第二章

数据表示、寻址方式与指令系统

Dr. Feng Li

fli@sdu.edu.cn

https://funglee.github.io

Outline

- 数据表示
- 寻址方式
- 指令系统的设计和优化
- 指令系统的发展和改进

数据表示与数据结构

- 数据表示：指的是能由机器硬件直接识别和引用的数据类型。由硬件实现的数据类型
- 数据结构：应用中要用到的各种数据元素或信息单元之间的结构关系。由软件实现的数据类型。
- 数据结构要通过软件映像，变换成机器中所具有的数据表示来实现。
- 不同数据表示可以为数据结构提供不同的实现，表现为实现 效率和方便性的不同。
- 数据表示的确定实质上是软、硬件的取舍问题
- 数据结构和数据表示是软、硬件的界面

- 机器的运算类指令和运算器结构主要是按照机器有什么样的数据表示来确定
  - 定点数运算指令和部件→定点数表示
  - 浮点数运算指令和部件→浮点数表示
  - 变址寄存器和变址加法器→向量、阵列数据表示

高级数据表示

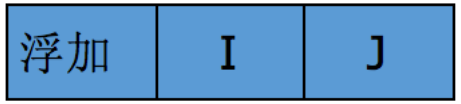
- 为复杂数据结构的实现提供更好的支持
- 自定义数据表示
  - 标志符数据表示
  - 数据描述符
- 向量、数组数据表示
- 堆栈数据表示

标志符数据表示

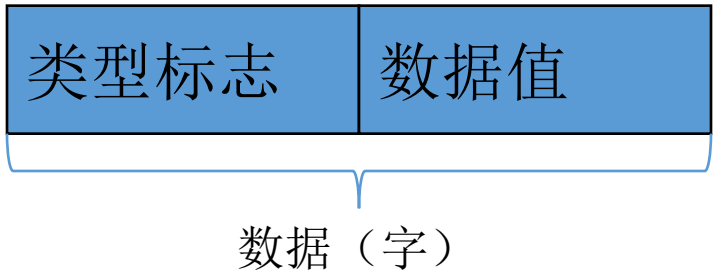
- 缩短高级语言与机器语言的语义差距
  - 高级程序语言（FORTRAN）

```
REAL I, J
I = I + J
```

  - 传统机器语言



- 类型标志位：说明数据值是二进制整数、十进制整数、浮点数、字符串还是地址字，将数据类型与数据本身直接联系起来，而非用操作码来说明数据类型
- 由编译程序建立，对高级语言程序透明，以减轻程序员负担

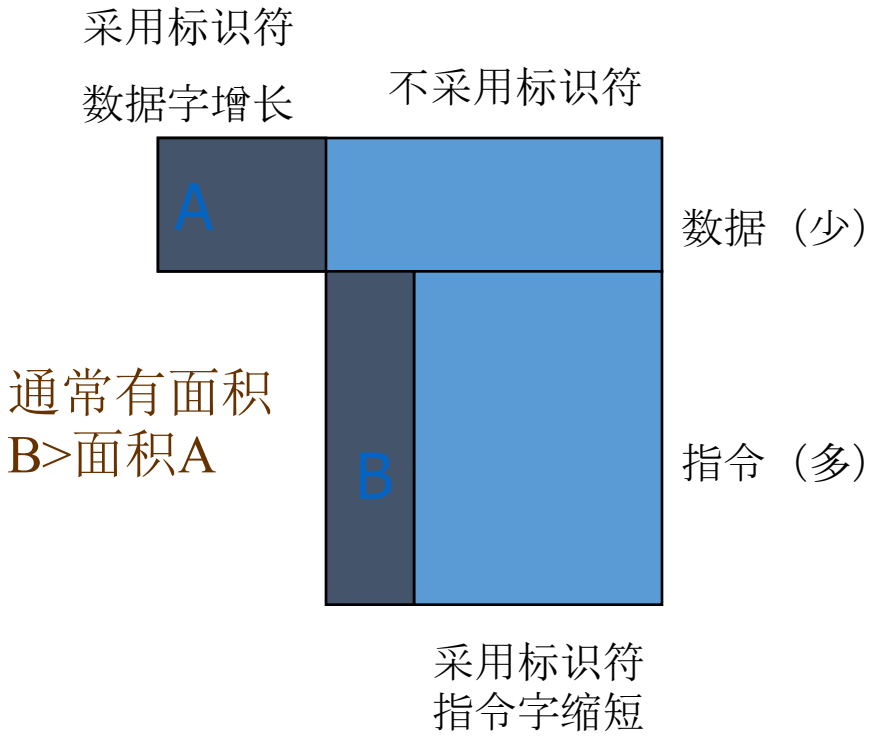


标志符数据表示的优点

- 简化指令系统和程序设计
- 简化编译程序
- 便于通过硬件实现一致性校验
- 能由硬件自动变换数据类型
- 支持数据库系统的实现与数据类型无关的要求，使程序不用修改即可处理多种不同类型的数据
- 为软件调试和应用软件开发提供了支持

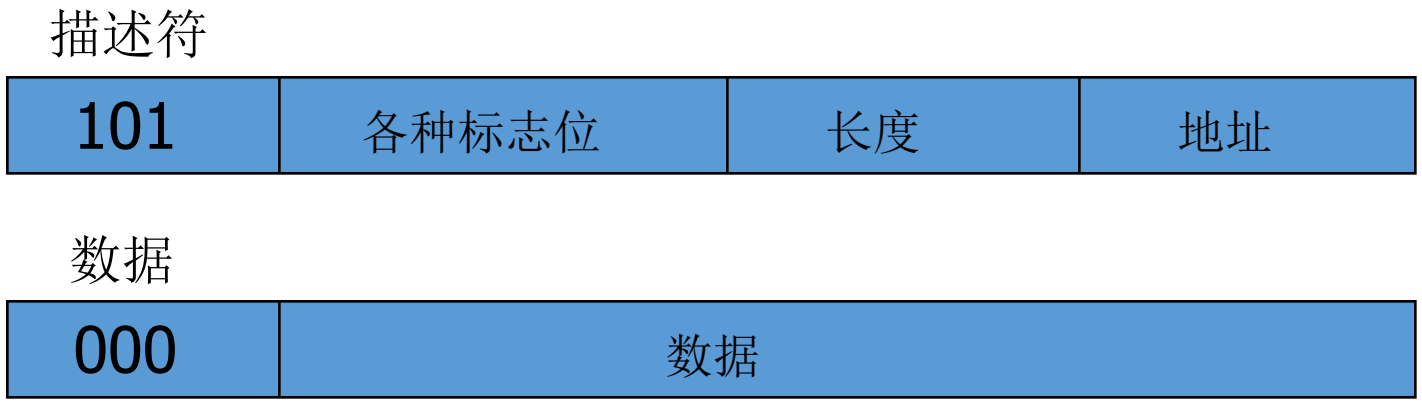
标志符数据表示

- 每个数据字因增设标志符，会增加程序所占用的空间主存，但由于简化了指令系统，缩短了操作码位数
- 采用标志符会降低单条指令的执行速度，但从总体上看，由于程序的编制和调试时间的缩短，从而使解题的总开销时间也减少了



