第2章 指 令 系 统

肖梦白
xiaomb@sdu.edu.cn
https://xiaomengbai.github.io

2.1 数据表示

2.2 寻址技术

2.3 指令系统的优化设计

2.4 指令系统的发展和改进

- 在机器上直接运行的程序是由指令组成的。
- 指令系统是软件与硬件之间的一个主要分界面，也是他们之间互相沟通的一座桥梁。
- 硬件设计人员采用各种手段实现指令系统，而软件设计人员则使用这些指令系统编制系统软件和应用软件，用这些软件来填补指令系统与人们习惯的使用方式之间的语义差距。
- 指令系统设计必须由软件设计人员和硬件设计人员共同来完成。
- 指令系统发展相当缓慢，需要用软件来填补的东西也就越来越多。

2.1 数据表示

2.1.1　数据表示与数据类型

2.1.2　高级数据表示

2.1.3　引入数据表示的原则

2.1.4　浮点数的表示方法

2.1.1 数据表示与数据类型

- **数据表示**：是指计算机硬件能够直接识别，可以被指令系统直接调用的那些数据类型
- **数据结构**：串、队、栈、向量、阵列、链表、树、图等软件要处理的各种数据结构，反映了如何组织数据元素
- 数据结构要通过软件映像，变换成机器所具有的数据表示来实现
- 不同的数据表示可为数据结构的实现提供不同的支持
- 数据结构和数据表示实际上是**软硬件的交界面**，需要在系统结构设计时予以确定

2.1.1 数据表示与数据类型

- 确定哪些数据类型用数据表示来实现的原则
  - 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
- 随着计算机系统的发展，数据表示也在不断上移，一些复杂的数据表示（如图、表等）也在某些计算机系统中出现

2.1.2 高级数据表示

1 自定义（Self-defining）数据表示

- **标志符数据表示**
  - 一般的计算机中，数据存储单元(寄存器、主存储器、外存储器等)只存放纯数据，数据的属性通过指令中的操作码来解释：
    - 数据的类型，如定点、浮点、字符、字符串、逻辑数、向量等；
    - 进位制，如2进制、10进制、16进制等；
    - 数据字长，如字、半字、双字、字节等；
    - 数据的功能，如地址、地址偏移量、数值、控制字、标志等；
    - 寻址方式，如直接寻址、间接寻址、相对寻址、寄存器寻址等；
  - 同一种操作(如加法)通常有很多条指令。

2.1.2 高级数据表示

1 自定义（Self-defining）数据表示

- 标志符数据表示
  - 一般的计算机中，数据存储单元(寄存器、主存储器、外存储器等)只存放纯数据，数据的属性通过指令中的操作码来解释
  - 高级语言使用类型说明语句指明数据类型，运算符不反映数据类型
  - 高级语言与机器语言之间的语义差距要靠编译器等填补

表 2.13 IBM370 系列机中的加法指令

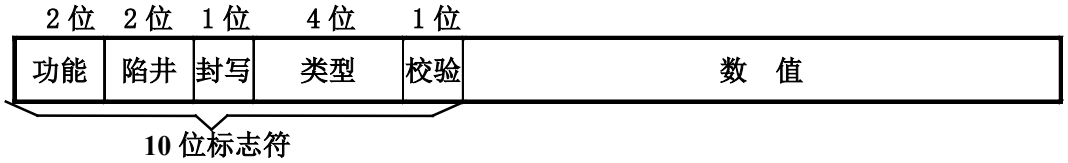
指令助记符	数据类型	字长(位)	进位制	寻址方式
AR	定点数	32	2	R—R
ADR	浮点数	64	尾数 16,阶码 2	R—R
AER	浮点数	32	尾数 16,阶码 2	R—R
AH	定点数	16	2	R—X
A	定点数	32	2	R—X
AD	浮点数	64	尾数 16,阶码 2	R—X
AE	浮点数	32	尾数 16,阶码 2	R—X
AP	定点十进制	64	10	S—S

2.1.2 高级数据表示

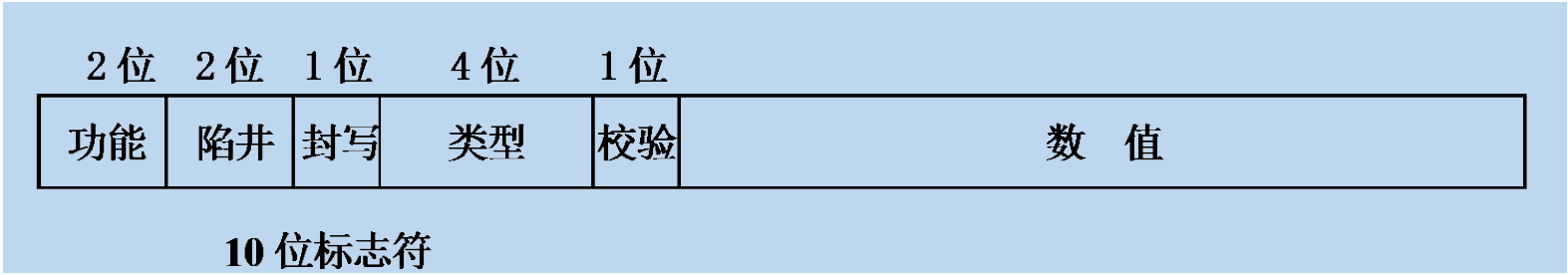
1 自定义（Self-defining）数据表示

- **标志符数据表示**
  - 为缩短高级语言与机器语言之间的语义差距，可以让机器中的每个数据都加上类型标志位
    - 在B5000大型机中，每个数据有一位标志符来区分操作数和描述符
    - 在B6500和B7500大型机中，每个数据有三位标志符来区分8种数据类型
    - 在R-2巨型机中采用10位标志符

带有标志符的数据表示方式



在 R-2 巨型机中带标志符的数据表示方式



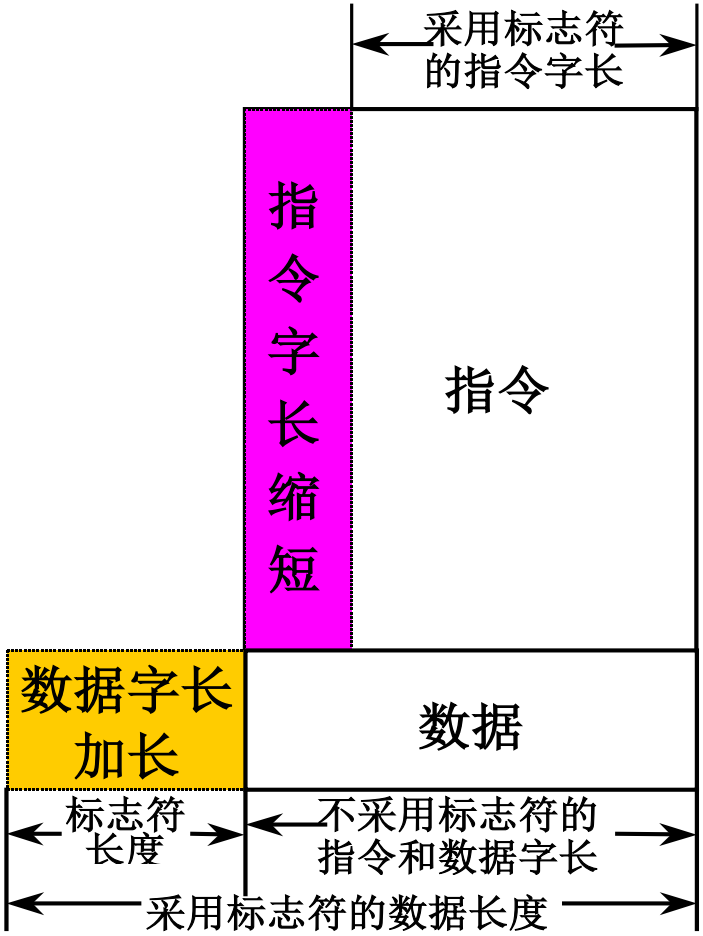
- R-2巨型机中的标志符
  - **功能位**：操作数、指令、地址、控制字
  - **陷井位**：由软件定义四种捕捉方式
  - **封写位**：指定数据是只读的还是可读可写
  - **类型位**：二进制, 十进制, 定点数, 浮点数, 复数, 字符串, 单精度, 双精度；绝对地址、相对地址、变址地址、未连接地址等。

2.1.2 高级数据表示

1 自定义（Self-defining）数据表示

- 标志符数据表示
  - 为缩短高级语言与机器语言之间的语义差距，可以让机器中的每个数据都加上类型标志位
  - 标志符由编译器或其它系统软件建立，对程序员透明
  - 程序（包括指令和数据）的存储量分析：数据存储量增加，指令存储量减少。

