言机器
    - 直接执行高级语言机器

2. 面向高级语言的优化实现

- (4)发展高级语言计算机（续）
  - 间接执行高级语言机器
    - 让高级语言直接成为机器的汇编语言，让高级语言与机器语言基本上一一对应；
    - 通过汇编程序把高级语言程序翻译成机器语言目标程序。
  - 直接执行高级语言机器
    - 把高级语言本身作为机器语言；
    - 由硬件或固件对高级语言程序的语句逐条解释执行，不需要编译，也不需要汇编。

3. 面向操作系统的优化实现

- 操作系统几乎占用了计算机系统资源的1/3，甚至1/2。
- 如果系统结构对操作系统的支持不够，不能使操作系统中诸多功能的实现具有更好的性能和更高的实现效率，那么计算机系统很难得到发展。

3. 面向操作系统的优化实现

- 但操作系统的实现不同于高级语言的实现，它更深地依赖于系统结构是否为它的实现提供了相应的硬件支持。
- 指令系统反映了这种支持的主要方面，但不全面。

3. 面向操作系统的优化实现

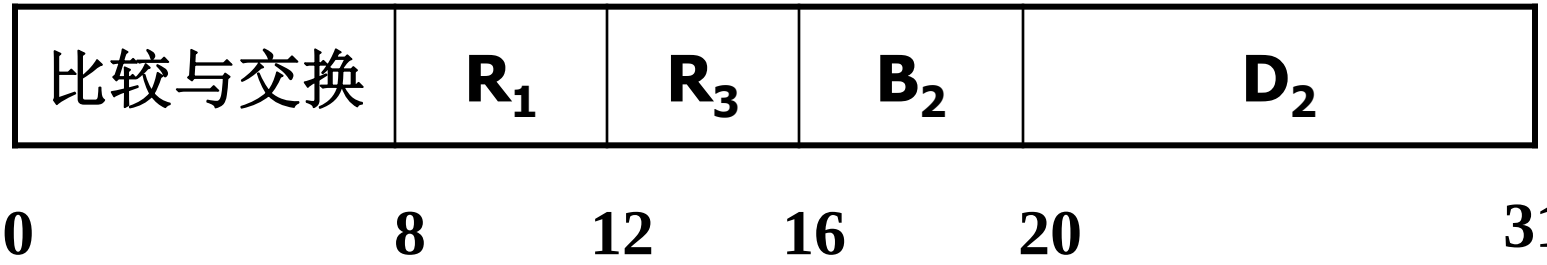
- 优化目标：
  - 缩短OS与系统结构语义差距；
  - 减少运行OS所需要的辅助操作时间；
  - 节省OS占用的存储空间。

3. 面向操作系统的优化实现

- **思路4种：**
  - 统计使用频度来改进；
  - 增设专用于OS的新指令；
  - 用硬件或固件实现OS的某些功能；
    - 把OS中使用频繁、对速度影响大的子程序进行硬化或固化，直接由硬件或微程序实现。
  - 由专门的处理机完成OS，实现功能分布处理系统结构。

3. 面向操作系统的优化实现

- **IBM 370就增设了如下的“比较与交换”(COMPARE AND SWAP)指令：**



对以下方面提供支持：

中断处理；进程管理；存储管理和保护；

系统工作状态的建立与切换；

3. 面向操作系统的优化实现

- **设置指令：**
  - 支持系统工作状态和访问方式转移的指令；
  - 支持进程转移的指令；
  - 支持进程同步和互斥的指令。

3. 面向操作系统的优化实现

- **注意：**
  - 尽可能缩小操作系统与系统结构的差异，并不意味用硬件或固件实现操作系统的全部功能；
  - 硬件和软件各有特点，应充分发挥硬件和软件的特长。

2.4.3. 按RISC方向发展与改进指令系统

- 通过精简指令，使计算机结构变得简单、合理、有效，并克服CISC结构的缺点。
- RISC已经成为计算机结构设计中的一种非常重要的思路。

1. 精简指令集思想的提出

- **CISC存在的问题**
  - 日益庞大复杂的指令系统实现越来越困难，而且还可能降低整个系统的性能。

机型（年代）	IBM 370/168 (1973)	VAX-11 (1978)	iAPX 432 (1982)
指令种类	208	303	222
微程序容量	420K	480K	64K
指令长度	16-48	16-456	6-321
制造工艺	ECL MSI	TTL MSI	NMOS VLSI
指令操作类型	Mem-Mem Mem-Reg Reg-Reg	Mem-Mem Mem-Reg Reg-Reg	面向堆栈 Mem-Mem
Cache容量	64K	64K	0

问题1：20%与80%规律

- CISC中，大约20%的指令占据了80%的处理机时间。
- 例如：8088处理机的指令种类约100种
  - 前11种（11%）指令的使用频度已经超过80%；
  - 前8种（8%）指令的运行时间已经超过80%；
  - 前20种（20%）指令使用频度达到91.1%，运行时间达到97.72%；
  - 其余80%指令使用频度只有8.9%，只占2.28%的处理机运行时间。