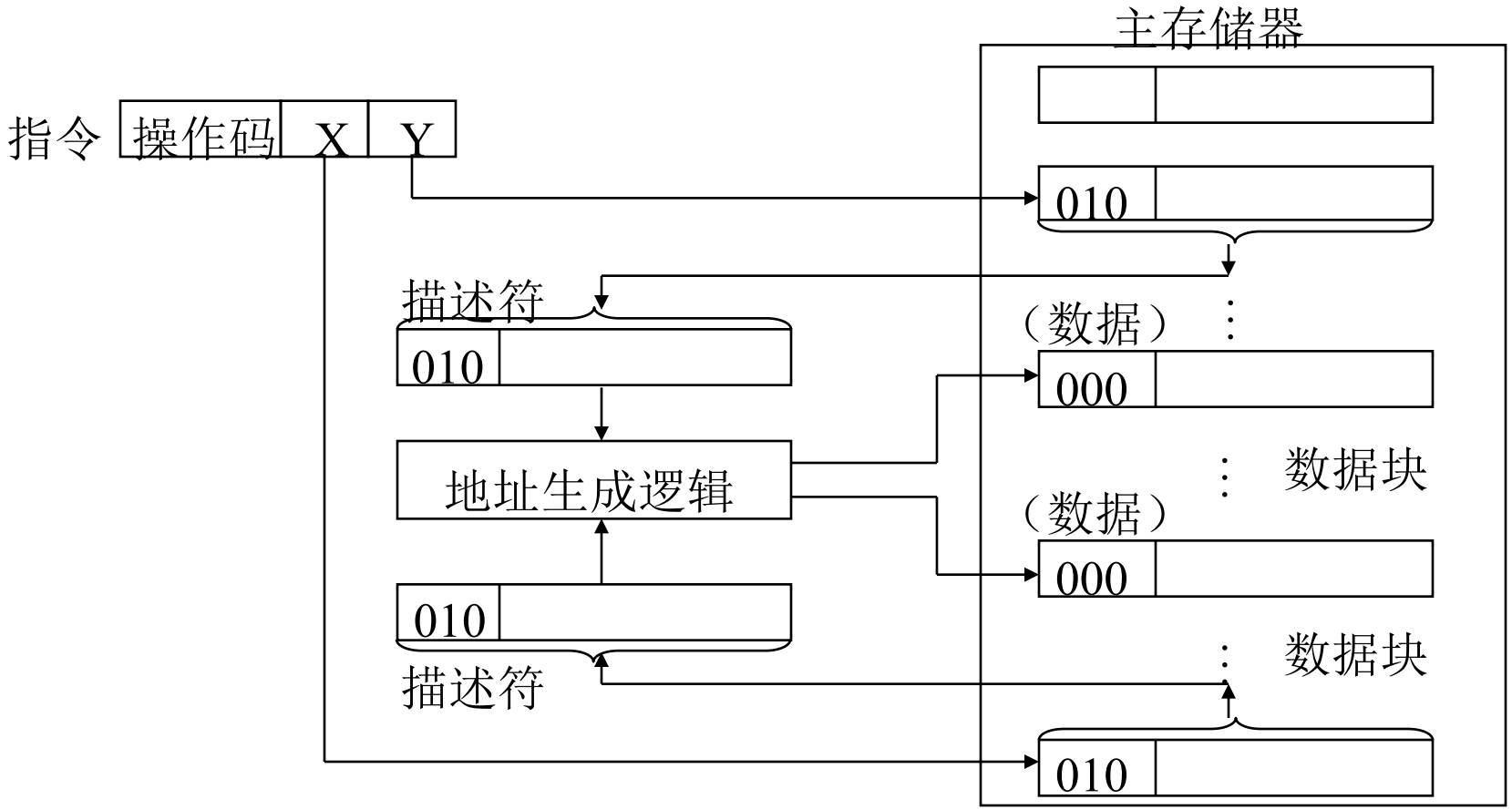
数据描述符

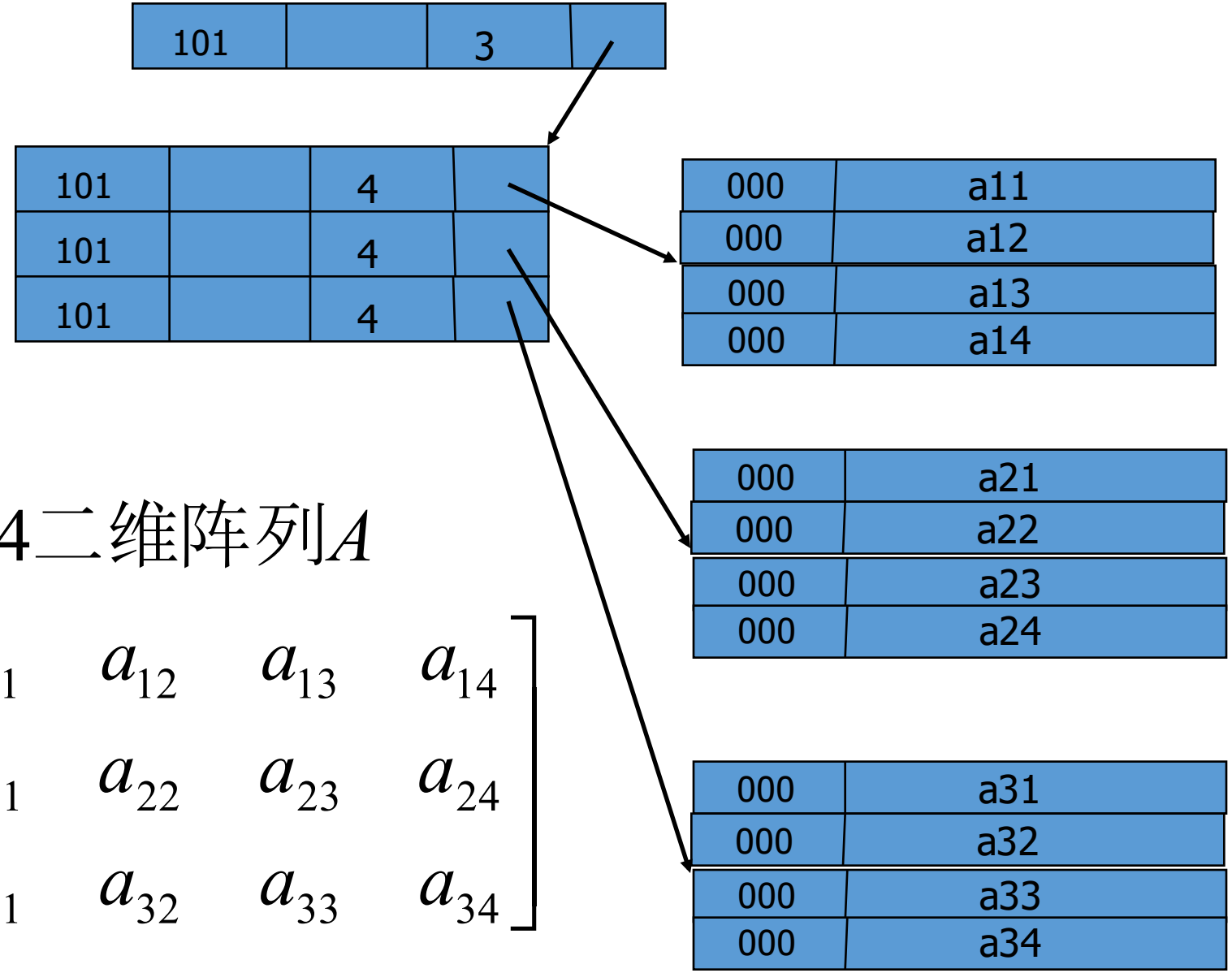
- 数据标志符：和每个数据相连，合并在一个存储单元中，描述当个数据的类型特征
- 数据描述符：与数据分开存放，描述所要访问的数据是整块还是单个，访问该数据块或者数据元素所要的地址以及其他信息



B6700的数据描述符和数据的形式

描述符的工作过程


主存储器
指令
操作码
X
Y
010
描述符
010
地址生成逻辑
010
描述符
（数据）
000
数据块
（数据）
000
数据块
010



3*4二维阵列A

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

数据描述符

- 实现阵列数据的索引比变址方法实现的好，而且能检查程序设计中阵列越界错误。
- 为向量、数组数据结构的实现提供一定的支持，有利于简化编译中的代码生成。

向量、数组数据表示

- 为向量、数组数据结构的实现和快速计算提供更好的支持
- 举例：

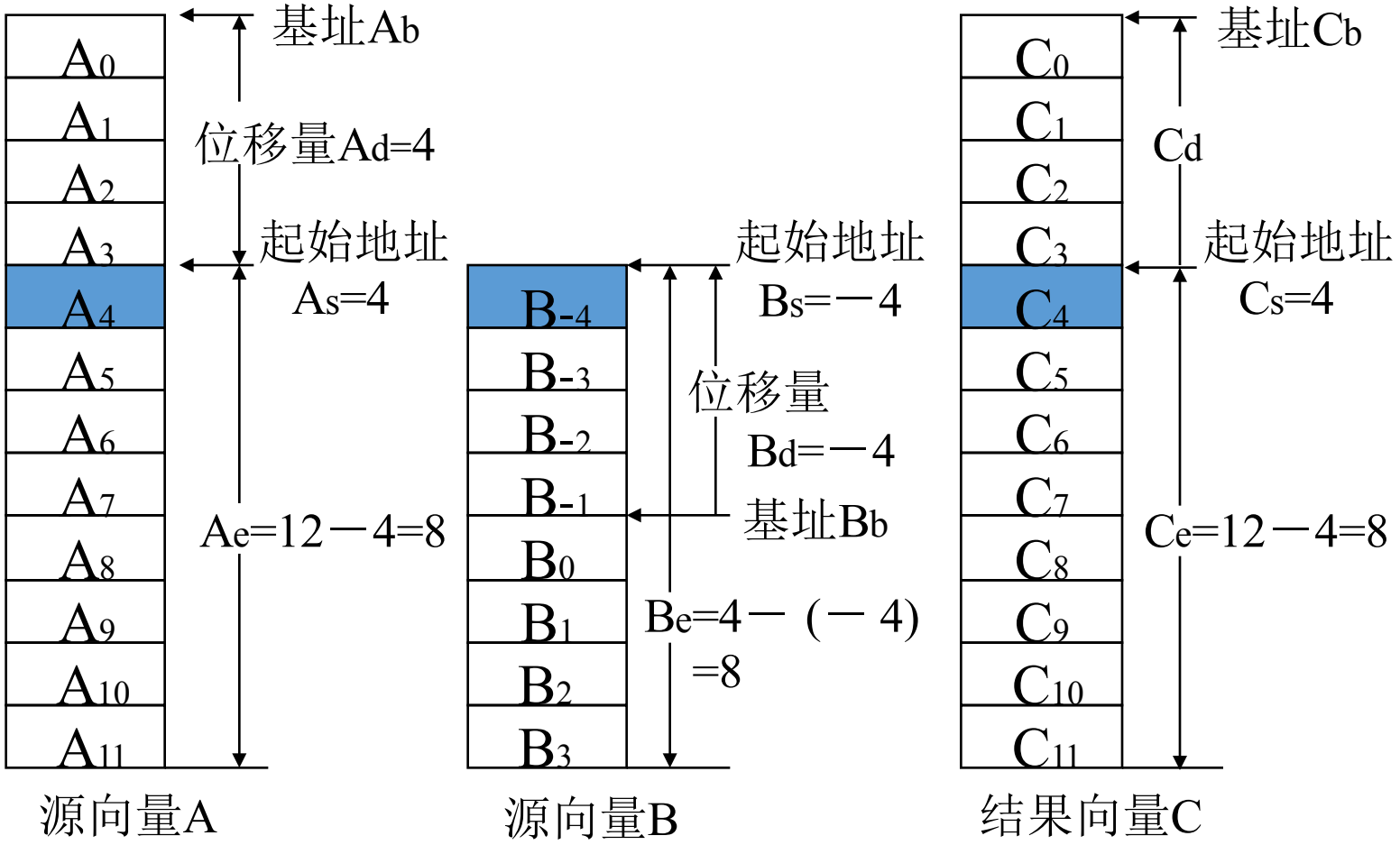
  计算 i=10、11….1000

  C语言： for (i=10; i<=1000; i++)　　c[i]=a[i]+b[i] ;
- 变址操作难以实现向量、数组计算的并行处理

- 在有向量、数组表示的向量处理机上，硬件上设置丰富的向量或者阵列运算指令，配有流水线或阵列方式处理的高速运算器

向量加法指令

向量加	A向量参数	B向量参数	C向量参数



引入向量、数组数据表示的好处

- 加快形成元素地址
- 对整个向量、数组成块地预取到，实现对整个向量、数组的高速处理
- 使用硬件进行越界判断，让越界判断和元素运算并行
- 增设对相关型交叉阵列进行处理的硬件，加快数据处理
- 设置相关指令和硬件对数据进行压缩、还原和运算，节省存储空间和处理时间
- 简化编译程序

- 向量处理机（向量计算机）（Vector Processor）：具有向量表示和相应的向量运算指令的计算机。

- 标量处理机（标量计算机）（Scalar Processor）：不具有向量表示和相应的向量运算指令的计算机。

堆栈数据表示

- 有利于编译和子程序调用
- 堆栈机器的特点
  - 有若干高速寄存器组成的硬件堆栈，并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，堆栈的容量是主存的
  - 有很丰富的堆栈操作类指令且功能很强，直接可对堆栈中的数据进行各种运算和处理
  - 有力地支持高级语言程序的编译 （逆波兰表达式）
  - 有力地支持子程序的嵌套和递归调用