常规数据表示方法与带标志符数据表示方法的比较



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。计算这两种不同类型的处理机中程序所占用的存储空间比。

解：X和Y处理机程序占用的存储空间总和分别为

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

程序占用存储空间的比值为

$$\frac{B_Y}{B_X} = \frac{15\mathrm{R} + 35}{16R + 32}$$

当$R > 3$时，有$\frac{B_Y}{B_X} < 1$;而实际应用中经常是$R > 10$。所以，**标志符的处理机所占用的存储空间通常要小**。

- **采用标志符数据表示方法的主要优点：**

1) 简化了指令系统。

2) 由硬件实现一致性检查和数据类型转换。

3) 简化编译器，使高级语言与机器语言之间的语义差距大大缩短。

4) 支持数据库系统，一个软件不加修改就可适用于多种数据类型。

5) 方便软件调试，在每个数据中都有陷井位。

- **采用标志符数据表示方法的主要缺点：**

1） 数据和指令的长度可能不一致

- 可以通过精心设计指令系统来解决。

2） 指令的执行速度降低

- 但是，程序的运行时间是由设计时间、编译时间和调试时间共同组成的。
- 采用标志符数据表示方法，程序的设计时间、编译时间和调试时间可以缩短。

3） 硬件复杂度增加

- 由硬件实现一致性检查和数据类型的转换。

2.1.2 高级数据表示

1 自定义（Self-defining）数据表示

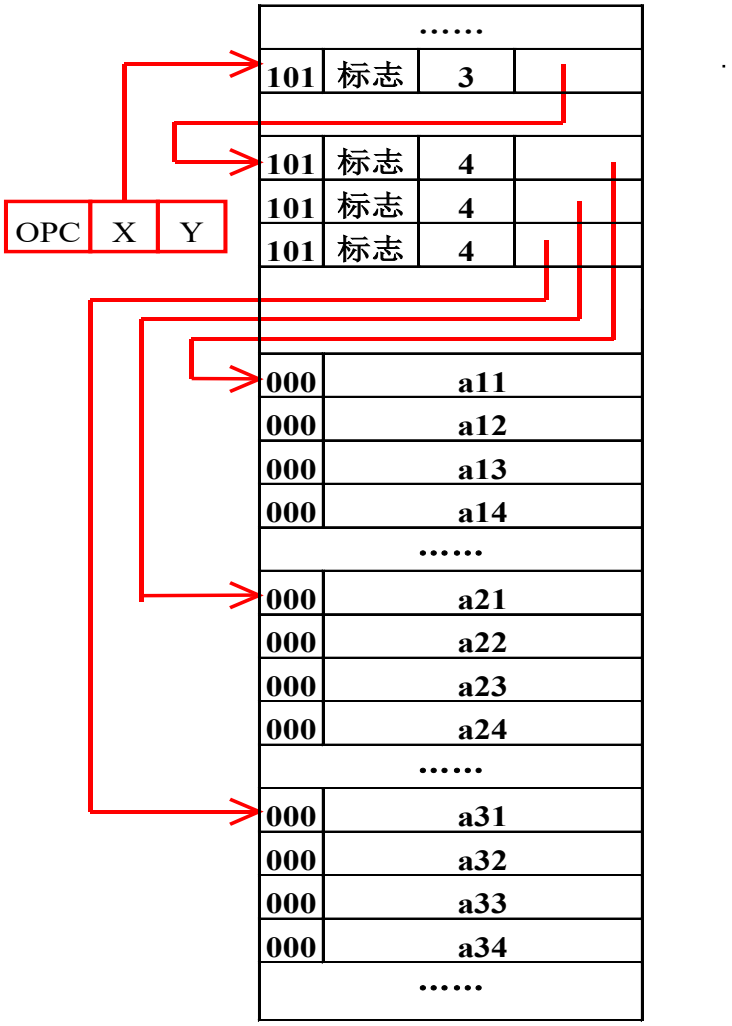
- **数据描述符**
  - 为进一步减少标志符所占用的存储空间，对向量、数据、记录等数据，由于元素属性相同，采用数据描述符。
  - 数据描述符与标志符的区别：标志符只作用于一个数据，而数据描述符要作用于一组数据。
  - Burroughs公司生产的B-6700机中采用的数据描述符表示方法：最高三位为101时表示数据描述符，最高三位为000时表示数据