问题1：20%与80%规律

执行频率排序	80X86指令	指令执行频率（%执行指令总数）
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	1%
10	返回	1%
合 计		96%

问题1：20%与80%规律

- 大量测试表明：
  - 最常使用的是一些比较简单的指令，这类指令仅占指令总数的20%，但在各种程序中出现的频度却占80%，其余大多数指令是功能复杂的指令，这类指令占指令总数的80%，但其使用频度很低，仅占20%。因此，人们把这种情况称为“20%-80%律”。
  - 从“20%-80%律”出发，人们开始了对指令系统合理性的研究，提出了精简指令系统的想法，出现了精简指令系统计算机，简称RISC。

问题2：软硬件的功能分配问题

- 复杂的指令使指令的执行周期大大加长
  - 一般CISC处理机的指令平均执行周期CPI都在4以上，有些在10以上。
- CISC增强了指令系统功能，简化了软件，但硬件复杂了。

问题3： VLSI技术的发展引起的问题

- VLSI工艺要求规整性
  - 但CISC不适应VLSI工艺的要求。
- 主存与控存的速度相当
  - 简单指令没有必要用微程序实现；
  - 复杂指令用微程序实现与用简单指令组成的子程序实现没有多大区别。

1. 精简指令集思想的提出

- CISC结构和思路存在的主要问题：
  - (1) 指令系统庞大。
  - (2) 指令执行速度低。
  - (3) 编译程序本身太长、太复杂。
  - (4) 各种指令使用频度都不会太高，且差别很大。

1. 精简指令集思想的提出

- 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32位 小型机IBM 801。
- 1986年，IBM正式推出采用RISC体系结构的工作站IBM RT PC。

John Cocke

The "Father" of RISC architecture

1. 精简指令集思想的提出

- 1979年，美国加州Berkeley的David Patterson 研究小组开始研究RISC。
- 1981年研制出32位 RISC 微处理器RISC I（31种指令，3种数据类型，2种寻址方式）。
- 1983年研制出32位 RISC 微处理器RISC II。

David A.Patterson

RISC的定义与特点

- 卡内基梅隆大学论述RISC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。

RISC的定义与特点

- 90年代初，IEEE的Michael Slater对RISC定义的描述：
  - RISC为使流水线高效率执行，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。

RISC的定义与特点

- 90年代初，IEEE的Michael Slater对RISC定义的描述（续）：
  - RISC为使优化编译器便于生成优化代码，应具有：
    - 三地址指令格式；
    - 较多的寄存器；
    - 对称的指令格式。

2. RISC设计的基本原则

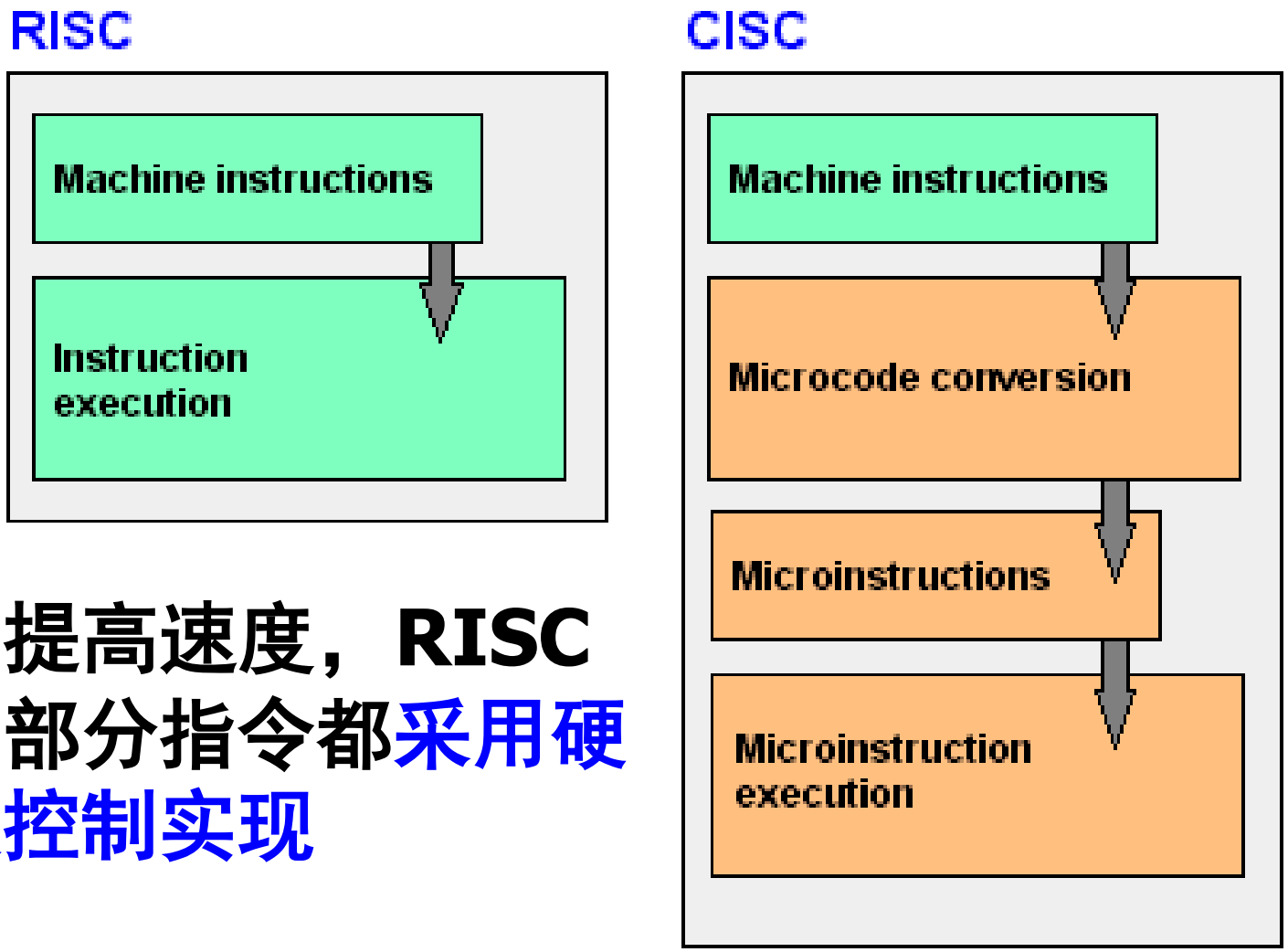
Patterson等人提出了精简指令系统计算机的设想。他们提出了设计RISC机器应当遵循的一般原则。