引入高级数据表示的原则

- 原则1：看系统效率是否显著提高，包括实现时间和存储空间是否有显著减少
  - 举例1：两个200*200的二维定点数组相加

    PL/1 A=A+B,

    无阵列型：6条指令，4条循环 200*200=40000

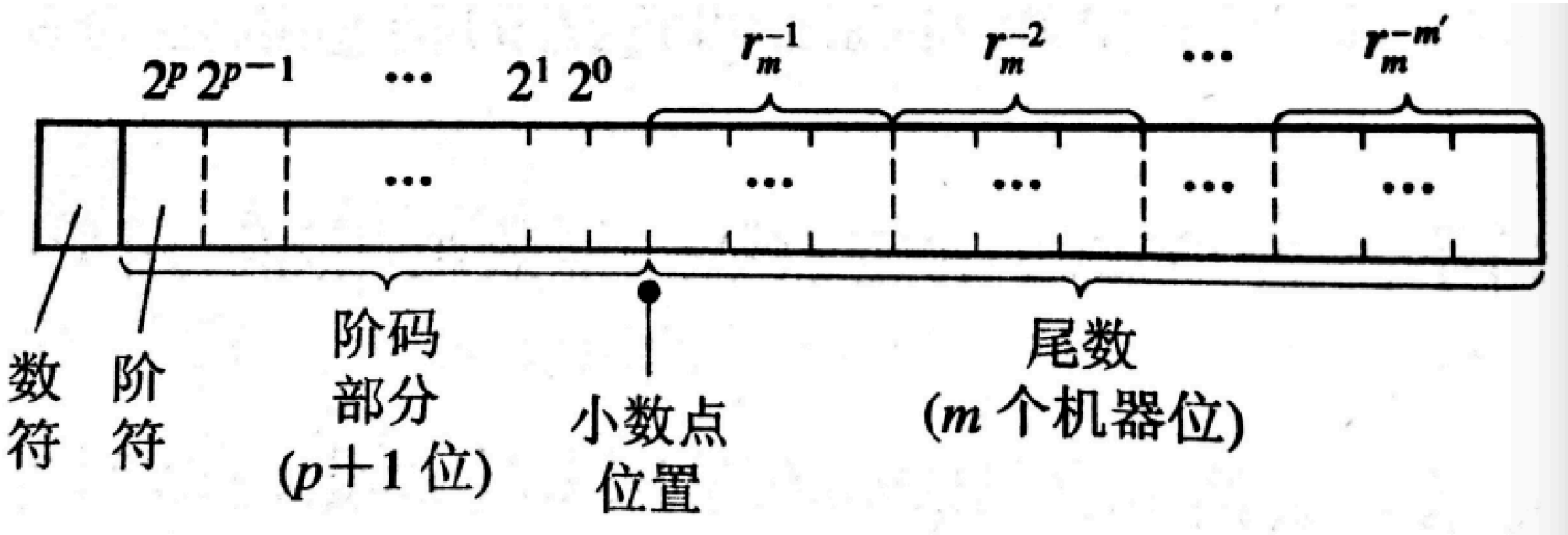
    有阵列型：1条指令，减少4*40000=160000字

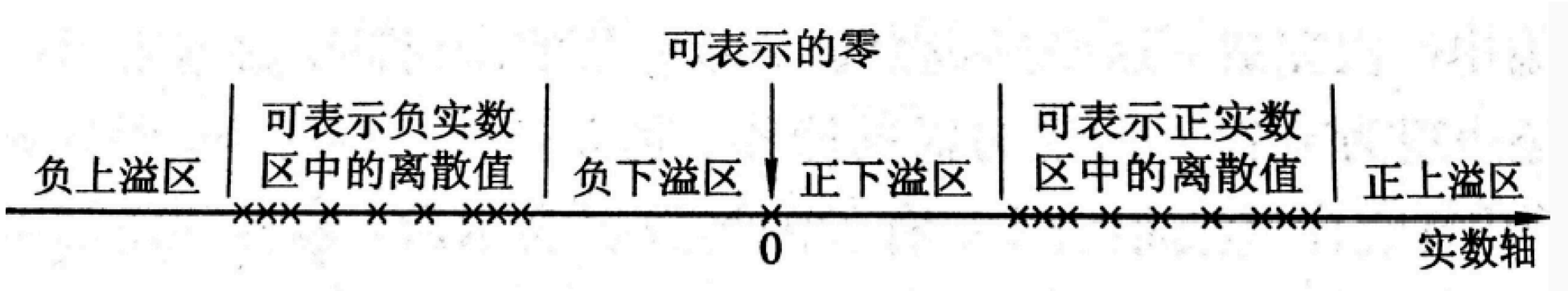
- 原则2：看引入该种数据表示之后，其通用性和利用率是否提高

浮点数据表示

- 定点数据表示的缺点
  - 编程困难
  - 表示数的范围小
  - 数据存储单元的利用率低
- 浮点数表示方式的核心：数据字长与这种数据表示方式的表数范围、表数精度和表数效率之间的关系
- 浮点数表示方式的关键：在数据字长已经确定的前提下，研究各种浮点数表示的表数范围、表数精度、表数效率及它们之间的关系等

- 浮点数阶值的位数p主要影响两个可表示区的大小，即表数范围
- 尾数的位数m主要影响在可表示区中能表示值的精度
- 由于尾数位数有限，不能对所有实数进行精确表示，只能用较为接近的可表示数进行近似表示，产生的误差大小就是数的表示精度





- 机器字长有限，只能表示正负区间上的有限的离散值
- 阶值位数p主要影响可表示正负区的大小，即可表示数的范围
- 尾数m主要影响在可表示值的精度
- 由于尾数位数有限，不能对所有实数进行精确表示，只能用较为接近的可表示数进行近似表示，产生的误差大小就是数的表示精度
- 阶码采用二进制，避免运算中因对阶造成的精度和有效数值的过多损失
- 问题：如何确定尾数的基值（即采用什么进制）

- r_m表示尾数的基
- 一个r_m进制的数位是用$\lceil \log_2 r_m \rceil$个机器位来表示
- 机器位数为m的时候，相当于r_m进制的尾数有$m' = m/\lceil \log_2 r_m \rceil$个数位

$$r_m^{-1}, r_m^{-2}, \ldots, r_m^{-m'}$$

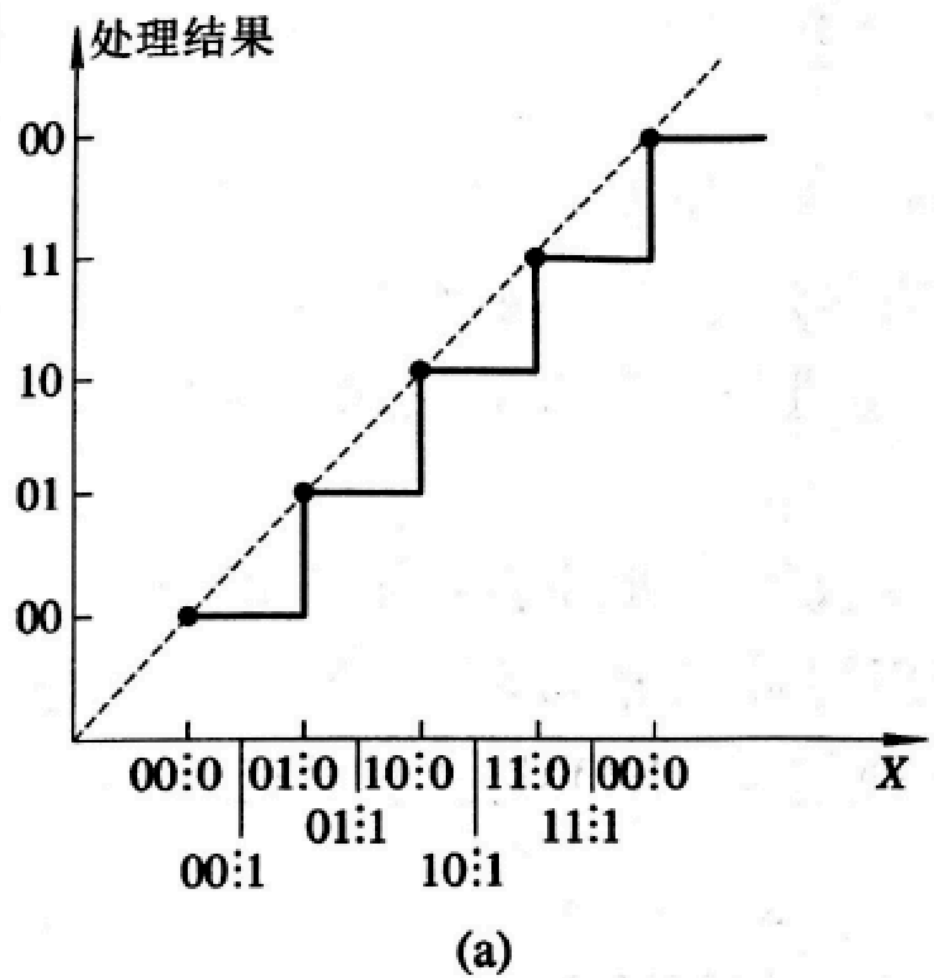
阶值：二进制p位，尾数：r_m进制m'位，非负阶、正尾数、规格化条件下浮点数的特性

- 最小尾数 $1\times r_m^{-1}$
- 最大尾数 $1-r_m^{-m'}$
- 最大阶值 2^p-1
- 可表示最小值 r_m^{-1}
- 可表示最大值 $r_m^{(2^p-1)}\times(1-r_m^{-m'})$
- 可表示尾数的个数 $r_m^{m'}(1-\frac{1}{r_m})$
- 可表示阶的个数 2^p
- 可表示数的个数 $2^p r_m^{m'}(1-\frac{1}{r_m})$

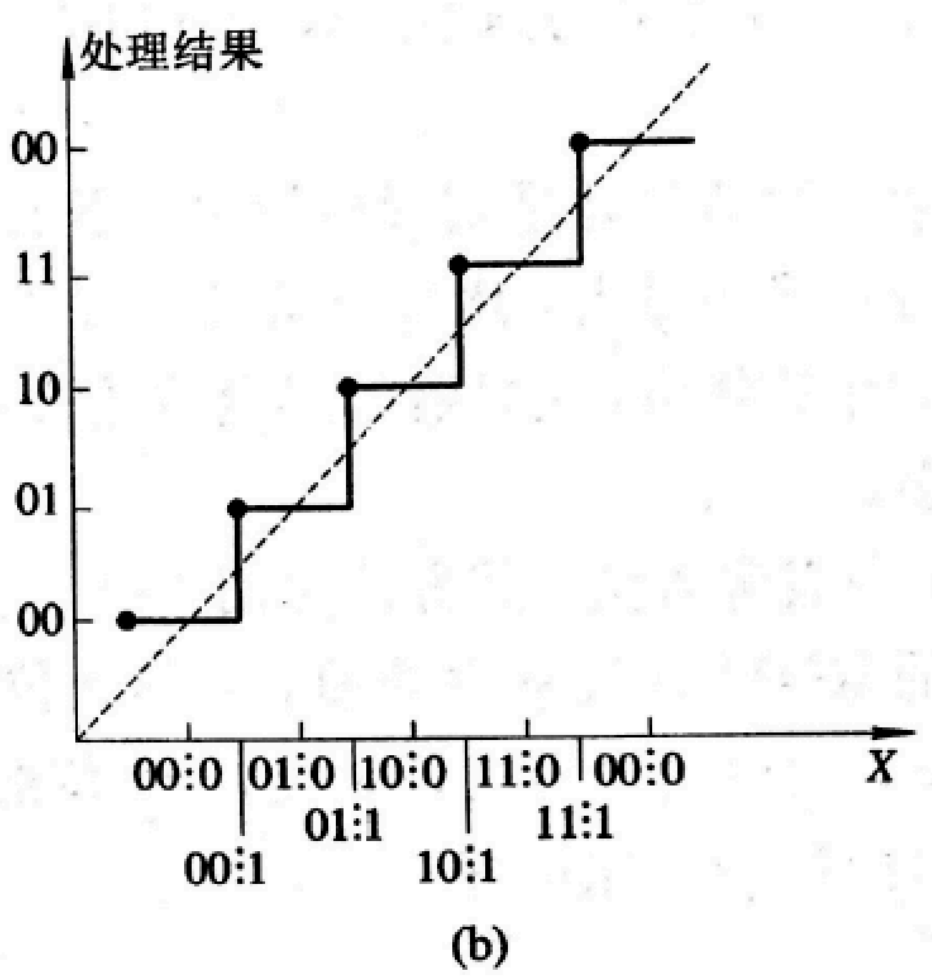
- 可表示数的范围：r_m越大，可表示数的范围越大
- 可表示数的个数：r_m越大，可表示数的个数越大
- 可表示精度：r_m越大，数的密度分布越稀，表示精度越低
- 运算中的精度损失：r_m越大，精度损失越小
- 运算速度：r_m越大，运算速度越快