数据描述符

数据

例：用数据描述符表示方法表示一个3×4的矩阵：

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



2.1.2 高级数据表示

2 向量、数组数据表示

- 为向量、数组数据结构的实现和快速运算提供个更好的硬件支持的方法是增设数组数据表示，组成向量机。

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)
```

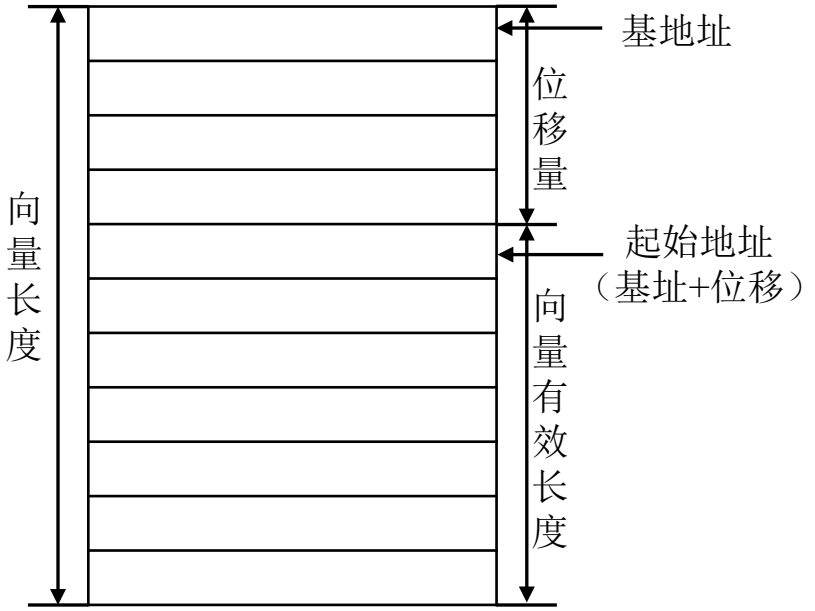
2.1.2 高级数据表示

2 向量、数组数据表示

- 为向量、数组数据结构的实现和快速运算提供个更好的硬件支持的方法是增设数组数据表示，组成向量机。

向量加	A向量参数	B向量参数	C向量参数

对参加运算的源向量A、B以及结果向量C都应指明其**基地址**、**位移量**、**向量长度**和**元素步距**等参数



2.1.2 高级数据表示

3 堆栈数据表示

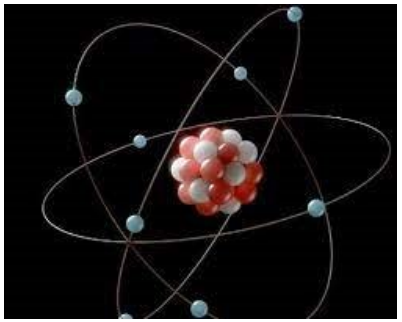
- 堆栈数据结构在编译和子程序调用中很有用，为高效实现，很多机器都设置有堆栈数据表示。
- 一般通用寄存器型机器对堆栈数据结构实现的支持较差：堆栈操作用的机器指令少、功能单一；堆栈置于存储器内，访问速度较低，通常只用于保存子程序调用时的返回地址，较少用于实现程序之间的参数传递
- 堆栈机器的特点：1）有高速寄存器组成的硬件堆栈；2）丰富的堆栈操作指令，功能强大；3）支持高级语言编译；4）支持子程序的嵌套和递归调用

2.1.3 引入数据表示的原则

- 看系统的效率是否显著提高，包括实现时间和存储空间是否显著减少。
- 看引入这种数据表示后，其通用性和利用率是否提高。

2.1.4 浮点数的表示方法

浮点数



- IEEE 754 single-precision (32-bit)
  - Scale: $1.401 \times 10^{-45} \longrightarrow 1.701 \times 10^{+38}$
- IEEE 754 double-precision (64-bit)
  - Scale: $4.941 \times 10^{-324} \longrightarrow 1.798 \times 10^{+308}$

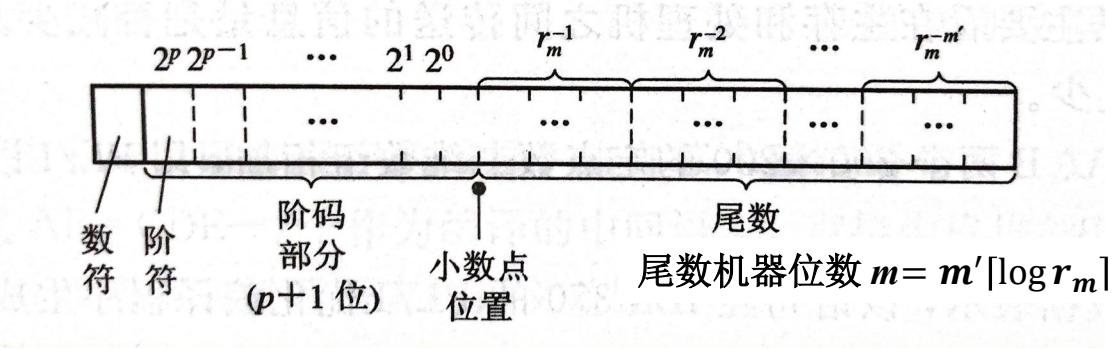
$$0.101 \times 2^{-3} = ?$$

$$(1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-3}$$

$$\left(\frac{1}{2} + \frac{1}{8}\right) \times \frac{1}{8} = 0.078125$$

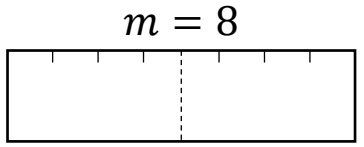
2.1.4 浮点数的表示方法

1. 浮点数的表示方式



- **尾数**：数制(小数或整数)和码制(原码或补码)
- **阶码**：整数，移码(偏码、增码、余码)或补码
- **尾数基值**：2、4、8、16和10进制等 r_m
- **阶码基值**： 通常为2进制
- **尾数长度**：尾数部分按基值计算的长度 m'
- **阶码长度**：阶码部分的二进制位数 p
- 尾数决定了浮点数的表示精度，阶值决定了浮点数的表示范围

$$0.101 \times 2^{-3}$$



$r_m = 2, m' = 8$

$r_m = 16, m' = 2$

2.1.4 浮点数的表示方法

相同浮点数的不同表示

$$0.101 \times 2^{-3} \quad vs. \quad 0.0101 \times 2^{-2}$$

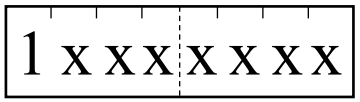
浮点数的规格化

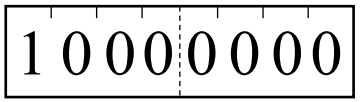
- 小数点总是点在最高有效位的左边

$$0.1xx \times 2^{e}$$

0.101×2^{-3} ~~0.0101×2^{-2}~~

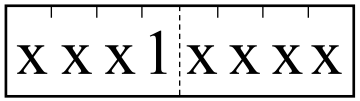
- r_m=2

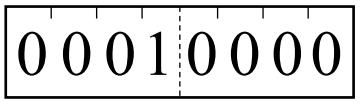




$\frac{1}{2}$

- r_m=16

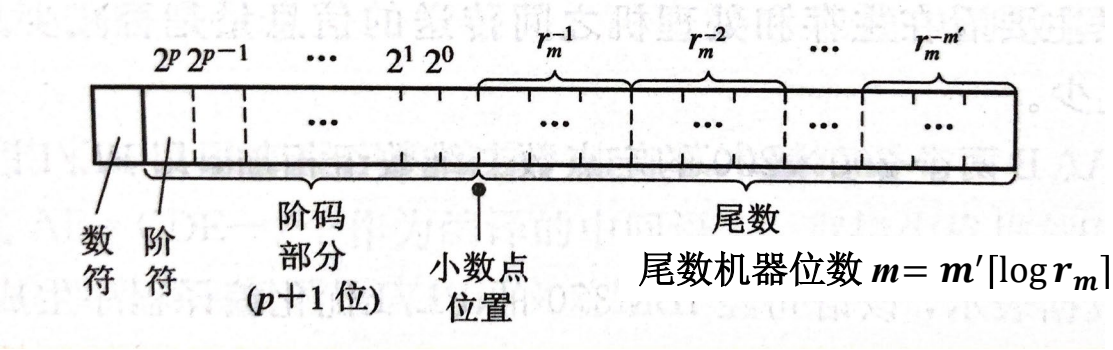




$\frac{1}{16}$

2.1.4 浮点数的表示方法

1. 浮点数的表示方式



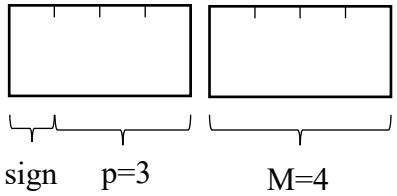
非负阶、规格化、正尾数

- 可表示最小尾数（小数点后第1个r_m 进制数位为1）：r_m^{-1}
- 可表示最大尾数（ r_m 进制尾数均为$r_m - 1$）：$1 - r_m^{-m'}$
- 最大阶值（阶值部分全为1）：$2^p - 1$
- 可表示最小值： r_m^{-1}
- 可表示最大值： $\left(1 - r_m^{-m'}\right) \times r_m^{2^p-1}$
- 可表示尾数个数：$r_m^{m'-1}(r_m - 1)$
- 可表示阶的个数：2^p
- 可表示数的个数： $2^p r_m^{m'-1}(r_m - 1)$

2.1.4 浮点数的表示方法

1. 习题

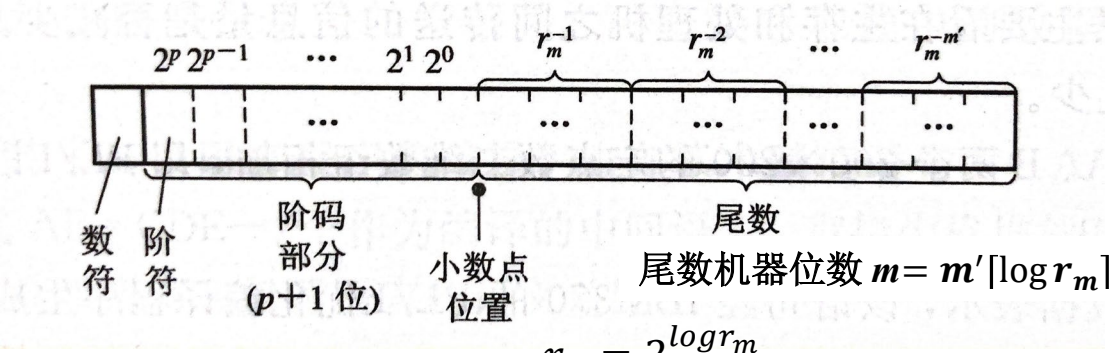
阶码长度p=3，尾数长度M=4，非负阶、规格化、正尾数



	$r_m=2$: $m'=4$	$r_m=16$: $m'=1$