- 只选择使用频度很高的指令，增加少量支持操作系统和高级语言等的最有用的指令。指令条数一般<100条；
- 减少寻址方式的种类，一般 <2 种；
- 简化指令格式，一般限制在2种以内，并让所有指令有相同的长度；
- 所有指令都在一个机器周期内完成；

2. RISC设计的基本原则

- 扩大通用寄存器的个数，一般≥32个，尽可能减少访存，除STORE和LOAD指令外，其他指令的操作都在寄存器之间进行；
- 为提高速度，大部分指令都采用硬联控制实现，少量可采用微程序实现；
- 通过精简指令和优化设计编译程序，以简单有效的方式支持高级语言的实现。

2. RISC设计的基本原则



为提高速度，RISC大部分指令都采用硬联控制实现

3. “减少”CPI是RISC的精华

Simple is fast, Small is fast.

time/program = [(instructions/program) *x* (cycles/instruction) *x* (time/cycle)]

RISC的速度要比CISC快3倍左右，关键是RISC的CPI减小了。

表 CISC与RISC指令条数、CPI 和 IPC 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

3. “减少”CPI是RISC的精华

- **RISC设计思想也可以用于CISC中。**

例如：Intel公司的80x86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、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。

3. “减少”CPI是RISC的精华

- RISC优点：
  - 简化了指令系统设计，适合于VLI实现；
  - 提高了执行速度和效率；
  - 减低了成本，提高了可靠性；
  - 可以提供直接支持高级语言的能力，简化了编译程序的设计；

3. “减少”CPI是RISC的精华

- RISC缺点：
  - 指令少，指令功能简单，最大了程序占用空间，加重了汇编程序员的负担，加大了指令信息流量；
  - 对浮点运算和虚拟存储器的支持很强大，但不够理想；
  - 比CISC的编译程序难写。