尾数下溢处理方法

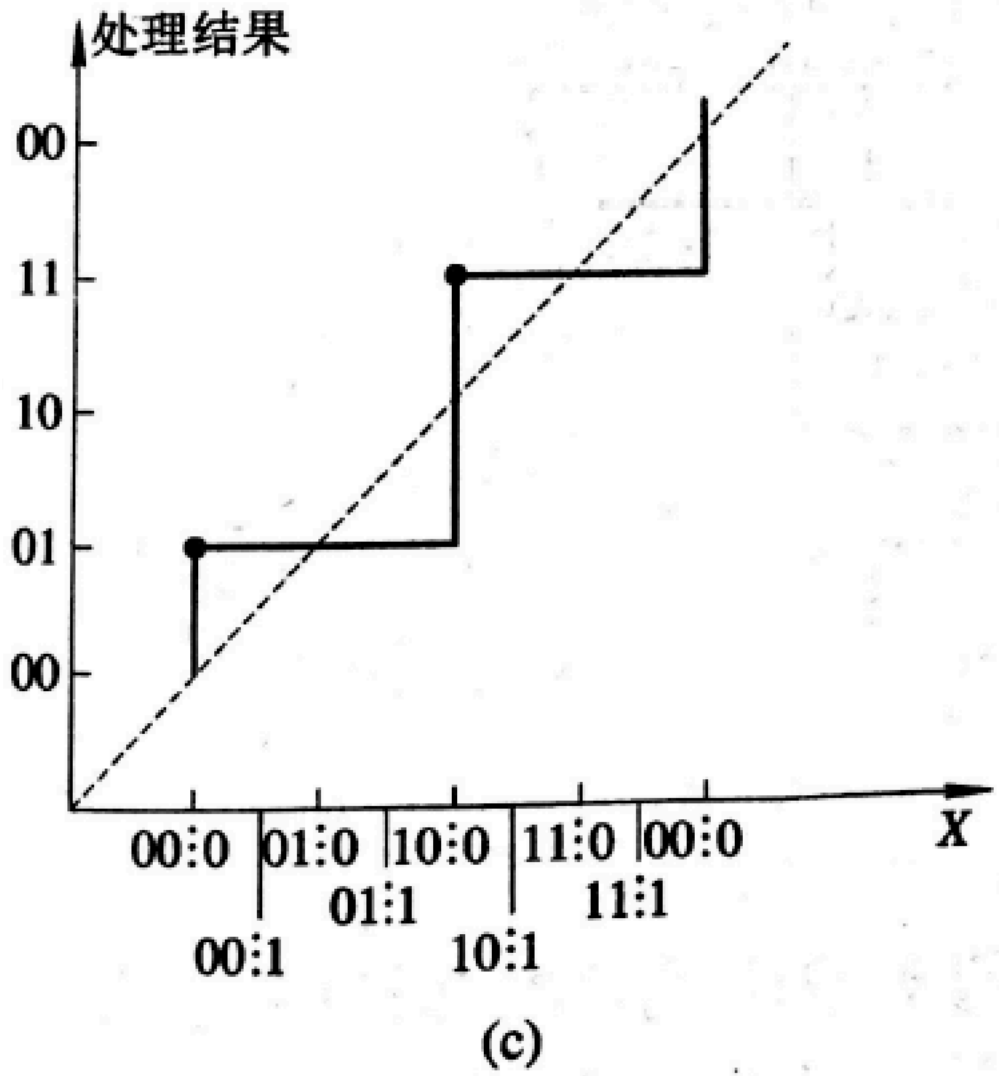
- 截断法
  - 将尾数超出字长的部分截去
  - 误差较大且无法调节，因而很少使用
- 舍入法
  - 在机器运算的规定字长之外设置附加位，存放溢出部分最高位
  - 最大误差小，平均误差接近于零
  - 处理速度慢
- 恒置“1”法
  - 将机器运算规定的字长的最低位恒置“1”
  - 实现简单、平均误差小
  - 最大误差大
- 查表舍入法
  - 在ROM或PLA（Programmable Logic Array）存放下溢处理表


处理结果
00
11
10
01
00
00:0
01:0
10:0
11:0
00:0
X
00:1
01:1
10:1
11:1

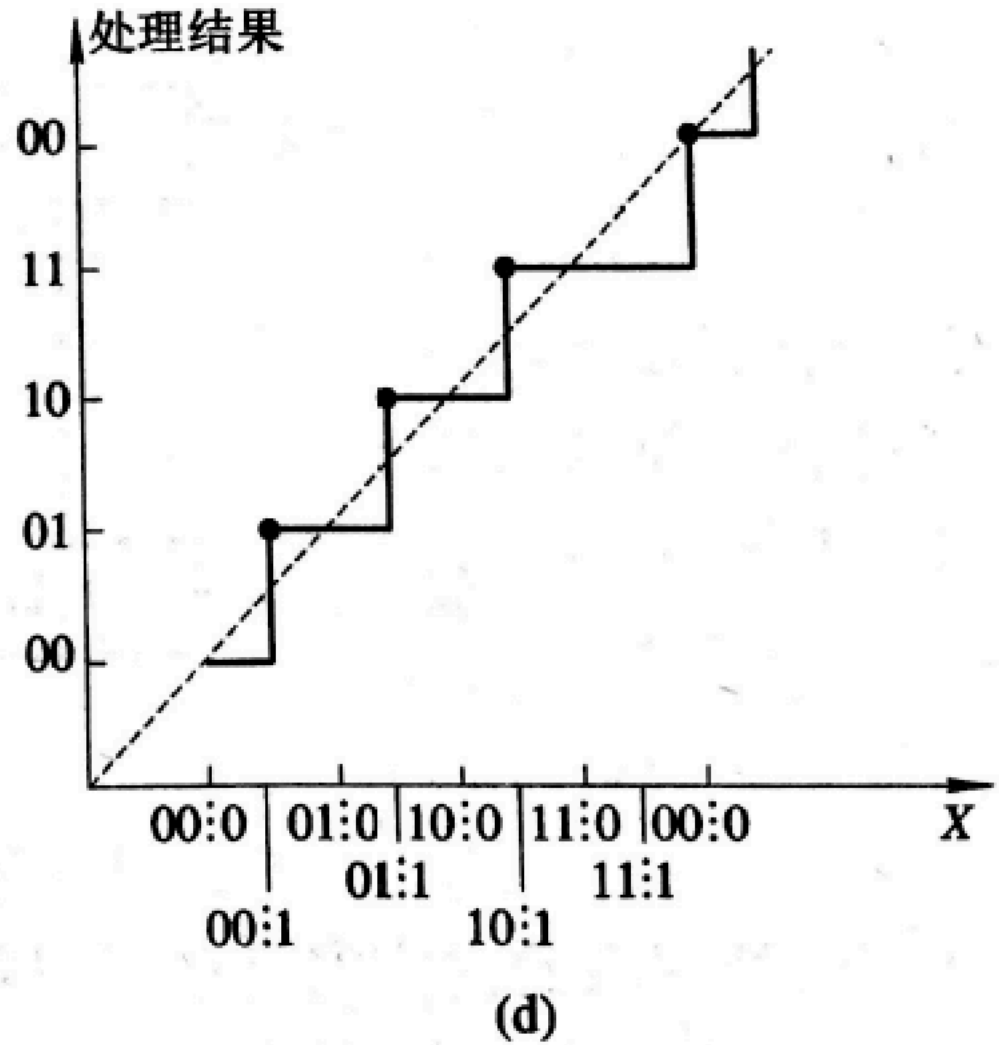
(a)


处理结果
00
11
10
01
00
00:0
01:0
10:0
11:0
00:0
X
00:1
01:1
10:1
11:1

(b)


处理结果
00
11
10
01
00
00:0
01:0
10:0
11:0
00:0
X
00:1
01:1
10:1
11:1

(c)


处理结果
00
11
10
01
00
00:0
01:0
10:0
11:0
00:0
X
00:1
01:1
10:1
11:1

(d)

寻址方式

- 寻址方式：是指令按什么方式寻找（访问）到所需的操作数或信息
- 三个面向
  - 面向主存：主要访问主存，少量访问寄存器
  - 面向寄存器：主要访问寄存器，少量访问主存和堆栈
  - 面向堆栈：主要访问堆栈，少量访问主存和寄存器
- 寻址方式在指令中如何指明
  - 占用操作码的某些位来指明
  - 在地址码部分专门设置寻址方式位字段指明

寻址方式的种类

- 寄存器寻址：指令中的地址码是寄存器的编号，而不是操作数地址或操作数本身
  - 寄存器直接寻址：寄存器内容即操作数本身
  - 寄存器间接寻址：寄存器内容是操作数的地址
- 立即寻址：指令的地址码字段指出的不是地址，而是操作数本身
- 直接寻址：指令的地址码字段中直接指出操作数在主存中的地址，即形式地址等于有效地址
- 间接寻址：指令地址码字段指向的存储单元中存储的不是操作数本身，而是操作数地址
- 相对寻址：把程序计数器PC的内容加上指令中的形式地址而形成操作数的有效地址，而程序计数器的内容就是当前指令的地址。

寻址方式的种类（续）

- 变址寻址：将变址寄存器的内容加上指令中的形式地址而形成操作数的有效地址，实现程序块的规律性变化
- 基址寻址：将基址寄存器的内容加上指令中的形式地址而形成操作数的有效地址，其优点是可以扩大寻址能力

逻辑地址与主存物理地址

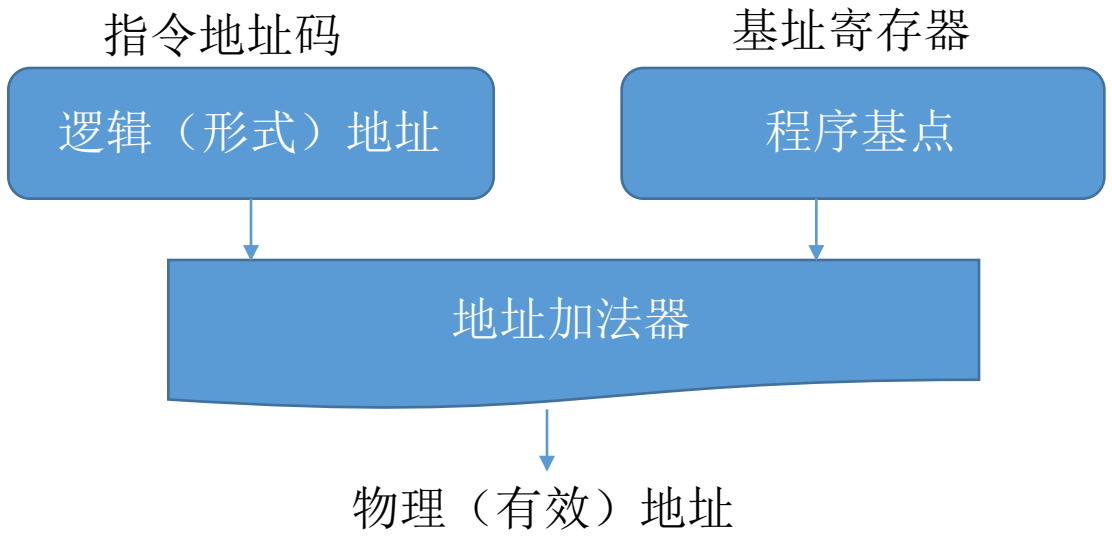
- 逻辑地址：程序员编写程序时使用的地址。
- 物理地址：程序在主存中的实际地址。

静态再定位

- 在目的程序装入主存的时候，由装入程序用软件方法把目的程序的逻辑地址变换为物理地址，程序执行时，物理地址不再改变
- 不利于多道程序执行

动态再定位

- 基址寻址
  - 增加相应的基址寄存器和地址加法器硬件，程序不做变换直接装入主存的同时，将装入主存的起始地址装入对应该道程序使用的基址寄存器中。
  - 在程序执行时，只要通过地址加法器将逻辑地址加上基址寄存器的程序基址形成有效物理地址后进行访存操作即可。
  - 需要通过上、下界寄存器进行地址越界判断。



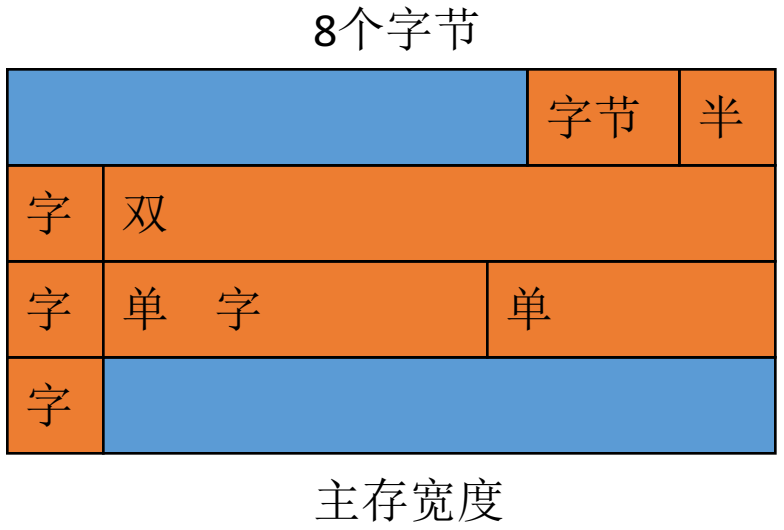
虚拟地址映像表

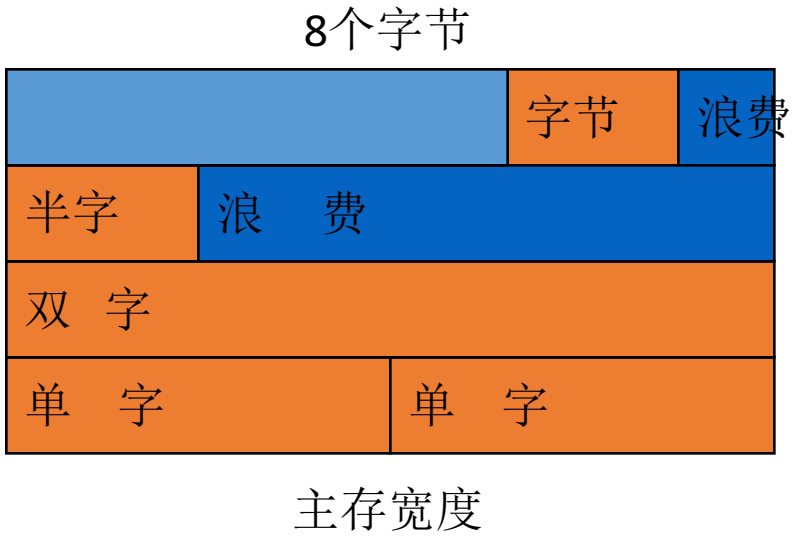
- 采用虚拟存储器，增加映像表硬件，使得程序空间可以超过实际主存空间

- 基址寻址
  - B为基址寄存器号，存放24位的基地址
  - $(B)_{8\sim31}$ + D 形成24位宽的访存物理地址

物理主存中的信息分布

- IBM 370
  - 信息宽度：字节（8位）、半字（双字节），单字（4字节），双字（8字节）
  - 主存宽度：8字节（64位）
- 任意存储
- 整数倍存储