可表示最小尾数：	$r_m^{-1}=2^{-1}$	$r_m^{-1}=16^{-1}$
可表示最大尾数：	$1-r_m^{-m'}=\frac{15}{16}$	$1-r_m^{-m'}=\frac{15}{16}$
最大阶值：	7	7
可表示最小值：	$r_m^{-1}=2^{-1}$	$r_m^{-1}=16^{-1}$
可表示最大值：	$\left(1-r_m^{-m'}\right)\times r_m^{2^p-1}=120$	$\left(1-r_m^{-m'}\right)\times r_m^{2^p-1}=16^6\times 15$
可表示尾数个数：	$r_m^{m'-1}(r_m-1)=8$	$r_m^{m'-1}(r_m-1)=15$
可表示阶的个数：	$2^p=8$	$2^p=8$
可表示数的个数：	$2^p r_m^{m'-1}(r_m-1)=64$	$2^p r_m^{m'-1}(r_m-1)=120$

2.1.4 浮点数的表示方法

1. 浮点数的表示方式



尾数机器位数 $m = m'[\log r_m]$

$$r_m = 2^{log r_m}$$

非负阶、规格化、正尾数（r_m 为2的整数次幂时，$r_m^{m'} = 2^m$）

- 可表示最小尾数（小数点后第1个r_m 进制数位为1）：r_m^{-1}
- 可表示最大尾数（r_m 进制尾数均为$r_m - 1$）：$1 - r_m^{-m'} = 1 - 2^{-m}$
- 最大阶值（阶值部分全为1）：$2^p - 1$
- 可表示最小值：r_m^{-1}
- 可表示最大值：$\left(1 - r_m^{-m'}\right) \times r_m^{2^p-1} = r_m^{2^p-1} \times (1 - 2^{-m})$
- 可表示尾数个数：$r_m^{m'-1}(r_m - 1) = 2^m \times (r_m - 1)/r_m$
- 可表示阶的个数：2^p
- 可表示数的个数：$2^p r_m^{m'-1}(r_m - 1) = 2^{p+m}(r_m - 1)/r_m$

2.1.4 浮点数的表示方法

1. 浮点数的表示方式

非负阶、规格化、正尾数（r_m 为2的整数次幂时，$r_m^{m'} = 2^m$）

- 可表示最小尾数（小数点后第1个r_m 进制数位为1）：r_m^{-1}
- 可表示最大尾数（ r_m 进制尾数均为$r_m - 1$）：$\sum_{i=1}^{m'} r_m^{-i} = 1 - r_m^{-m'} = 1 - 2^{-m}$
- 最大阶值（阶值部分全为1）：$2^p - 1$
- 可表示最小值：r_m^{-1}
- 可表示最大值：$\left(1 - r_m^{-m'}\right) \times r_m^{2^p-1} = r_m^{2^p-1} \times (1 - 2^{-m})$
- 可表示尾数个数：$r_m^{m'-1}(r_m - 1) = 2^m \times (r_m - 1)/r_m$
- 可表示阶的个数：2^p
- 可表示数的个数：$2^p r_m^{m'-1}(r_m - 1) = 2^{p+m}(r_m - 1)/r_m$

1）可表示数的范围：随着r_m的增大，可表示最小值减小，可表示最大值增大，即可表示数的范围增大了

2.1.4 浮点数的表示方法

1. 浮点数的表示方式

非负阶、规格化、正尾数（r_m 为2的整数次幂时，$r_m^{m'} = 2^m$）

- 可表示最小尾数（小数点后第1个r_m 进制数位为1）：r_m^{-1}
- 可表示最大尾数（r_m 进制尾数均为$r_m - 1$）：$\sum_{i=1}^{m'} r_m^{-i} = 1 - r_m^{-m'} = 1 - 2^{-m}$
- 最大阶值（阶值部分全为1）：$2^p - 1$
- 可表示最小值：r_m^{-1}
- 可表示最大值：$\left(1 - r_m^{-m'}\right) \times r_m^{2^p-1} = r_m^{2^p-1} \times (1 - 2^{-m})$
- 可表示尾数个数：$r_m^{m'-1}(r_m - 1) = 2^m \times (r_m - 1)/r_m$
- 可表示阶的个数：2^p
- 可表示数的个数：$2^p r_m^{m'-1}(r_m - 1) = 2^{p+m}(r_m - 1)/r_m$

2）可表示数的个数：随着r_m的增大，可表示数的个数增大

2.1.4 浮点数的表示方法

1. 浮点数的表示方式

3）数在数轴上的分布：当$r_m = 2$时，可表示浮点数的个数2^{p+m-1}, 可表示数的最大值$2^{2^p-1} \times (1-2^{-m})$，则当$r_m > 2$时， 求小于$\mathbf{2^{2^p-1} \times (1-2^{-m})}$的浮点数的个数

- 当$r_m > 2$时，存在阶值q，使得 $r_m^q \times (1-2^{-m}) \approx 2^{2^p-1} \times (1-2^{-m})$，即 $r_m^q \approx 2^{2^p-1}$
- 因此 $q = \frac{2^p-1}{\log r_m}$
- 尾数≤ $(1-2^{-m})$且阶值≤ q的所有尾数基值> 2的规格化浮点数都在$r_m = 2$可表示的最大值以内
- 有$q+1$种阶值和$2^m \times (r_m-1)/r_m$种尾数，共有$2^m(1-r_m^{-1})(q+1)$中可表示的值在$r_m = 2$可表示的最大值以内
- 表示比

$$e \approx \frac{2^m(1-r_m^{-1})(q+1)}{2^{p+m-1}} = 2^{1-p}(1-r_m^{-1})\left(1+\frac{2^p-1}{\log r_m}\right)$$