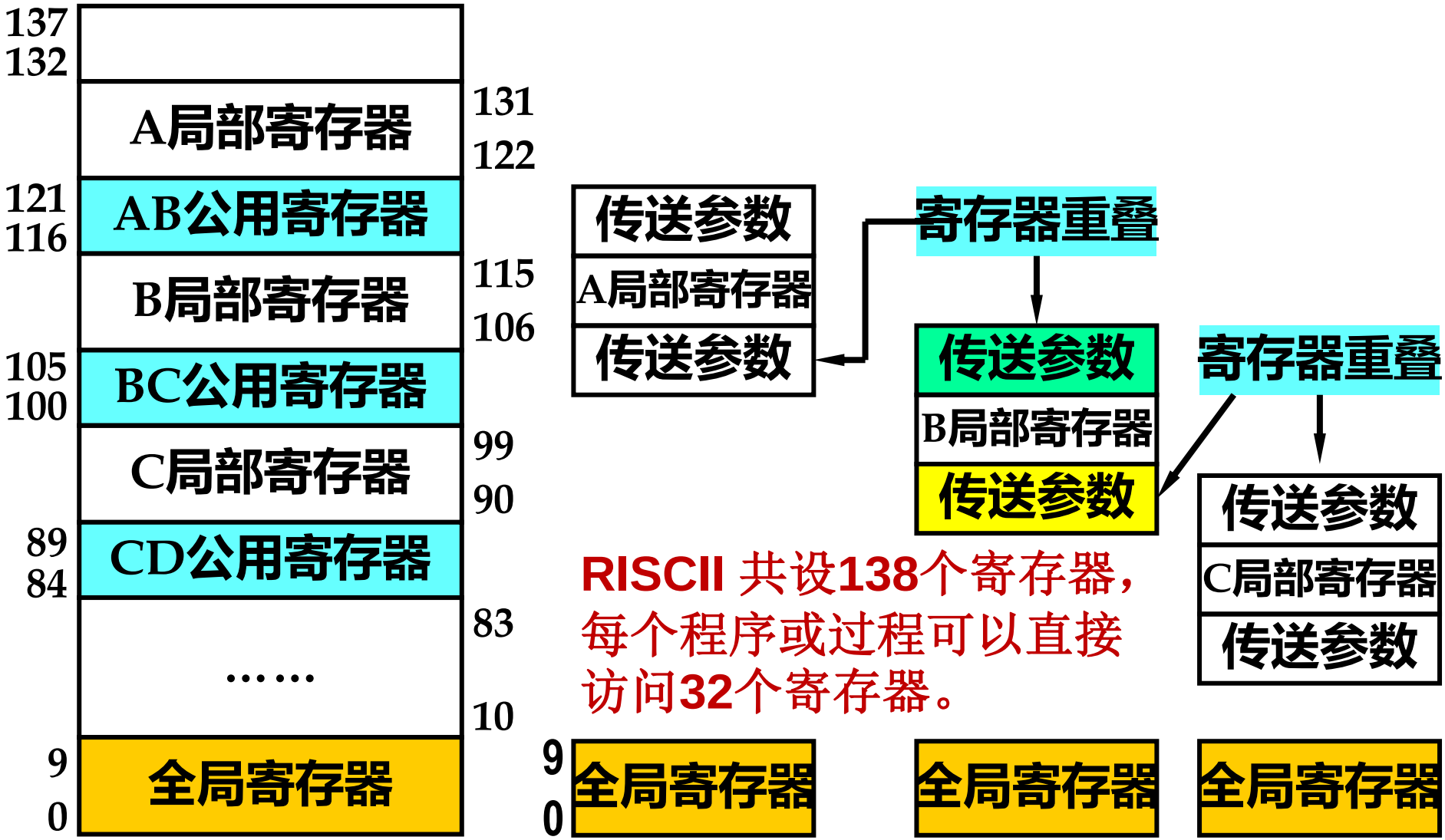
2.4.4 设计RISC的关键技术

- 1. 重叠寄存器窗口技术
- 2. 流水线技术与延时转移技术
- 3. 指令取消技术
- 4. 指令流调整技术
- 5. 优化编译系统设计的技术

1. 重叠寄存器窗口技术

- 由美国加洲大学伯克利分校的 F. Baskett 提出
- 原因：RISC中，子程序比CISC中多, 传送参数而访问存储器的信息量很大。
- 解决：让每个过程使用一个有限数量的寄存器窗口，并让各过程寄存器窗口部分重叠。

1. 重叠寄存器窗口技术


137
132
A局部寄存器
131
122
121
116
AB公用寄存器
B局部寄存器
115
106
105
100
BC公用寄存器
C局部寄存器
99
90
89
84
CD公用寄存器
.....
83
10
9
0
全局寄存器
传送参数
A局部寄存器
传送参数
寄存器重叠
传送参数
B局部寄存器
传送参数
寄存器重叠
传送参数
C局部寄存器
传送参数
RISCII 共设138个寄存器，
每个程序或过程可以直接
访问32个寄存器。
9
0
全局寄存器
全局寄存器
全局寄存器

1. 重叠寄存器窗口技术

寄存器窗口技术的效果

程序名称	Quick Sort	Puzzle
调用次数	111K (0.7%)	43K (8.0%)
最大调用深度	10	20
RISC II溢出次数	64	124
RISC II访问次数	4K (0.8%)	8K (1.0%)
VAX-11访问次数	696K (50%)	444K (28%)

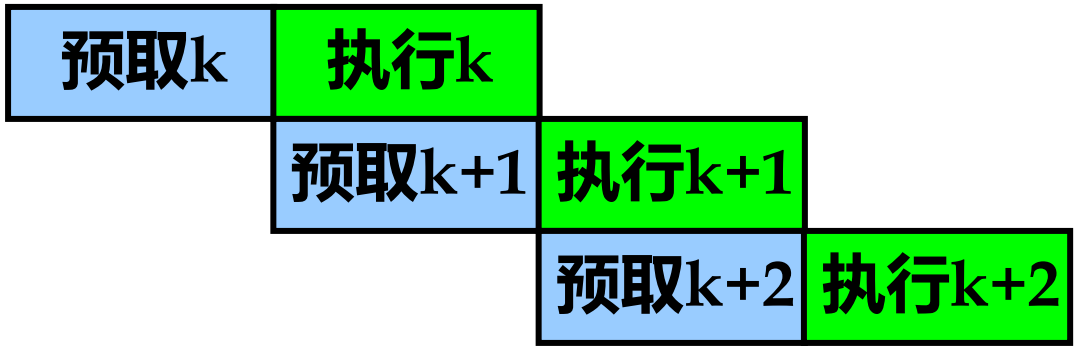
注：Quick sort程序的调用次数多，深度不大，而Puzzle程序正好相反。