指令系统的设计和优化

- 指令系统是从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计算机具有的基本功能
  - 指令系统是计算机系统结构的主要组成部分
  - 指令系统是软件与硬件分界面的一个主要标志
  - 指令系统是软件与硬件之间互相沟通的桥梁
- 指令系统的设计主要包括指令的功能（操作类型、寻址方式和具体操作内容）和指令格式的设计.

指令设计的步骤

- 根据应用，初拟出指令的分类和具体的指令；
- 试编出用该指令系统设计的各种高级语言的编译程序；
- 用大量测试程序进行模拟测试，看指令系统的操作码和寻址方式效能是否都比较高；
- 将程序中高频出现的指令串复合改成一条强功能新指令，即改用硬件方式实现；而将频度很低的指令的操作改成基本的指令组成的指令串来完成，即用软件方式实现；

指令类型

- 非特权型：主要供应应用程序员使用，也可供系统程序员使用，包括算术逻辑运算、数据传送、浮点运算、字符串、十进制运算、控制转移及系统控制等；
- 特权型：系统程序员使用，用户无权使用，包括启动I/O（多用户环境下）、停机等待、存储管理保护、控制系统状态、诊断等；

指令系统的设计（编译程序设计）

- 设计的原则：如何支持编译系统能高效、简易地将源程序翻译成目标代码。
  - 规整性：对相似的操作做相同的规定
  - 对称性
  - 独立性和全能性
  - 正交性
  - 可组合性
  - 可扩充性

指令系统的设计（系统设计）

- 指令码密度适中
  - 高密度指令：强功能复合指令，可代替功能强的指令串功能
  - 优点：减少程序长度、访存次数、Cache、虚存访问调度次数、程序运行时间；
  - 缺点：指令系统复杂，硬件实现困难；
- 兼容性
  - 对系列机而言，为保证向后兼容，只能增加指令，不能删除或者更改原有功能
- 适应性
  - 当工艺技术发展变化后，指令系统仍可以方便地用硬件来实现

指令格式的优化

- 指令=操作码+地址码
- 指令格式的优化：如何用最短的位数来表示指令的操作信息和地址信息，使程序指令的平均字长最短
- 操作码的优化：缩短指令字长，减少程序总位数及增加指令字能表示的操作信息和地址信息。

操作码的优化

- 操作码I_i的使用频度p_i
- 操作码的信息源熵（信息源所含的平均信息量）

$$H = -\sum_i p_i \log p_i$$

- 信息冗余度

$$\frac{\text{实际平均码长} - H}{\text{实际平均码长}}$$

举例

- $n = 7$条指令，定长码表示，需要$\lceil \log n \rceil = 3$位

指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
频度	0.4	0.3	0.15	0.05	0.04	0.03	0.03

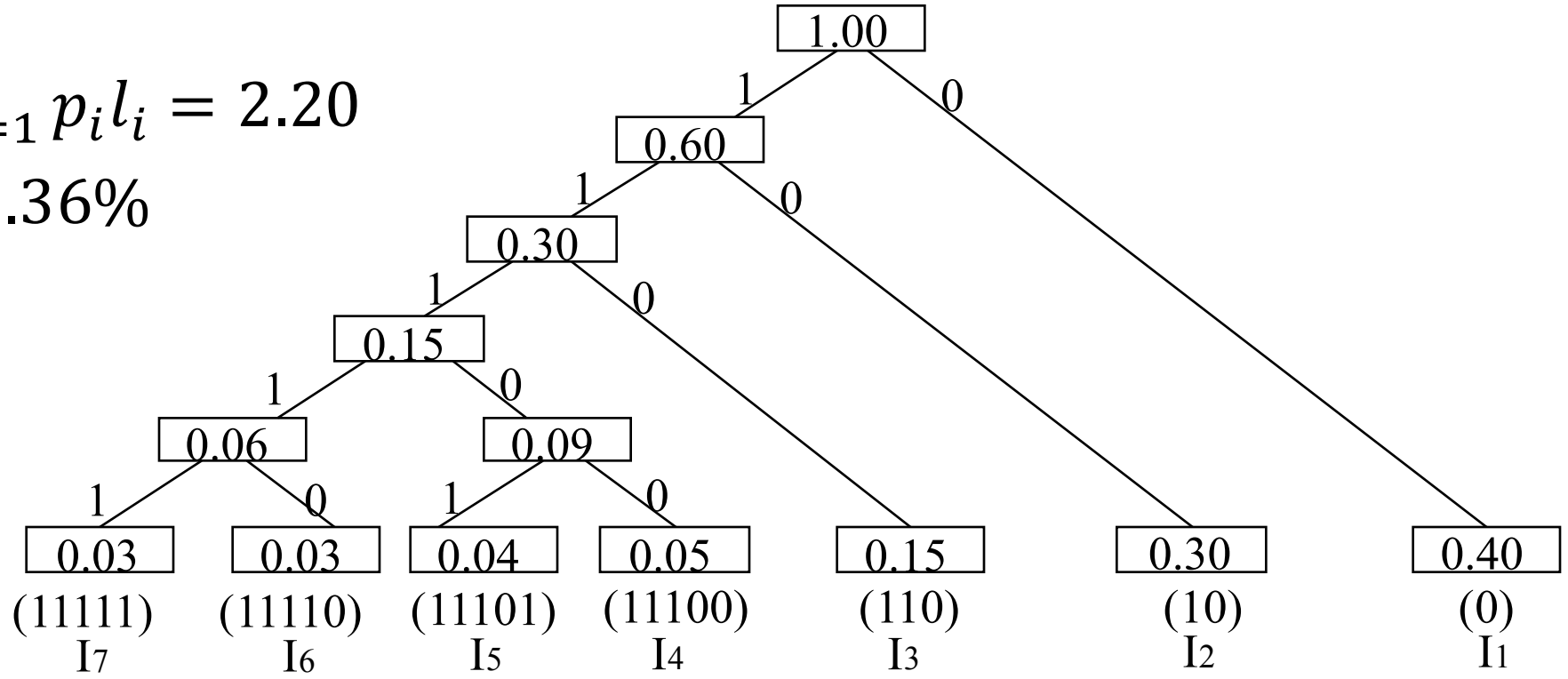
- $H = -\sum_{i=1}^{7} p_i \log p_i$=2.17
- 冗余度= $\frac{3-2.17}{3} \approx 28\%$

哈夫曼（Huffman）压缩

- 哈夫曼（Huffman）压缩：当各种事件发生的概率不均等时，采用优化技术，对发生概率高的事件用最短的位数（时间）来表示（处理），而对出现概率较低的事件允许使用较长的位数（时间）来表示（处理），以缩短表示（处理）的平均位数（时间）。
- 用于代码压缩、程序压缩、空间压缩和时间压缩
- 哈夫曼编码不唯一，但平均码长唯一。

哈夫曼编码

- 平均码长$=\sum_{i=1}^{7} p_i l_i = 2.20$
- 信息冗余量$=1.36\%$
- 长度个数$=4$



指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
频度	0.4	0.3	0.15	0.05	0.04	0.03	0.03

基于哈夫曼编码的扩展操作码编码

- 完全哈夫曼编码码长种类太多，不利于译码和实现
- 界于定长二进制编码和完全哈夫曼编码之间的一种编码方式
- 操作码长度不定，但只有有限种选择
- 高概率用短码，低概率用长码
- 短码不能是长码的前缀
- 缩短操作码的平均长度，以降低信息冗余度

举例

指令	Huffman编码	扩展操作码编码
I_1	0	00
I_2	10	01
I_3	110	10
I_4	11100	1100
I_5	11101	1101
I_6	11110	1110
I_7	11111	1111

平均码长=2.30

信息冗余量=0.0565=5.65%

操作码等长扩展编码法

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 16 种

等长 15/15/15……扩展法

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

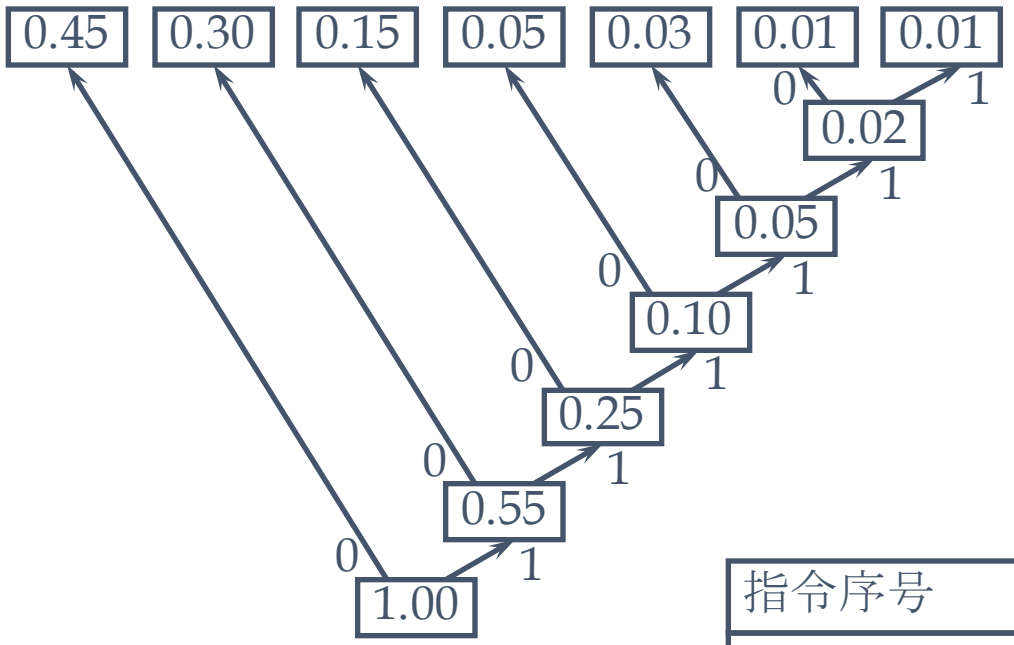
等长 8/64/512……扩展法

举例

- 假设一台模型计算机共有7种不同的操作码。已知各种操作码在程序中出现的概率如下表，计算采用Huffman编码法的操作码平均长度，并计算Huffman操作码的信息冗余量。

  利用Huffman树进行操作码编码的方法，又称为最小概率合并法。

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



操作码平均长度＝1.97
信息熵 ＝ 1.95

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

指令	概率	1-2-3-5扩展编码	2-4扩展编码
1	0.45	0	00
2	0.30	10	01
3	0.15	110	10
4	0.05	11100	1100
5	0.03	11101	1101
6	0.01	11110	1110
7	0.01	11111	1111
平均长度		2.0	2.2
信息冗余量		2.5%	11.4%