- 实际机器中p至少为8，若$r_m = 16$，则$e = 0.47$
- **结论： r_m越大，在与r_m=2的浮点数相重叠的范围内，数的密度分布越稀疏**

2.1.4 浮点数的表示方法

1. 浮点数的表示方式

4）可表示的精度：r_m越大，数在数轴上的分布越稀疏，数的表示精度下降

5）运算中的精度损失：r_m取大后，对阶移位的机会和次数减少，数的表示范围扩大，使尾数溢出需要右规的机会也变少，精度损失越小

5）运算速度：r_m取大后，由于对阶或尾数溢出需右移及规格化需左移的次数减少，运算速度提高

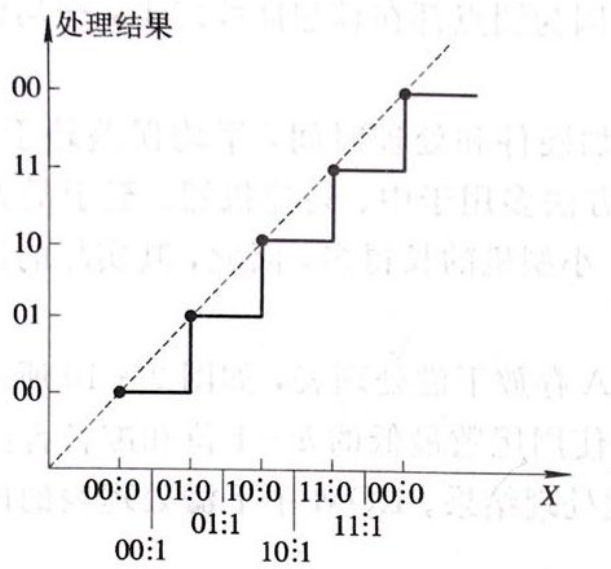
总结：尾数基值增大，会扩大浮点数表示范围，增加可表示数的个数，减少移位次数，降低右移造成的精度损失，提高运算速度，但也会降低数据的表示精度，数值的分布变稀疏

2.1.4 浮点数的表示方法

2. 尾数下溢处理方法

（1）截断法：将尾数超出机器字长的部分截去。优点是实现简单，不增加硬件，不需要处理时间；缺点是平均误差较大且无法调节

		尾数有效字长 p 位	有效字长之外 g 位	误差情况
正数区	舍入前	0.xxx……xx	00……000	$\delta=0$
		0.xxx……xx	00……001	$\delta=-2^{-p-g}$
		0.xxx……xx	00……010	$\delta=-2^{-p-g+1}$
		……	……	……
		0.xxx……xx	11……111	$\delta=-2^{-p}(1-2^{-g})$
	舍入后	0.xxx……xx		$\sigma=-2^{-p-1}(2^{g}-1)$
负数区	舍入前	−0.xxx……xx	00……000	$\delta=0$
		−0.xxx……xx	00……001	$\delta=+2^{-p-g}$
		−0.xxx……xx	00……010	$\delta=+2^{-p-g+1}$
		……	……	……
		−0.xxx……xx	11……111	$\delta=+2^{-p}(1-2^{-g})$
	舍入后	−0.xxx……xx		$\sigma=+2^{-p-1}(2^{g}-1)$



$$\sigma = a_0 n + d\frac{(n-1)\cdot n}{2} = -2^{-p-1}(2^{g}-1)$$

2.1.4 浮点数的表示方法

2. 尾数下溢处理方法

（1）截断法：

	p=2		g=2		误差
	1/2	1/4	1/8	1/16	
舍入前	x	x	0	0	0
	x	x	0	1	-1/16
	x	x	1	0	-1/8
	x	x	1	1	-3/16
舍入后	x	x			-3/8
平均误差					-3/32

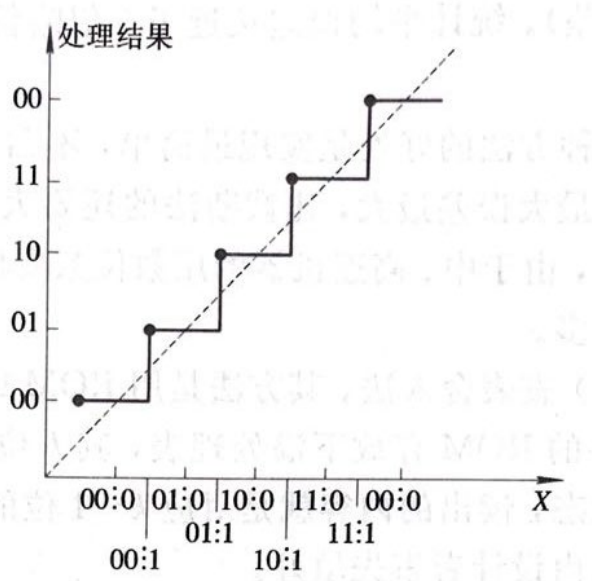
	p=2		g=3			误差
	1/2	1/4	1/8	1/16	1/32	
舍入前	x	x	0	0	0/1	0 or -1/32
	x	x	0	1	0/1	-1/16 or -3/32
	x	x	1	0	0/1	-1/8 or -5/32
	x	x	1	1	0/1	-3/16 or -7/32
舍入后	x	x				-7/8
平均误差						-7/64

2.1.4　浮点数的表示方法

2. 尾数下溢处理方法

（2）舍入法：在机器运算的规定字长之外增设一位附加位，存放溢出部分的最高位，每当进行尾数下溢处理时，将附加位加1。优点是实现简单，增加硬件很少，最大误差小，平均误差接近于零；缺点是处理速度慢，需要花费在附加位上加1以及因此产生的进位时间

正数区	尾数有效字长 p 位	有效字长之外 g 位	误差δ
舍入前	0.xxx……xx	00……000	0
	0.xxx……xx	00……001	-2^{-p-g}
	0.xxx……xx	00……010	-2^{-p-g+1}
	……	……	……
	0.xxx……xx	01……111	$-2^{-p-1}(1-2^{-g})$
	0.xxx……xx	10……000	$+2^{-p-1}$—积累误差
	0.xxx……xx	10……001	$+2^{-p-1}(1-2^{-g})$
	……		……
	0.xxx……xx	11……110	$+2^{-p-g+1}$
	0.xxx……xx	11……111	$+2^{-p-g}$
舍入后	0.xxx……xx 0.xxx……xx+2^{-p}	最高位为 **0** 最高位为 **1**	积累误差σ＝＋2^{-p-1}



2.1.4　浮点数的表示方法

2. 尾数下溢处理方法

（2）舍入法

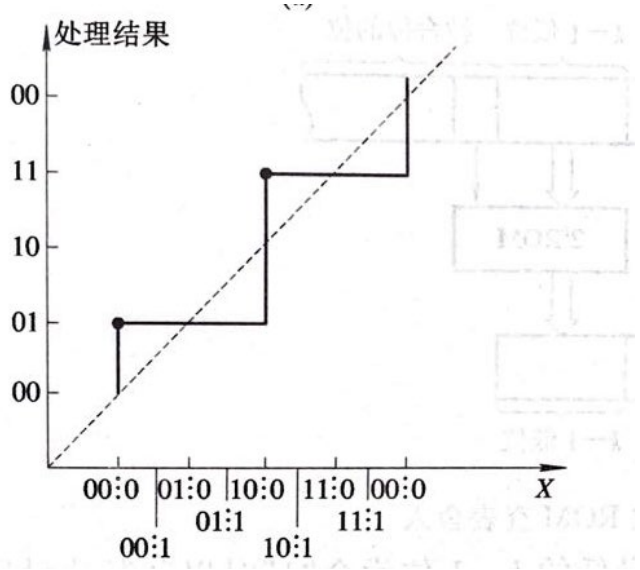
	p=2		g=2		误差
	1/2	1/4	1/8	1/16	
舍入前	x	x	0	0	0
	x	x	0	1	-1/16
	x	x	1	0	1/8
	x	x	1	1	1/16
舍入后	x	x			1/8
	x	x+1			
平均误差					1/32

2.1.4 浮点数的表示方法

2. 尾数下溢处理方法

（3）恒置“1”法：把有效字长的最低一位置成1。优点是实现简单，不需要增加硬件和处理时间，平均误差接近0；缺点是最大误差较大

正数区	尾数有效字长 p 位	有效字长外 g 位	误差情况
舍入前	0.xxx……xx0	00……000	$+2^{-p}$ —— 误差积累
	0.xxx……xx0	00……001	$+2^{-p}(1-2^{-g})$
	0.xxx……xx0	00……010	$+2^{-p}(1-2^{-g+1})$
	……	……	……
	0.xxx……xx0	11……111	$+2^{-p-g}$
	0.xxx……xx1	00……000	0