2. 流水线技术与延时转移技术

■ 流水线技术

- 为在每个周期完成一条指令，RISC一般采用流水线技术：

本条指令的执行与下一条指令的预取相重叠。



■ 流水线的级数因机器而异。

2. 流水线技术与延时转移技术

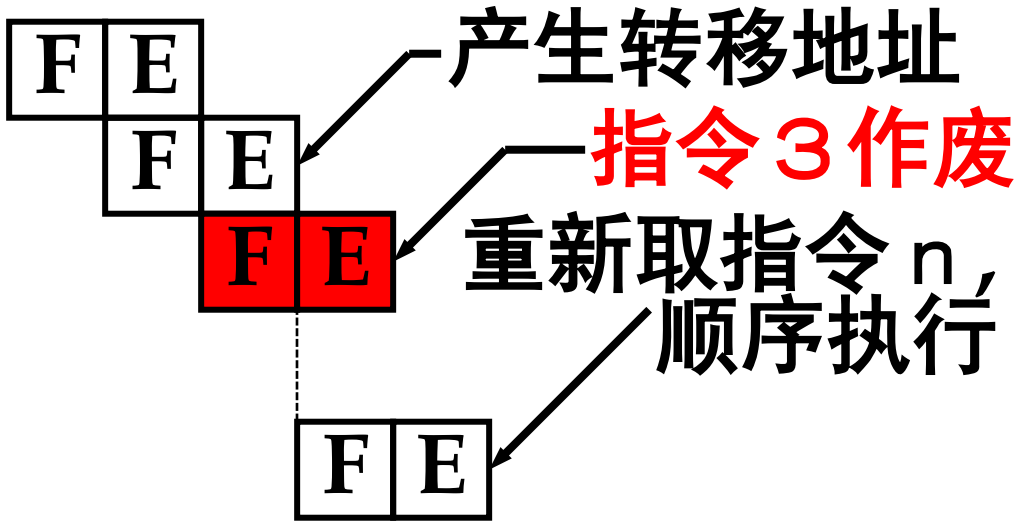
延迟转移技术

- 为避免无条件转移指令或成功条件转移指令造成指令预取浪费，辅助开销增大，流水线断流等问题，提出了延迟转移思想。
- 在转移指令之后插入一条不相关的有效指令，使预取不作废，流水线不断流，而转移指令好像被延迟执行了的技术称为延迟转移技术。

2. 流水线技术与延时转移技术

■ 延迟转移技术：调整指令顺序

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

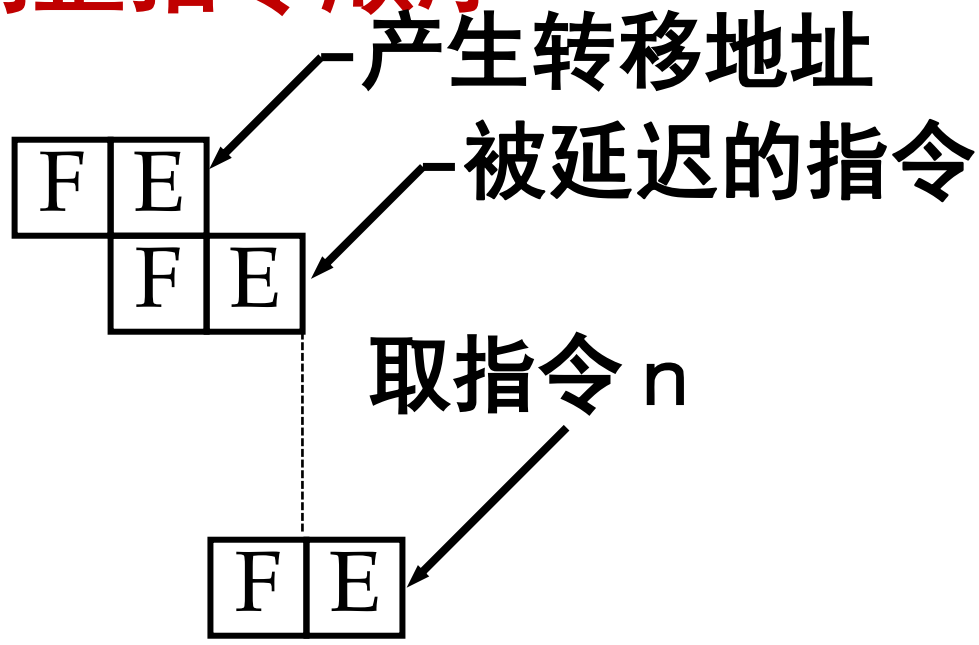


因转移指令引起的流水线断流

2. 流水线技术与延时转移技术

- 延迟转移技术：调整指令顺序

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



延迟转移指令不会引起流水线断流

2. 流水线技术与延时转移技术

■ 延迟转移技术：调整指令顺序

```
1:      move  r1, r2
2:      cmp   r3, r4     ;(r3)与(r4)比较
3:      beq   next       ;如果(r3)＝(r4)则转移到next
4:      add   r4, r5     ;转移成功时预取的该指令作废
        ……
n:  next:    move r4, a
```