- 为便于实现分级译码，一般采用等长扩展法
  - 4-8-12扩展法，每次扩展4位
  - 3-6-9扩展法，每次扩展3位
- 对于等长扩展法，根据采用不同的扩展标志，还可以有多种不同的扩展方法
  - 对于4-8-12扩展法，可以采用保留一个码点标志的15/15/15……扩展法，也可以采用每次保留一个标志位的8/64/512……扩展法

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15/15/15……扩展法

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

等长8/16/512……扩展法

4-6-10不等长扩展编码法

编码方法	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

举例

- 三地址指令4条

- 单地址指令255条

- 零地址指令16条

- 指令字长12位，每个地址码长3位
- 能否以扩展操作码为其编码

000 三地址操作码

001 三地址操作码

010 三地址操作码

011 三地址操作码

100 拓展标志 $+2^6$

101 拓展标志 $+2^6$

110 拓展标志 $+2^6$

111 拓展标志 $+2^6$

$4\times2^6=256$

255条单地址操作码

1个拓展码 → $2^3=8$ 条零地址操作码

- 如果单地址指令改为254条？

000 三地址操作码

001 三地址操作码

010 三地址操作码

011 三地址操作码

100 拓展标志 $+2^6$

101 拓展标志 $+2^6$

110 拓展标志 $+2^6$

111 拓展标志 $+2^6$

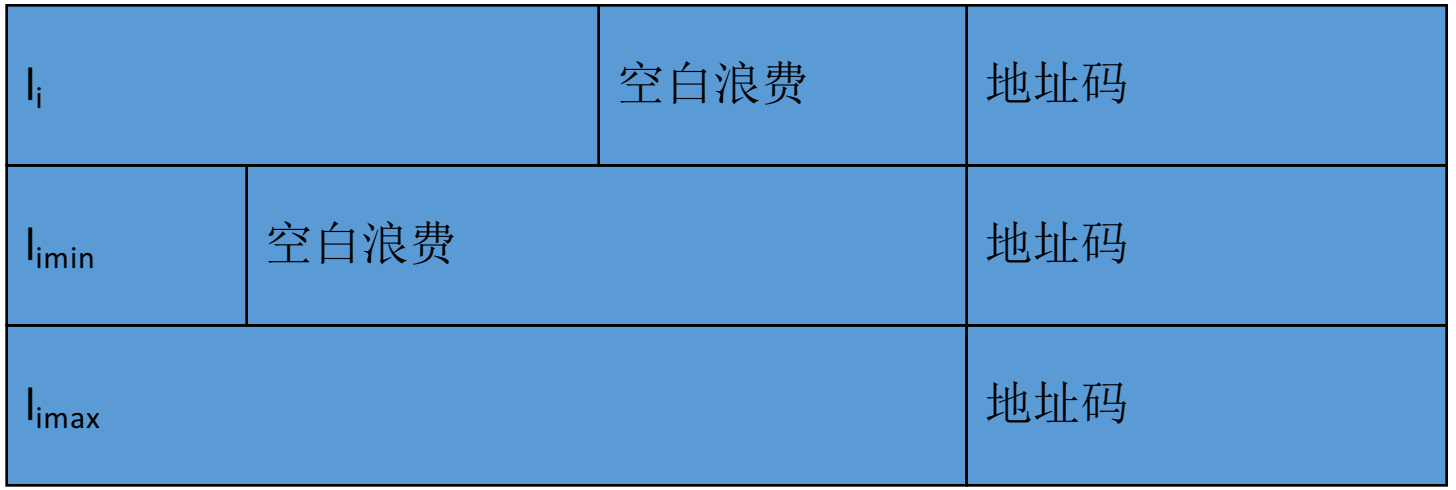
$4\times2^6=256$

254条单地址操作码

2个拓展码 → $2\times2^3=16$ 条零地址操作码

指令字格式的优化

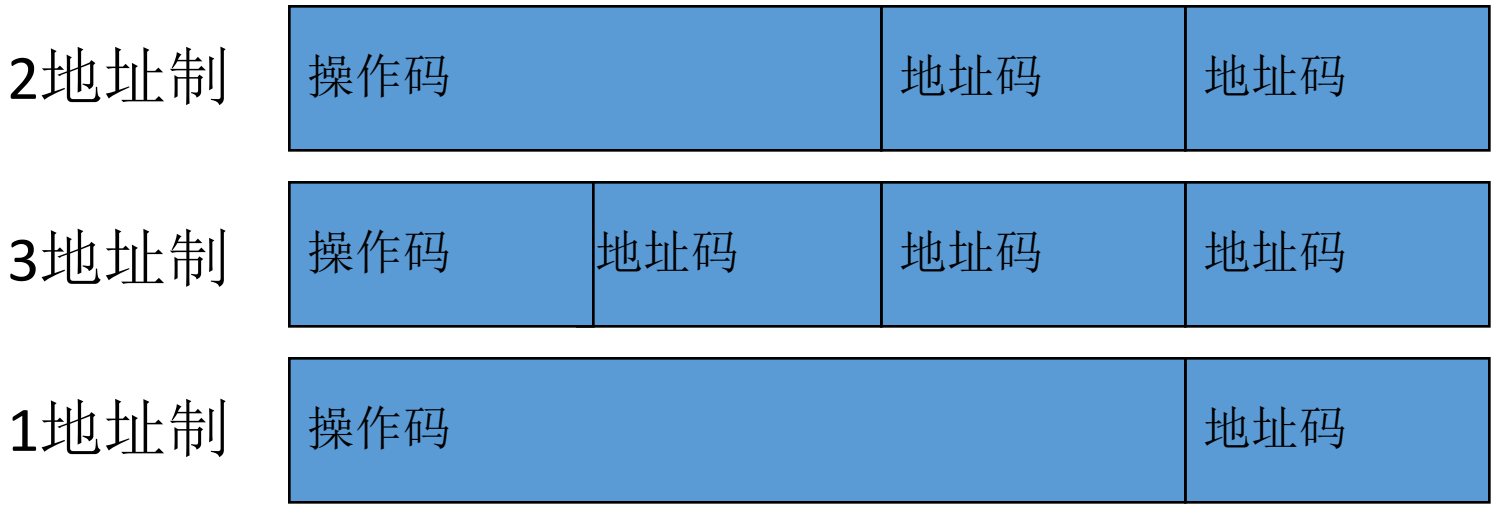
- 主存按位编址，指令字不按整数边界存储，而是逐条紧挨存储
  - 程序总位数减少
  - 访存指令速度下降
- 指令字按整数边界存储



指令字格式的优化

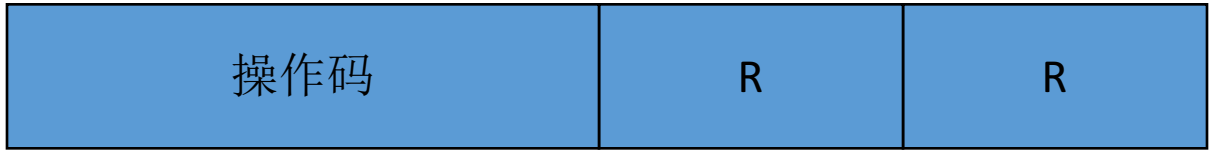
- 扩大操作数地址的寻址范围
  - 采用基址寄存器寻址、相对寻址或寄存器寻址
  - 基于分段的编址 （段号＋段内地址）
- 操作数的地址码长度可以有很宽的 变化范围，可以与可变长操作码配合，可以显著减少存储空间的浪费
- 多种地址制，同一种地址制还可以采用多种地址形式和长度，也可以直接用空白处来存放直接操作数或常数

在定长指令字内实现多种地址制



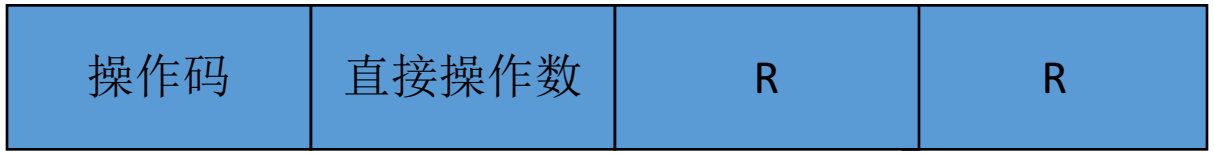
同种地址制下的多种地址形式和长度

寄存器－寄存器型



寄存器－存储器型

带直接操作数



指令字格式优化的措施

- 采用扩展操作码，根据指令的频度的分布选择合适的编码方式，以缩短操作码的平均码长
- 采用多种寻址方式，以缩短地址码长度，并在有限地址长度内提供更多地址信息
- 采用多种地址制，增强指令功能，从宏观上缩短程序长度，加快程序执行速度
- 在同种地址制内再采用多种地址形式，让每种地址字段可以有多种长度，且让长操作码与短地址码进行组配
- 在维持指令字在存储器中按整数边界存储的前提下，使用多种不同的指令字长

- 某模型机9条指令的使用频度如下。要求有两种指令字长，都按双操作数指令格式编排，采用扩展操作码，并限制只能有两种操作码码长。设该机有若干通用寄存器，主存16位宽度，按字节编址，采用按整数边界存储，任何指令都可以在一个指令周期中取得，短指令为寄存器—寄存器型，长指令为寄存器—主存型，主存地址应能变址寻址。

指令	频度	指令	频度	指令	频度
ADD 加	30%	SUB 减	24%	JOM 按负转移	6%
STO 存	7%	JMP 转移	7%	SHR 右移	2%
CIL 循环左移	3%	CLA 清加	20%	STP 停机	1%

- 完全哈夫曼编码 （平均长度 $\sum_{i=1}^{9} p_i l_i = 2.61$）

指令	频度	哈夫曼编码	指令	频度	哈夫曼编码	指令	频度	哈夫曼编码
ADD 加	30%	01	JOM 按负转移	6%	0001	SHR 右移	2%	000001
SUB 减	24%	11	STO 存	7%	0011	CIL 循环左移	3%	00001
CLA 请加	20%	10	JMP 转移	7%	0010	STP 停机	1%	000000

- 扩展操作码（平均长度 $\sum_{i=1}^{9} p_i l_i = 0.74\times2 + 0.26\times5 = 2.78$）

指令	频度	扩展编码	指令	频度	扩展编码	指令	频度	扩展编码
ADD 加	30%	00	JOM 按负转移	6%	11000	SHR 右移	2%	11011
SUB 减	24%	01	STO 存	7%	11001	CIL 循环左移	3%	11100
CLA 请加	20%	10	JMP 转移	7%	11010	STP 停机	1%	11101

该机允许使用的的通用寄存器个数为$2^3 = 8$

- 短指令格式

操作码（2位）　寄存器1（3位）　寄存器2（3位）

- 长指令格式

操作码（5位）　寄存器1（3位）　变址寄存器（3位）　相对位移（5位）

指令系统的发展和改进

- CISC和RISC
- 按CISC方向发展和改进指令系统
- 按RISC方向发展和改进指令系统

指令系统的优化设计

- 优化指令系统设计的**3**个阶段：
  - **CISC**：复杂指令系统**60**年代至**70**年代中期
  - **RISC**：精简指令系统**70**年代后期至现在
  - **VLIW**：**80**年代初期至现在
- 关键在软硬件的功能分配，系统的综合性能时间与空间；执行、编译、编写时间

复杂指令系统计算机

- Complex Instruction Set Computer CISC
- 增强原有指令的功能，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。
- 原因
  - 当高级语言（如C语言）取代汇编语言后，就不断增加新的复杂指令来支持高级语言程序的高效实现；
  - 由于访主存的速度显著低于访CPU寄存器的速度，因此在功能相同时，不断用一条功能复杂的新指令来取代原先需一连串指令完成的功能，将程序软件固化和硬化；
  - 系列机软件要求向上兼容和向后兼容，使得指令系统不断扩大和增加；