	0.xxx……xx1	00……001	-2^{-p-g}
			……
	0.xxx……xx1	11……110	$-2^{-p}(1-2^{-g+1})$
	0.xxx……xx1	11……111	$-2^{-p}(1-2^{-g})$
舍入后	0.xxx……xx1		积累误差$\sigma=2^{-p}$



2.1.4 浮点数的表示方法

2. 尾数下溢处理方法

（3）恒置“1”法：

	p=2		g=2		误差
	1/2	1/4	1/8	1/16	
舍入前	x	0	0	0	1/4
	x	0	0	1	3/16
	x	0	1	0	1/8
	x	0	1	1	1/16
	x	1	0	0	0
	x	1	0	1	-1/16
	x	1	1	0	-1/8
	x	1	1	1	-3/16
舍入后	x	1			1/4
平均误差					1/32

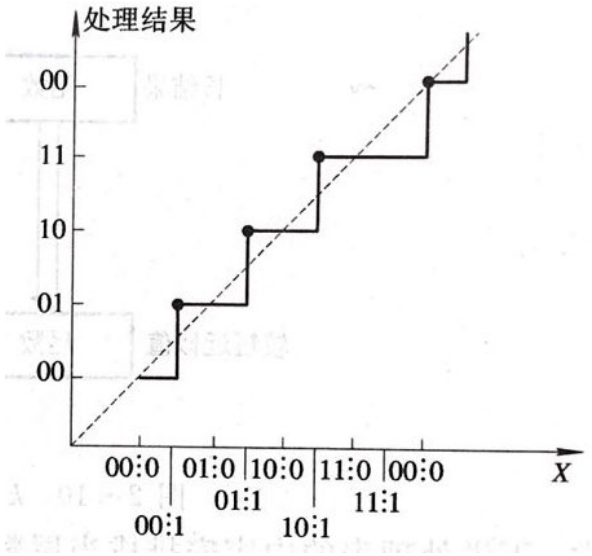
2.1.4 浮点数的表示方法

2. 尾数下溢处理方法

（4）查表舍入法：用ROM或者PLA存放下溢处理表。优点是速度快，平均误差可以调节到0；缺点是硬件量大。

ROM地址	舍入前(p+g位)	舍入后(p位)	误差情况	
000	xx…x00|0xx…x	xx…x00	$-2^{-p-1}(1-2^{-g+1})\sim 0$	$\}+2^{-p-1}$
001	xx…x00|1xx…x	xx…x01	$+2^{-p-g}\sim 2^{-p-1}$	
010	xx…x01|0xx…x	xx…x01	$-2^{-p-1}(1-2^{-g+1})\sim 0$	$\}+2^{-p-1}$
011	xx…x01|1xx…x	xx…x10	$+2^{-p-g}\sim 2^{-p-1}$	
100	xx…x10|0xx…x	xx…x10	$-2^{-p-1}(1-2^{-g+1})\sim 0$	$\}+2^{-p-1}$
101	xx…x10|1xx…x	xx…x11	$+2^{-p-g}\sim 2^{-p-1}$	
110	xx…x11|0xx…x	xx…x11	$-2^{-p-1}(1-2^{-g+1})\sim 0$	$\}-2^{-p-1}(2^{g}-1)$
111	xx…x11|1xx…x	xx…x11	$-2^{-p}(1-2^{-g})\sim -2^{-p-1}$	

n=2



- 前$2^{n+1}-2$行采用下舍上入法，总的积累误差为：

  $(2^n-1)2^{-p-1}$，　　与g无关

- 最后2行采用恒舍法，总的积累误差为：

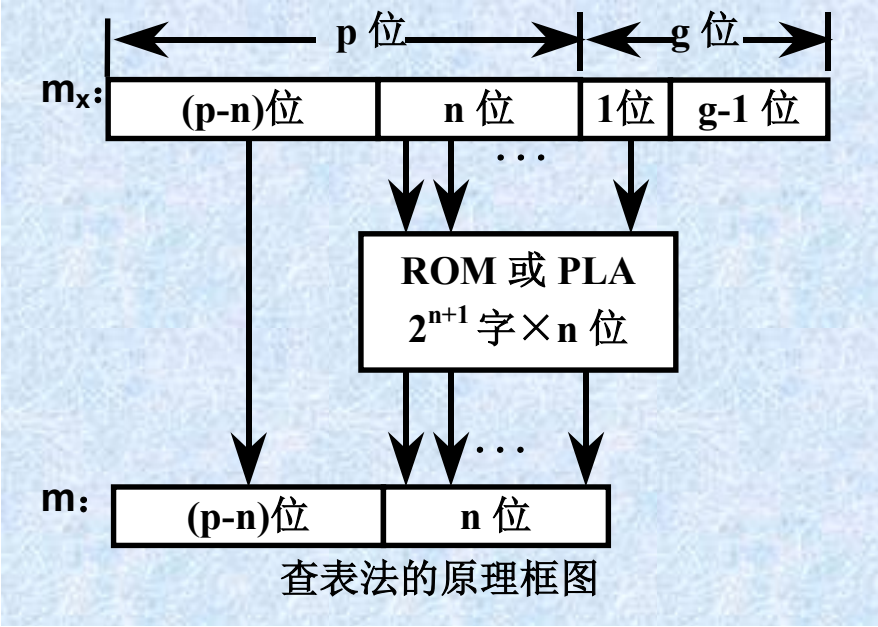
  $-2^{-p-1}(2^g-1)$，　　与g有关

- 总的积累误差为：$2^{-p-1}(2^n-2^g)$
  - 当n=g的时候，积累误差完全平衡
  - 当n>g的时候，积累误差为正
  - 当n<g的时候，积累误差为负
- 在负数区积累误差的分析方法与正数区相同，但要把+、-、>、<等符号反过来。
- 通过精心设计ROM中所存储的内容，针对各种不同应用领域，使积累误差尽可能小。

2.1.4　浮点数的表示方法

2. 尾数下溢处理方法

（4）查表舍入法：用ROM或者PLA存放下溢处理表。优点是速度快，平均误差可以调节到0；缺点是硬件量大。



查表法的原理框图

- 恒置法虽有少量的积累误差，且损失一位精度，但由于实现很容易，普遍在小型微型机中使用。
- 舍入法只有少量积累误差，且精度比较高，但实现很复杂，用于软件实现的算法中。
- 查表法实现比较容易，积累误差很小，且可以通过改变ROM或PLA中的内容来修正积累误差，是一种很有前途的舍入方法。

2.2 寻址技术

2.2.1 编址方式

2.2.2 寻址方式

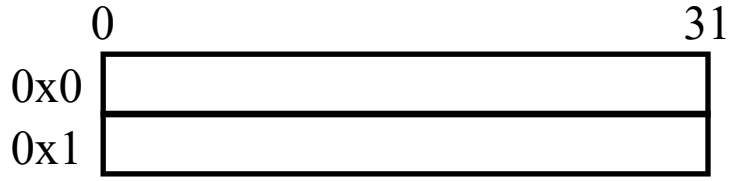
2.2.3 定位方式

2.2.1 编址方式

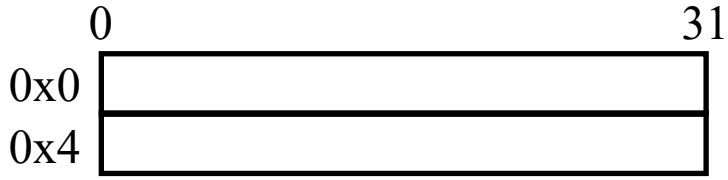
1. 编址单位

- 常用的编址单位：字编址、字节编址、位编址、块编址等
- 编址单位与访问字长
  - **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，位访问

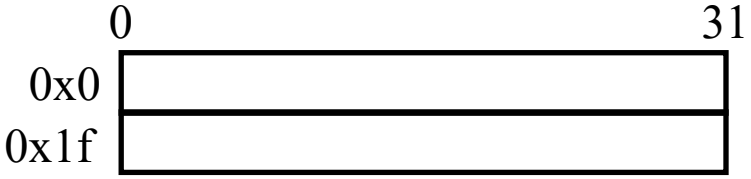
字编址:



字节编址:

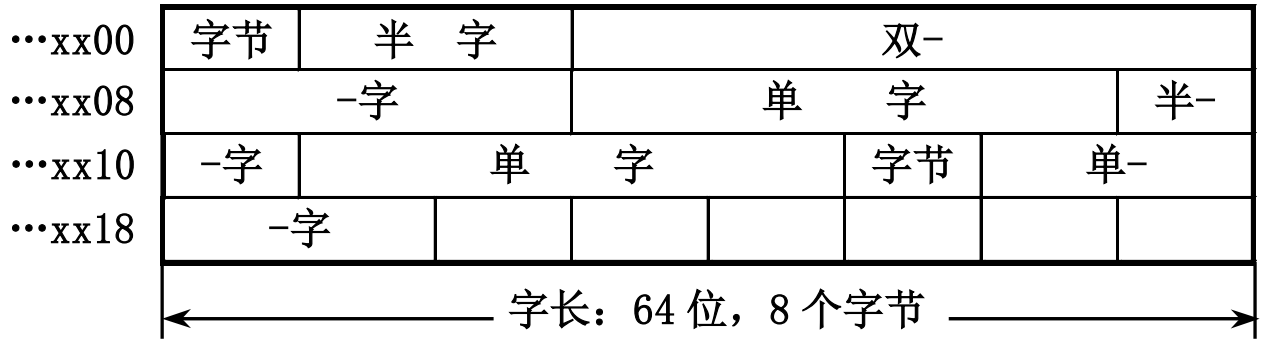


位编址:



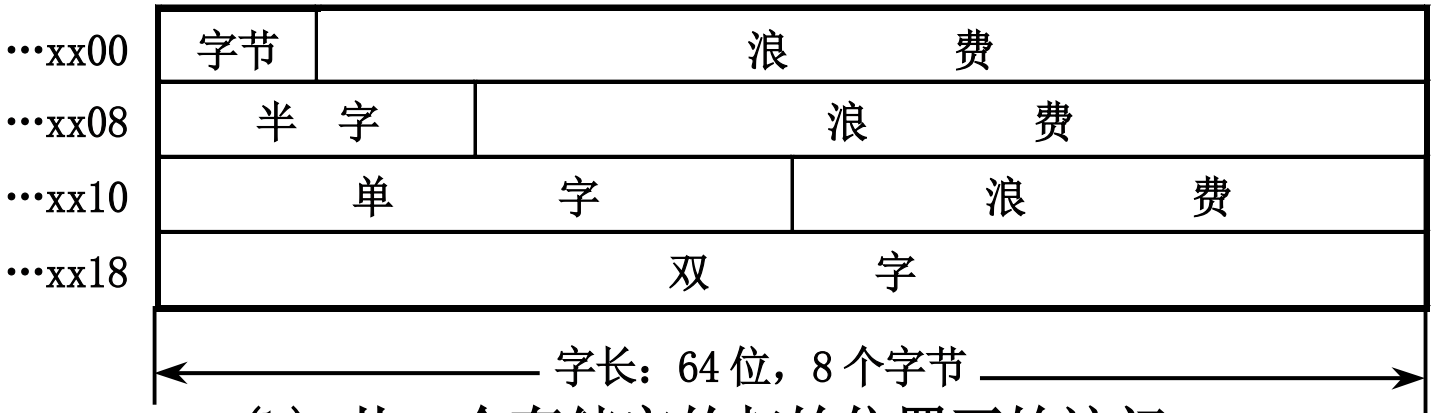
- 字节编址字访问的优点：有利于符号处理
- 字节编址字访问的问题：
    1) 地址信息/存储器空间浪费 or 读写逻辑复杂
    2) 大端(Big Endian)与小端(Little Endian)问题

- 数据紧凑储存，所有地址有效
- 虽然节省存储资源，但是可能降低读写速度
- 读写控制逻辑复杂



(a) 可从任意位置开始访问

- 数据按字对齐储存
- 地址信息浪费，低3位（64位机）或低2位（32位机）始终是0
- 储存空间浪费
- 读写速度快，读写控制逻辑简单，但是浪费存储器资源



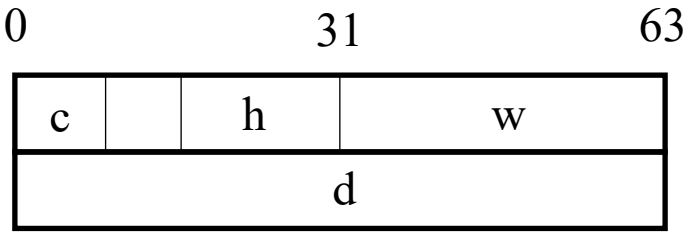
(b) 从一个存储字的起始位置开始访问

- 不同数据对齐位置不同：双字地址最末三个二进制位必须为000，单字地址最末两位必须为00，半字地址最末一位必须为0
- 无论访问哪种类型的数据，都能在一个周期内完成
- 在一定程度上浪费了存储器资源，控制逻辑仍然比较复杂

（c）从地址的整倍数位置开始访问

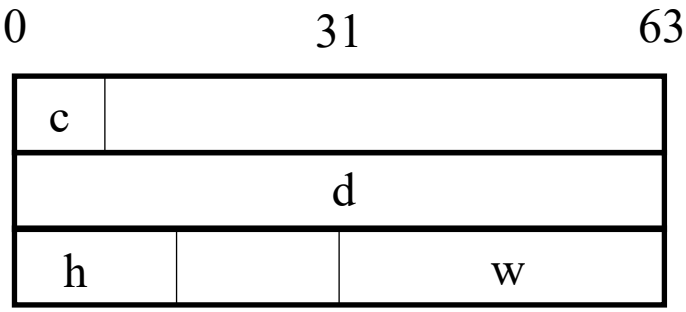
Examples on a laptop