因转移指令引起的流水线断流

2. 流水线技术与延时转移技术

- 延迟转移技术：调整指令顺序

```
1:      cmp   r3, r4     ;(r3)与(r4)比较
2:      beq   next       ;如果(r3)＝(r4)则转移到next
3:      move  r1, r2     ;转移成功时预取的该指令有效
4:      add   r4, r5
        ……
n:  next:    move r4, a
```

延迟转移指令不会引起流水线断流

2. 流水线技术与延时转移技术

- 例：设X、Y为主存单元，R_d, R_0, R_b, R_c为寄存器单元，且R_0中放值 0。有一个未采用延迟转移的程序：

指令地址	指　令	功　能
210	取 X，R_d	(X)→R_d
211	加 R_d，#1，R_d	(R_d)+1→R_d
212	条转〈条件〉，215	条件满足转 215，否则执行 213
213	加 R_d，R_0，R_b	(R_d)→R_b
214	减 R_b，R_c，R_b	(R_b)−(R_c)→R_b
215	存 R_d，Y	(R_d)→Y
216	……	

2. 流水线技术与延时转移技术

- 当执行到地址为 212 的条件转移指令时，如果转移成功，则预取的 213 指令就应作废，即不应将R_d的内容转送到R_b。
- **一种方法是：**为了保证程序的正确性，就应在 212 后面插入一条“加R_0，R_0，R_0”指令，相当于插入一条空操作指令(该指令执行结果仍然存的是 0)，如下面的左侧代码所示。因此，不管 212是否成功转移，都不会影响到其他运算的中间结果或最后结果，但这样，不管条件转移是否发生总要多花一个周期。
- **另一种方法是：**将转移指令之前的一条不相关指令调整到转移指令之后，如右侧代码所示。

2. 流水线技术与延时转移技术

210　取 X，R_d
211　加 R_d，#1，R_d
212　条转〈条件〉，216
213　加 R_0，R_0，R_0
214　加 R_d，R_0，R_b
215　减 R_b，R_c，R_b
216　存 R_d，Y

插入一条空操作指令 213

210　取 X，R_d
211　条转〈条件〉，215
212　加 R_d，#1，R_d
213　加 R_d，R_0，R_b
214　减 R_d，R_c，R_b
215　存 R_d，Y
216　……

将转移指令之前的一条不相关指令211调整到转移指令之后

2. 流水线技术与延时转移技术

延时转移技术

- 采用延迟转移技术的两个限制条件：
  - 1) 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。
  - 2) 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。
- 注意：延迟转移对用户程序是透明的，但对编译程序设计者不透明。

3. 指令取消技术

- 采用指令延时技术，在许多情况下找不到可以用来调整的指令，故有些RISC采用指令取消技术，分为三种情况：
  - (1) 向后转移（循环程序）
  - (2) 向前转移（if-then）
  - (3) 隐含转移技术

(1)向后转移（循环程序）

- 实现方法：
  - 循环体的第一条指令经调整后安排在两个位置，第一个位置是在循环体的前面，第二个位置安排在循环体的后面。
  - 如果转移成功，则执行循环体后面的指令，然后返回到循环体开始；否则，则取消循环体后面的指令，继续执行后面的指令。

(1)向后转移（循环程序）

- 例如　调整前

```
loop:  X X X
       Y Y Y
       ……
       Z Z Z
       cmp r1, r2, loop
       W W W
```

- 例如　调整后

```
       X X X
loop:  Y Y Y
       ……
       Z Z Z
       cmp r1, r2, loop
       X X X
       W W W
```

效果：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。

(2)向前转移（if - then）

- 实现方法：
  - 如果转移不成功执行下条指令，否则取消下条指令。

```
例如：R R R                    ; If部分的程序代码
        ……
        S S S                  ; If部分的程序代码
        cmp r1, r2, thru       ; 若转移，则取消TTT
        T T T                  ; Then部分的程序代码
         ……                    ;
        U U U                  ; Then部分的程序代码
thru:   VVV
```

效果：成功与不成功的概率通常各为50%。

(3)隐含转移技术

- **应用场合：**用于 if…then…结构，且then部分只有一条指令。
- **实现方法：**把if的条件取反，如果取反后的条件成立则取消下条指令，否则执行下条指令