复杂指令系统计算机

- 方法：用一条指令代替一串指令
  - 增加新的指令
  - 增强指令功能，设置功能复杂的指令
  - 增加寻址方式
  - 增加数据表示方式
- 按CISC方向发展和改进指令系统
  - 面向目标程序的优化实现改进
  - 面向高级语言的优化实现改进
  - 面向操作系统的优化实现改进

面向目标程序的优化实现来改进1

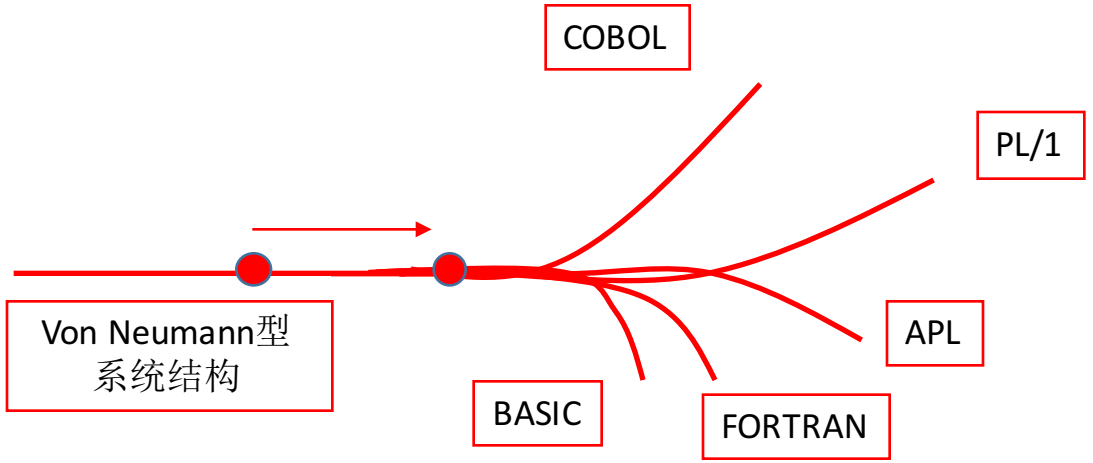
- 对使用频度高的指令增强其功能
  - 静态使用频度---减少存储空间
  - 动态使用频度---减少执行时间
- 对高频指令可增强其功能，加快执行速度，缩短字长
- 对低频指令考虑将其整合到高频指令中，或在将来将其取消
- 对高频指令串可增设新指令取代，减少存取指令的次数，加快目标程序执行，缩短目标程序长度

面向目标程序的优化实现来改进2

- 增设强功能复合指令来取代原先由常用宏指令或子程序实现的功能
- 由微程序实现可以大大提高运算速度，减少程序调用额外开销，减少子程序所占的主存空间
- 实质是尽量减少程序中如存、取、传送、转移、比较等不执行数据变换的非功能型指令的使用，让真正执行数据变换的加、减、乘、除、与、或等功能型指令所占的比例提高

面向高级语言的优化实现来改进1

- 缩短高级语言和机器语言的语义差距，增加对高级语言和编译系统支持的指令功能，缩短编译程序长度和编译时间
- 对源程序中各种高级语言语句的使用频度进行统计，对高频高级语言增设与之语义差距较小的新指令。很难做到对多种高级语言都是 优化的，只能面向用户所用的语言。
- 面向编译，优化代码生成，增强系统结构的规整性，实现存储单元的均匀利用。

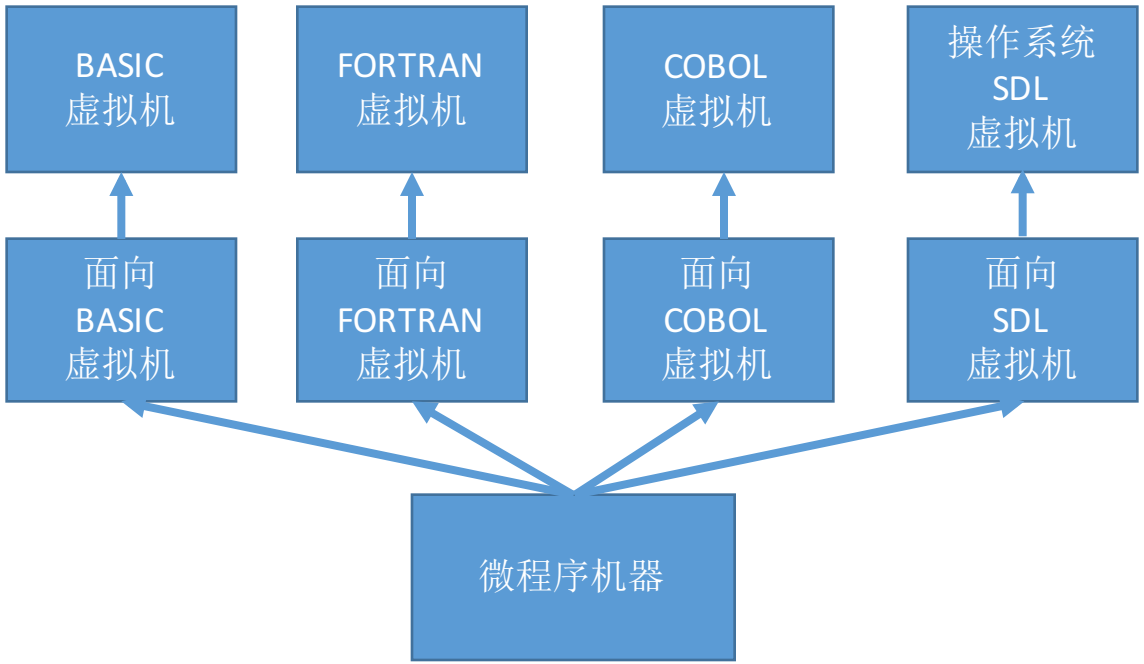


面向高级语言的优化实现来改进2

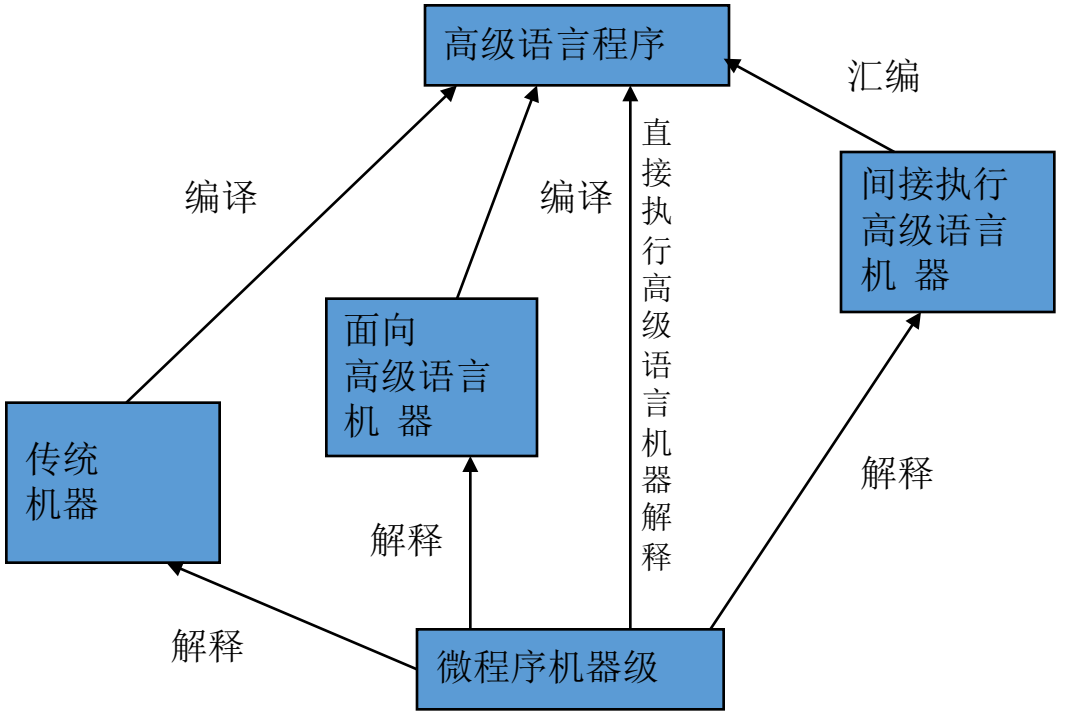
- 改进指令系统，使它与各种语言间的语义差距都有同等的缩小
  - 例如IBM370通过设置“小于等于转移”之类的复合指令，对加快FORTRAN DO、ALGOL DO和COBOL LOOP的实现有好处，同等程度地缩短了系统结构与各种语言之间的语义差距

面向高级语言的优化实现来改进3

- **多指令、多系统结构**的面向问题的动态自寻优系统
  - 以高级语言为主，指令系统为从
  - 各种高级语言各自对应一套由微程序解释的面向这种语言的机器指令系统和数据表示格式，工作前由操作系统根据所用的高级语言切换控制存储器中所对应的微程序



- 编译主要采用翻译技术，在微程序控制的机器上，机器语言用解释实现
- 面向编译通过缩小语义差距改进指令系统的思路实际上意味着增大解释的比重，减少翻译的比重



各种机器的语义差距

面向高级语言的优化实现来改进4

- 发展高级语言计算机
  - 间接执行的高级语言机器：让高级语言直接成为机器的汇编语言，通过汇编（用软件或者硬件实现）把高级语言源程序翻译成机器语言目标程序
  - 直接执行的高级语言机器：让高级语言本身作为机器语言，由硬件或固件逐条解释执行，既不用编译，也不用汇编
  - 高级语言种类繁多，很难全部涵盖，性价比低，不是所有高级语言都可以用解释高效实现

面向操作系统的优化实现来改进

- 通过缩短操作系统与计算机系统结构之间的语义差距，来进一步减少运行操作系统的时间和节省操作系统软件所占用的存储空间。
- 途径1：通过对操作系统中常用的指令和指令串的使用频度进行统计和分析来改进
- 途径2：考虑如何增设专用于操作系统的新指令
  - 在IBM 360系统上对多个进程使用公用区的管理提供“测试与置定”指令，使一个进程必须同时占有标志位和CPU两个资源
  - 在IBM 370系统上增设“比较与交换”指令，既避免死锁，又保证多个进程对公用区的正确使用

面向操作系统的优化实现来改进

- 途径3：把操作系统中频繁使用的，对速度影响较大的机构型软件子程序硬化或者固化，直接用硬件或微程序解释实现
  - 在尽量缩小语义差距的前提下，充分发挥软、硬件的各自特长
  - 硬件实现用于提高系统的执行速度和效率，减少操作系统的时间开销；软件实现用于提供系统应有的灵活性
  - 宜于硬（固）化实现的是机构型功能，而不是策略型功能
    - 机构型功能：基本的、通用的功能，如进程管理、信息保护和存储管理等
    - 策略型功能：随不同环境而异，允许用户修改，如作业排队、用户标识和资源管理等
- 途径4：发展让操作系统由专门的处理机来执行的功能分布处理系统结构

CISC的主要问题

- 指令系统庞大，指令功能异常复杂，导致VLSI设计困难，不利于自动化设计，延长设计周期，增大设计成本，降低系统可靠性。
- 许多指令的操作繁杂，执行速度很低，甚至不如简单基本指令组合实现
- 由于指令系统庞大，高级语言编译程序选择目标指令范围太大，因此难以优化生成高效机器语言程序，编译程序也太长、太复杂。
- 指令系统利用率低，增加了设计负担，降低了系统性价比。

精简指令系统计算机

- Reduced Instruction Set Computer RISC
  - 减少指令总数和简化指令的功能，降低硬件设计的复杂性，提高指令的执行速度。
  - 只保留功能简单的指令
  - 功能较复杂的指令用软件实现
  - 提高流水线效率