```
4 typedef struct {
5     char c;
6     int16_t h;
7     int32_t w;
8     int64_t d;
9 } showcase;
```



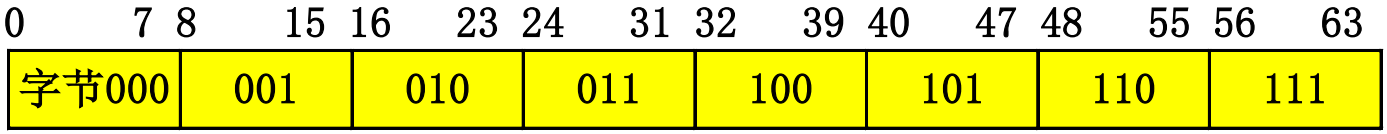

```
sizeof(showcase) = 16
```

```
4 typedef struct {
5     char c;
6     int64_t d;
7     int16_t h;
8     int32_t w;
9 } showcase;
```

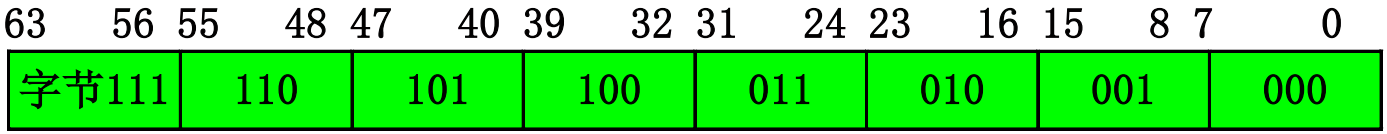