例子：

```
 if (a<b) then b=b+1
```

改为：

```
cmp >=, ra, rb  ;若 (ra)>=(rb)则取消下条指令
inc rb
```

4. 指令流调整技术

- 目标：通过变量重新命名消除数据相关，提高流水线执行效率。
- 例：调整后的指令序列比原指令序列的执行速度快一倍。

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

5. 优化编译系统设计的技术

- RISC机器使用了大量寄存器，编译程序必须努力优化这些寄存器的分配和使用，以减少相关，减少访存次数等。
- 设计出高质量的编译程序是提高RISC机器性能的关键设计之一。

5. 优化编译系统设计的技术

- 设A、A+1、B、B+1 为主存单元，则程序：

取A, R_a ; (A)→R_a

存R_a, B ; (R_a)→B

取A+1, R_a ; (A+1)→R_a

存R_a, B+1 ; (R_a)→B+1

实现的是将A和A+1 两个主存单元的内容转存到B和B+1 两个主存单元。

5. 优化编译系统设计的技术

- 问题：R_a相关

  由于取和存两条指令交替进行，又使用同一个寄存器R_a，出现寄存器R_a必须先取得A的内容，然后才能由R_a存入B，即上条指令未结束之前，下条指令无法开始。后面的指令也是如此。因此，指令之间实际上不能流水，每条指令均需两个机器周期。

5. 优化编译系统设计的技术

- **解决：如果通过编译调整其指令的顺序为：**

取A, R_a ; (A)→R_a

取A+1, R_b ; (A+1)→R_b

存R_a, B ; (R_a)→B

存R_b, B+1 ; (R_b)→B+1

2.4.5 RISC发展

- 现在仍然在研究以下内容：
  - 减少指令条数；
  - 缩短指令执行的平均周期；
  - 缩短程序执行时间。

2.4.5 RISC发展

- 采取的措施包括：
  - 采用高效率的编码技术和更优化的算法；
  - 提高CPU的主频，缩短PCI；
  - 增大结构内部的并行性，提高IPC。包括：
    - 分别设立指令Cache和数据Cache
    - 增大Cache
    - 设立多端口寄存器组
    - 采用超级流水线 等。

2.4.5 RISC发展

- 将RISC和CISC概念和技术相结合
  - 为克服RISC的不足和问题，CPU的设计向着RISC和CISC概念和技术加以密切结合，互相取长补短的方向发展。
  - 例如：MC68030/Intel 80486/Pentium等虽然是CISC结构，但也引进某些RISC结构特点。

CISC	RISC
Price/Performance Strategies	
■**Price:** move complexity from software to hardware. ■**Performance:** make tradeoffs in favor of decreased code size, at the expense of a higher CPI.	■**Price:** move complexity from hardware to software ■**Performance:** make tradeoffs in favor of a lower CPI, at the expense of increased code size.
Design Decisions	
■A large and varied instruction set that includes simple, fast instructions for performing basic tasks, as well as complex, multi-cycle instructions that correspond to statements in an HLL. ■Support for HLLs is done in hardware. ■Memory-to-memory addressing modes. ■A microcode control unit. ■Spend fewer transistors on registers.	■Simple, single-cycle instructions that perform only basic functions. Assembler instructions correspond to microcode instructions on a CISC machine. ■All HLL support is done in software. ■Simple addressing modes that allow only LOAD and STORE to access memory. All operations are register-to-register. ■direct execution control unit. ■spend more transistors on multiple banks of registers. ■use pipelined execution to lower CPI.

	CISC	RISC
价格	硬件复杂，芯片成本高	硬件较简单，芯片成本低
性能	减少代码尺寸，增加指令的执行周期数	使用流水线降低指令的执行周期数，增加代码尺寸
指令集	大量的混杂型指令集，有专用指令完成特殊功能	简单的单周期指令，不常用的功能由组合指令完成
应用范围	通用机	专用机
功耗与面积	含有丰富的电路单元，功能强、面积大、功耗大	处理器结构简单，面积小，功耗小
设计周期	长	短

2.5 典型RISC处理器

- 后PC时代 为 普适计算 时代
- 普适计算
  - 强调和环境融为一体的计算，而计算机本身则从人们的视线里消失。在普适计算的模式下，人们能够在任何时间、任何地点、以任何方式进行信息的获取与处理。
- 嵌入式系统
  - 一种“完全嵌入受控器件内部，为特定应用而设计的专用计算机系统”

2.5.1 ARM嵌入式处理器

- ARM（Advanced RISC Machines）公司是全球领先的16/32位RISC微处理器知识产权设计供应商。
- ARM公司通过转让它的高性能、低成本、功耗低的RISC微处理器、外围和系统芯片设计技术给合作伙伴来生产各具特色的芯片。ARM公司已成为移动通信、手持设备、多媒体数字消费嵌入式解决方案的RISC标准。

2.5.1 ARM嵌入式处理器

- ARM公司位于英国，1990年成立Advanced RISC Machines Limited（后来简称为ARM Limited）。
- 20世纪90年代，ARM扩展到世界范围，占据了高性能、低功耗、低成本的嵌入式应用领域的领先地位。