RISC的基本原则

- 确定指令系统时，只选择使用频度高的指令，再增加少量能有效支持操作系统、高级语言实现及其他功能的指令，大大减少指令 条数。
- 减少指令系统所用的寻址方式种类，一半不超过两种。
- 限制精简指令格式，使用长度相同的指令
- 让所有指令在一个周期内完成
- 扩大通用寄存器个数（≥ 32），尽量减少访存，所有指令只有存 / 取指令可以访存，其他指令一律只对寄存器操作
- 提高指令执行速度，大多数指令都用硬联控制实现，少数指令才用微程序实现
- 通过精简指令和优化设计编译程序，简单有效支持高级语言实现

RISC的基本技术

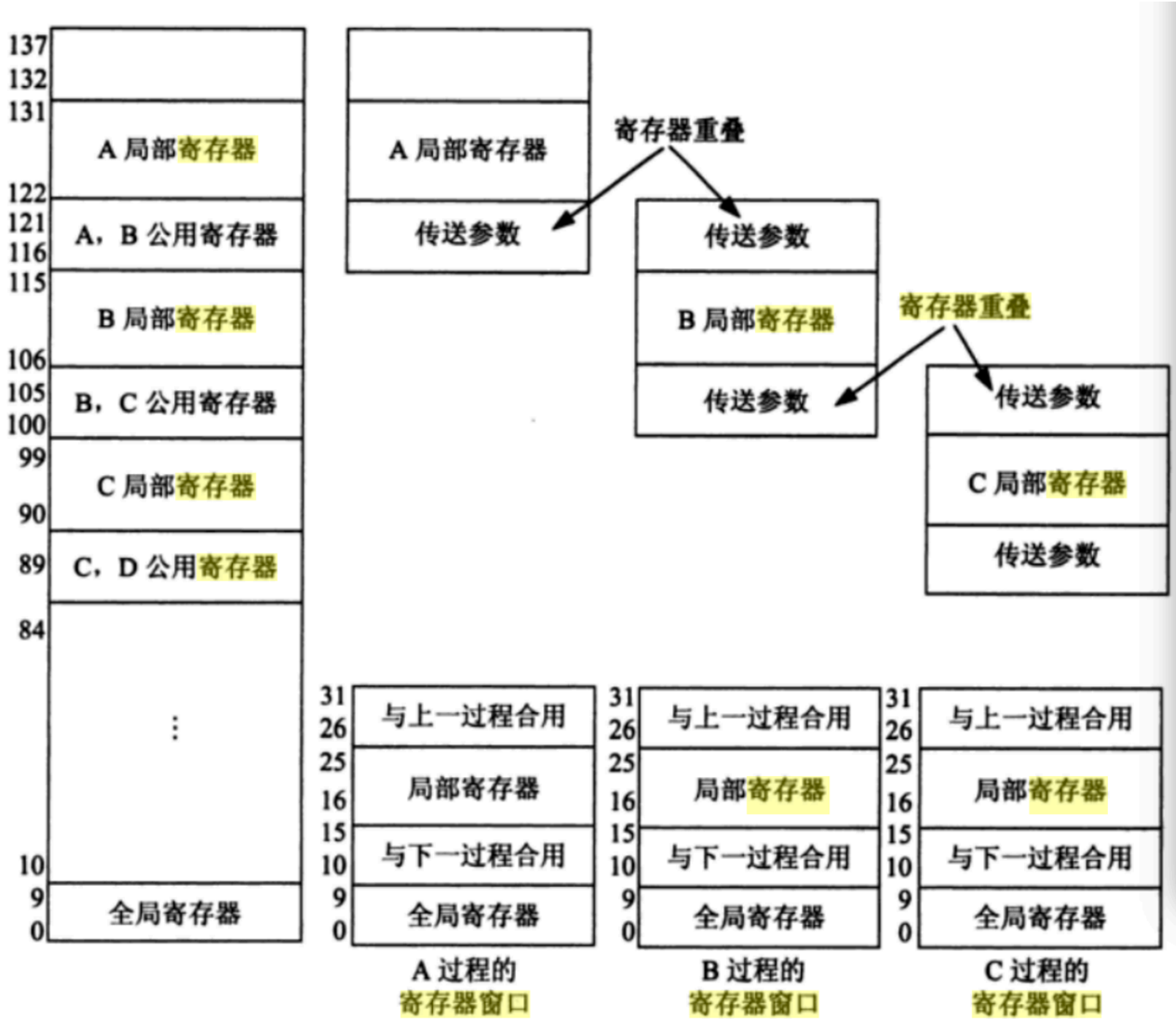
- 按照一般原则来设计
  - 选取使用频度高的最常用的基本指令，并增设一些对操作系统、高级语言、应用环境等支持最有用的指令，精简指令数
  - 使指令功能、格式和编码设计上尽可能简化、规整
  - 使指令尽可能等长，简化寻址方式，指令的执行尽可能在一个机器周期内完成
- 逻辑实现采用硬联和微程序相结合
  - 简单指令用硬联方式实现，复杂功能指令用微程序解释实现
- 在CPU中设置大量工作寄存器并采用重叠寄存器窗口
  - 尽量让指令操作在寄存器间执行，以提高执行速度，缩短指令周期，简化寻址方式和指令格式
  - 有效支持高级语言中的过程调用
  - 简单直接实现过程间的参数传递

RISC的基本技术

- 指令用流水和延迟转移
  - 流水：指令取出和执行的重叠
  - 延迟转移：将转移指令与前面的一条指令或多条指令兑换一下位置，让成功转移总是在紧跟的指令被执行之后发生，从而使预取指令不必作废，可以节省一个机器周期
- 采用高速缓冲存储器Cache
- 优化设计编译系统
  - 使用大量寄存器，优化寄存器的分配和使用，提高效率，减少访存次数。
  - 减少局部变量和工作变量的中间传递。
  - 优化调整指令的执行次序，减少机器的空等时间

重叠寄存器窗口

- 在RISC结构中，为了减少过程调用中保存现场和建立新现场，以及返回时恢复现场等辅助操作，通常将所有寄存器分成若干个组，称为寄存器窗口。每组中有若干个寄存器，每当有过程调用时，就分配一个未被使用的寄存器窗口，这样就可减少保存和恢复现场的开销。此外在每个寄存器窗口中，又分成大小固定的高区、本地和低区三个区段。其中本地区用来存放局部变量，高区在被调用时用来保存调用过程送来的参数，而在返回主调用过程时，存放返回结果。而低区在调用时存放欲送往被调用过程的参数，而在被调用过程返回时用来存放返回结果。在使用时，每一对调用和被调用过程的寄存器窗口各自的低区和高区相互重叠。一旦发生过程调用或返回，在控制由一个窗口转换到另一窗口时，这些参数就通过两个窗口间的公共寄存器区自动的被传送而不需要再用额外的传送时间


137
132
131
A 局部寄存器
122
121
A，B 公用寄存器
116
115
B 局部寄存器
106
105
B，C 公用寄存器
100
99
C 局部寄存器
90
89
C，D 公用寄存器
84
10
9
0
全局寄存器
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
31
26
25
16
15
10
9
0
与上一过程合用
局部寄存器
与下一过程合用
全局寄存器
A 过程的
寄存器窗口
B 过程的
寄存器窗口
C 过程的
寄存器窗口

延迟转移技术

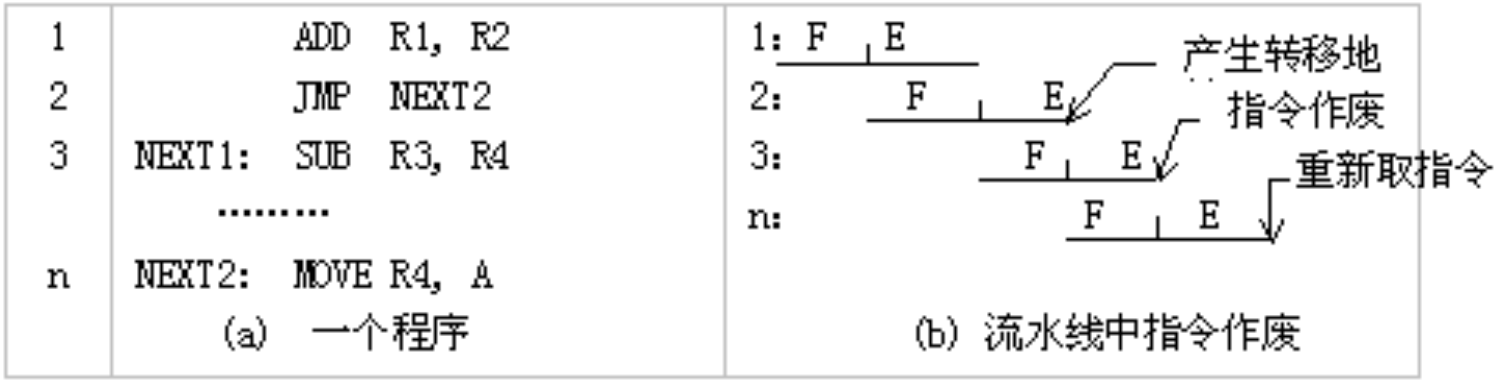


图 2.14 因转移指令引起的流水线断流

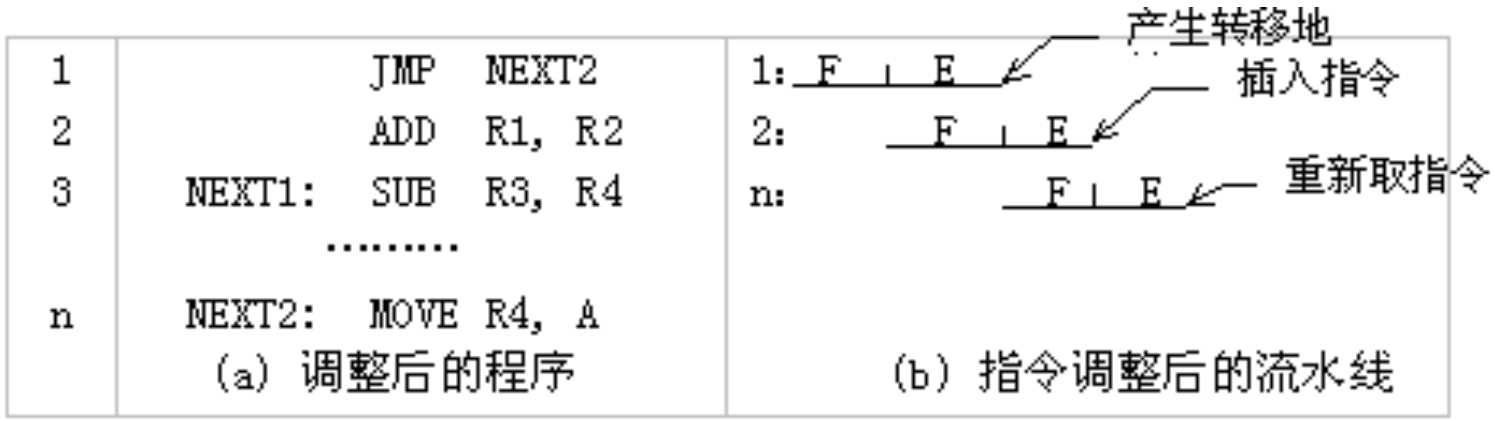


图 2.15 采用延时转移技术的指令流水线

- RISC的优势
  - 简化指令系统设计，适合VLSI的实现
  - 提高机器执行速度和效率
  - 降低设计成本，提高系统可靠性
  - 可直接支持高级语言实现，简化编译程序设计
- RISC的不足
  - 指令少，需要多条指令才能完成一项复杂功能，加重汇编语言程序设计负担，增加机器语言程序长度，占用存储空间多，加大了指令信息流
  - 对浮点运算的执行和虚拟存储器的支持不足
  - RISC机器的编译程序比CISC的难写
- RISC与CISC相结合