2.5.1 ARM嵌入式处理器

- ARM处理器有三大特点：
  - 1. 小体积、低功耗、低成本而高性能；
  - 2. 16/32位双指令集；
  - 3. 全球众多的合作伙伴。

2.5.1 ARM嵌入式处理器

- ARM系列处理器

  ARM7、ARM9、ARM9E、ARM10

  ARM11、Cortex和SecurCore。

- 其中ARM7是低功耗的32位核，最适合应用于对价位和功耗敏感的产品。ARM9以上是高端产品，智能手机中使用ARM处理器。

2.5.1 ARM嵌入式处理器

- Cortex处理器采用ARMv7体系结构。在命名方式上，不再延用ARM加数字编号的命名方式,而是以Cortex命名。基于v7A的称为“Cortex-A系列”，基于v7R的称为“Cortex-R系列”，基于v7M的称为“Cortex-M系列”
- Cortex-A系列是针对日益增长的，运行包括Linux、Windows CE和Symbian操作系统在内的消费娱乐和无线产品；
- Cortex-R系列是针对需要运行实时操作系统来进行控制应用的系统，包括汽车电子、网络和影像系统；
- Cortex-M系列则是为那些对开发费用非常敏感同时对性能要求不断增加的微控制器应用所设计的。

2.5.1 ARM嵌入式处理器

- ARM公司针对手机移动游戏市场推出ARMv8体系结构(又称架构)的64位微处理器，支持Android Lollipop。
- ARMv8-A 64位架构处理机有2种执行模式：AArch32和AArch64。
  - AArch32与ARMv7A完全兼容，支持Thumb2和ARM指令集。AArch32与64位AArch64一起（side-by-side）实现原生的（native）32位AArch32执行，由于没有仿真因而32位代码能全速运行，与运行在AArch64执行状态的应用可完全共存。
  - AArch64采用新的更现代化的A64指令集架构ISA（instruction set architecture）

2.5.2 MIPS嵌入式处理器

- Microprocessor without Interlocked Pipeline Stages。
- MIPS技术公司是一家设计制造高性能、高档次及嵌入式32位和64位处理器的厂商。在RISC处理器方面占有重要地位。
- MIPS公司设计RISC处理器始于20世纪80年代初， MIPS公司的战略发生变化，把重点放在嵌入式系统。

2.5.2 MIPS嵌入式处理器

- MIPS处理器由斯坦福（Stanford）大学Hennessy教授领导的研究小组研制出来的。
- MIPS公司的R系列就是在此基础上开发的RISC工业产品的微处理器。
- MIPS是出现最早的商业RISC架构芯片之一，
- 国外计算机体系结构教材《Computer Architecture: A Quantitative Approach 》由Standford大学的John L. Hennessy和加州大学Berkely分校的David A . Patterson编写，以MIPS指令集作为教学实例。

2.5.2 MIPS嵌入式处理器

- MIPS公司陆续开发了高性能、低功耗的32位处理器内核（core）MIPS 32 4Kc与高性能64位处理器内核MIPS 64 5Kc。
- 中科院计算所的龙芯CPU与MIPS兼容

2.6 Intel嵌入式处理器

- Intel公司现在主推EIA(Embedded Intel Architecture)嵌入式架构。
- 英特尔公司非常明确的是除PC机、笔记本和服务器外的都是嵌入式系统。
- 现在英特尔有两个很重要的方向是低功耗和高集成度SOC的设计。
- 英特尔力推嵌入式Linux开发，推出Atom处理器（凌动）和Quark（夸克）处理器。

2.6.1 Intel Atom处理器

- Intel Atom为新一代移动网络设备平台（MID, Mobile Internet Device），是近来非常热门的移动产品，轻巧便携，方便日常办公。随着体积的减小，产品的散热是需要考虑的问题。
- Intel Atom采用45纳米工艺制造。Atom专门为小型设备设计，旨在降低产品功耗，同时也保持了同酷睿2双核指令集的兼容，产品还支持多线程处理。
- Intel Atom 使用16级指令流水线为了达到低功耗并且延长电池的寿命。

2.6.1 Intel Quark处理器

- 与英特尔现有的凌动处理器相比，Quark的体积为其五分之一，功耗仅为其十分之一。英特尔将利用这款小体积、低功耗的Quark处理器进军物联网和可穿戴产品市场。
- “夸克”源自1997年的第一代奔腾（P54C），还是32位的，只不过采用了32nm工艺制造，体积得以缩小到这种程度。
- 当然，夸克并不是奔腾完全照搬而来。它借鉴了原来的架构核心设计，对非核心扩展部分进行适当的更新拓展。

本章重点

- 高级数据表示；自定义数据表示方法（带标志符的数据表示法和数据描述符表示法）；
- 程序定位；
- 指令格式的优化设计（重点：操作码的优化）；
- Huffman编码和扩展编码的区别及计算；
- CISC存在的问题；20-80律；
- RISC基本原则，RISC优缺点；
- RISC关键技术；
- CISC与RISC的比较

第2章 作业2