```
sizeof(showcase) = 24
```

字内编码表示



(a) 从左边开始编址



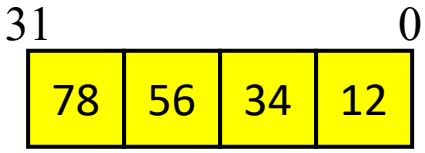
(b) 从右边开始编址

一个存储字的两种编址方式

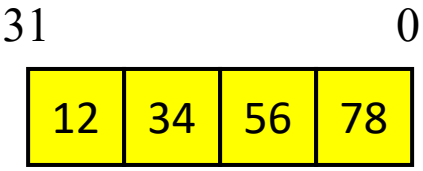
大端(Big Endian)与小端(Little Endian)问题

0x12345678

Big endian



Little endian



2. 零地址空间个数

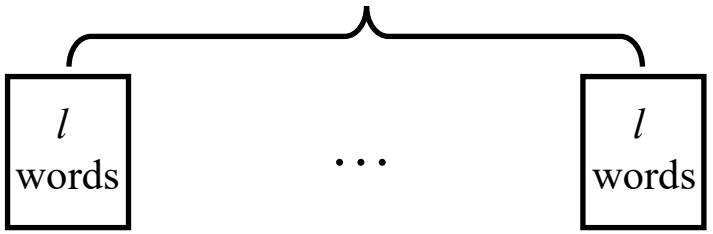
- 三个零地址空间：通用寄存器、主存储器、输入输出设备独立编址
- 两个零地址空间：主存储器与输入输出设备统一编址
- 一个零地址空间：最低端是通用寄存器，最高端是输入输出设备，中间为主存储器
- 隐含编址方式：堆栈、Cache等

3. 并行存储器的编址技术

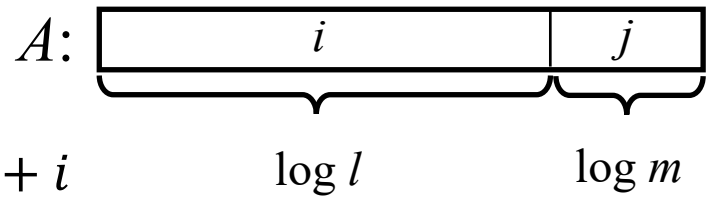
- 高位交叉编址：主要用来扩大存储器容量。
- 低位交叉编址：主要是提高存储器速度。

模m低位交叉编址

- CPU字在主存中按模m低位交叉编址
  - 单体容量为l的m个分体，其M_j体的编址模式为$m \times i + j$，其中$i = 0, 1, 2, \cdots, l-1$，　$j = 0, 1, 2, \cdots, m-1$



- 寻址规则
  - 体地址　$j = A \bmod m$
  - 体内地址　$i = A/m$
    - $M_0: 0, m, 2m, \cdots, m(l-1)+0$
    - $M_i: i, m+i, 2m+i, \cdots, m(l-1)+i$

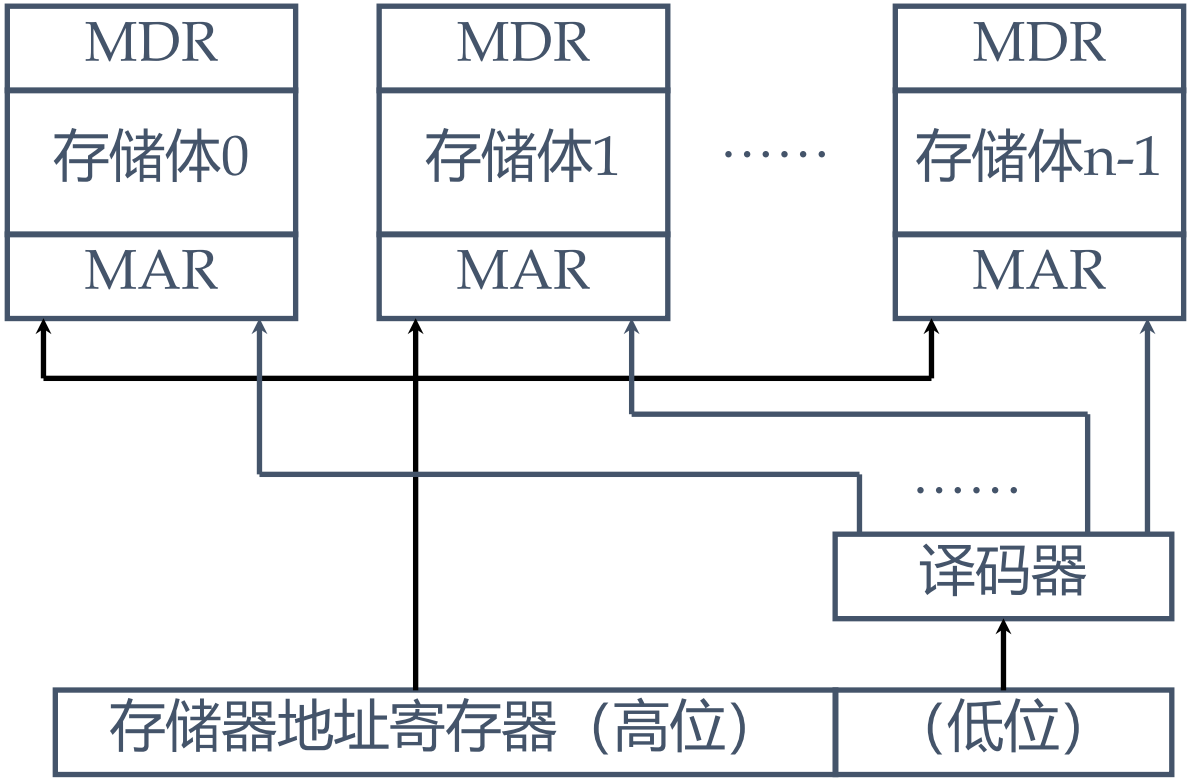


- 提高主储存器的速度

模4低位交叉编址

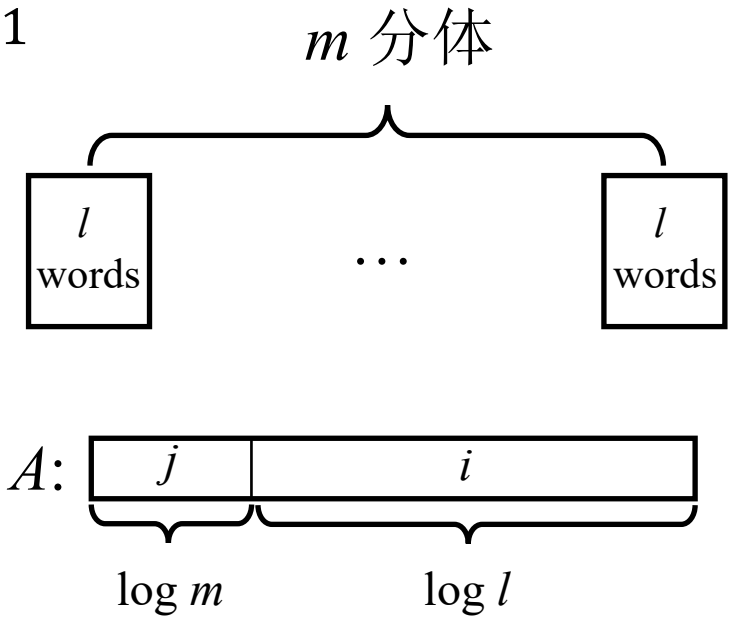
模体	地址编址序列	对应二进制地址码最末二位状态
M_0	0, 4, 8, 12, …, 4i+0, …	00
M_1	1, 5, 9, 13, …, 4i+1, …	01
M_2	2, 6, 10, 14, …, 4i+2, …	10
M_3	3, 7, 11, 15, …, 4i+3, …	11

MDR: memory data register
MAR: memory address register

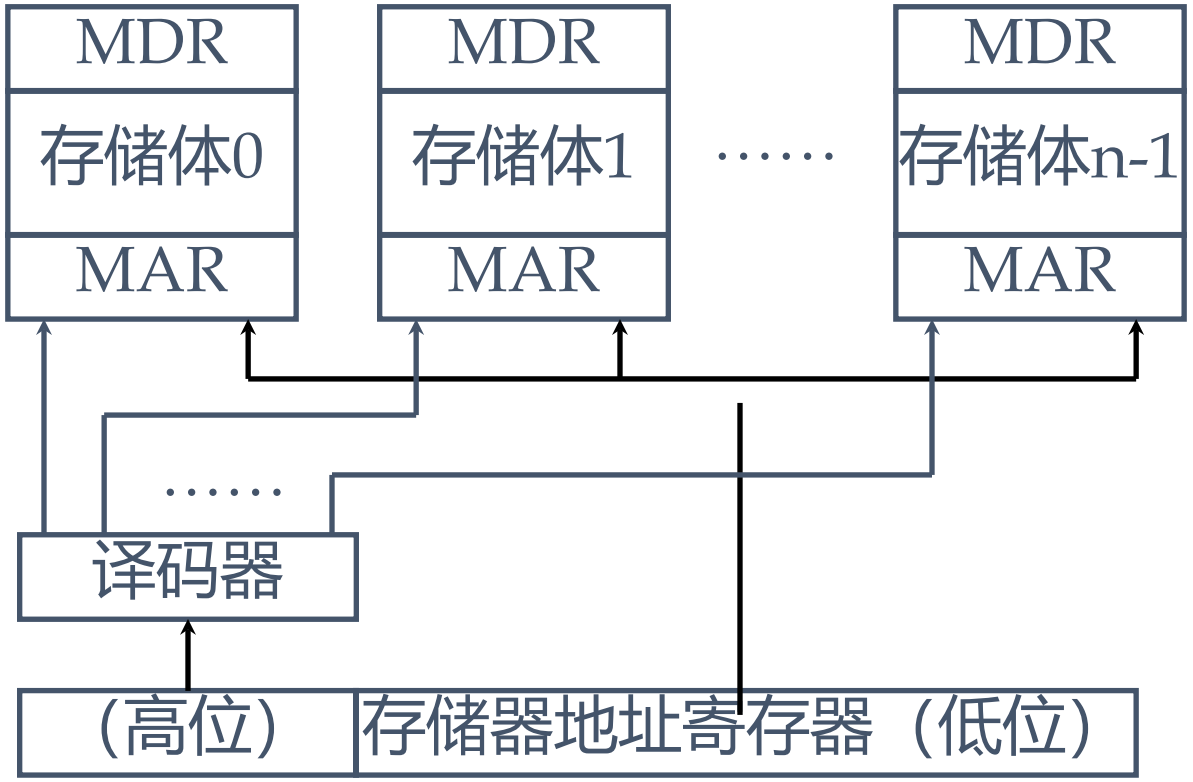


模m高位交叉编址

- CPU字在主存中按模m高位交叉编址
  - 单体容量为l的m个分体，其M_j体的编址模式为$l \times j + i$，其中$i = 0, 1, 2, \cdots, l-1$，　$j = 0, 1, 2, \cdots, m-1$



- 寻址规则
  - 体地址　$j = A / l$
  - 体内地址　$i = A \bmod l$
    - $M_0: 0, 1, 2, \cdots, l-1$
    - $M_i: il, il+1, \cdots, (i+1)l-1$
- 便于主储存器的扩充


MDR
存储体0
MAR
MDR
存储体1
MAR
……
MDR
存储体n-1
MAR
……
译码器
(高位)
存储器地址寄存器（低位）

3. 输入输出设备的编址

- 一台设备一个地址：仅对输入输出设备本身进行编址，需要通过指令中的操作码来识别该输入输出设备接口上的有关寄存器
- 一台设备两个地址：数据寄存器、状态或控制寄存器。
- 多个编址寄存器共用同一个地址的方法：
  - 依靠地址内部来区分，适用于被编址的寄存器的长度比较短
  - “下跟法”隐含编址方式，必须按顺序读写寄存器。
- 一台设备多个地址

2.2.2 寻址方式

1. 寻址方式的设计思想

- **立即数寻址方式**：直接在指令中给出操作数
  - 用于数据比较短，且为源操作数的场合　　OPC　　R，*n*
- **面向寄存器的寻址方式**：指令在执行过程中所需要的操作数来自于寄存器，运算结果也写回到寄存器中

OPC　　R

OPC　　R，R

OPC　　R，R，R

2.2.2 寻址方式

1. 寻址方式的设计思想

- **面向主存储器的寻址方式**：
  - **直接寻址**：在指令中直接给出参加运算的操作数及运算结果所存放的主存地址。
  - **间接寻址**：指令中给出的是操作数地址的地址，必须经过两次（或两次以上）的访主存操作才能得到操作数。
  - **变址寻址（相对寻址、基址寻址）**：指令执行时，用一个硬件加法器，把变址寄存器中给出的基地址加上指令中给出的偏移量，才能得到有效地址。

OPC M　　OPC M, M, M

OPC M, M

2.2.2 寻址方式

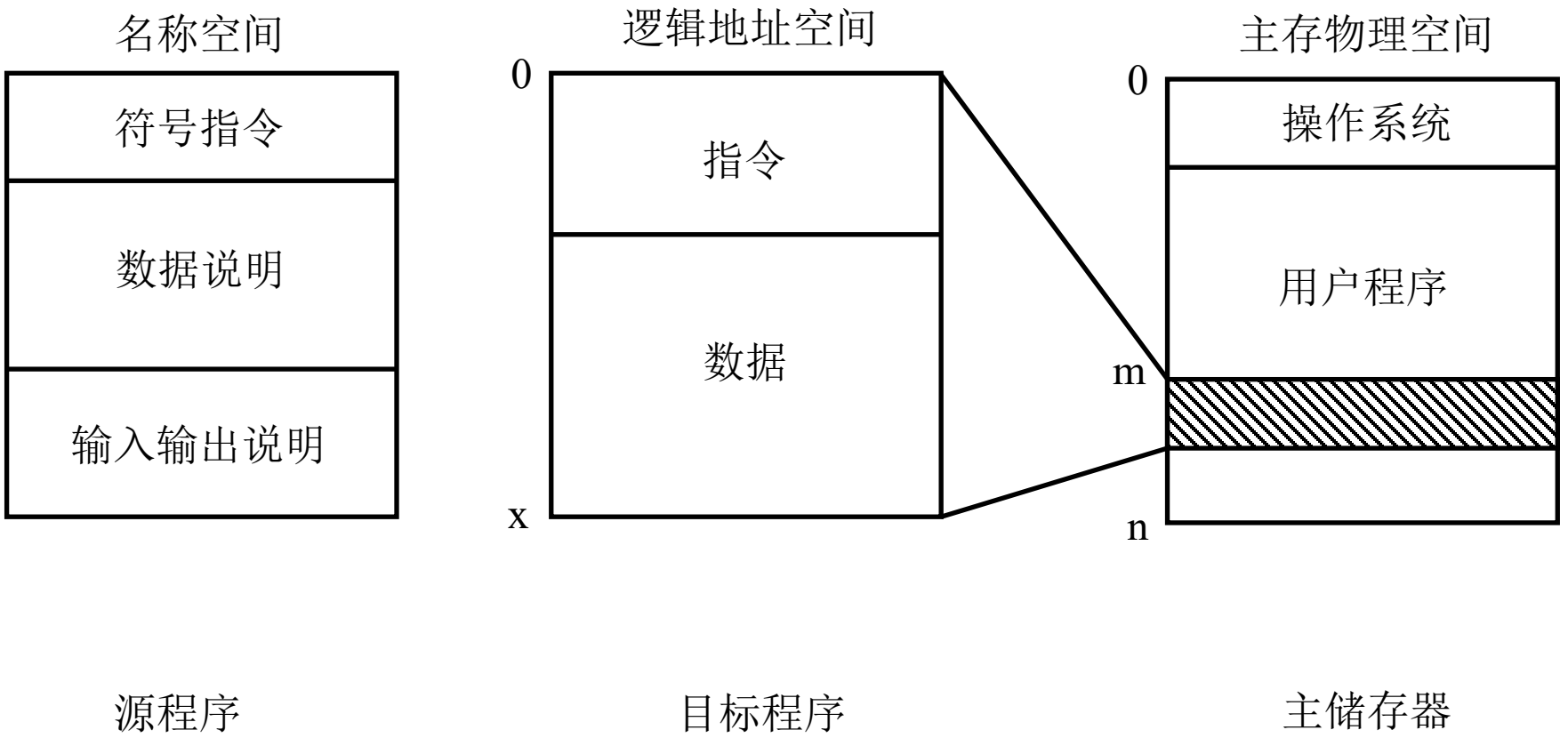
1. 寻址方式的设计思想

- **面向堆栈的寻址方式**：堆栈寻址方式的地址是隐含的，在指令中不需要给出操作数的地址

```
OPC          ;运算型指令
OPC  M       ;数据传送型指令
```

2.2.3 定位方式

- 程序所分配到的主存物理空间和程序本身的逻辑地址空间是不相同的，需要把指令和数据中的逻辑地址（相对地址）转换成主存储器的物理地址（绝对地址）
- 定位方式主要研究**程序的主存物理地址在什么时间确定**，**采用什么方式来实现**
- 程序需要定位的主要原因：
  - 程序的独立性
  - 程序的模块化设计
  - 数据结构在程序运行过程中，其大小往往是变化的
  - 有些程序本身很大，大于分配给它的主存物理空间


名称空间
符号指令
数据说明
输入输出说明
逻辑地址空间
0
指令
数据
x
主存物理空间
0
操作系统
用户程序
m
n
源程序
目标程序
主储存器

2.2.3 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就已经确定了的称为直接定位方式。
- **静态定位**：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**：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。

2.2.3 定位方式

- **动态定位**：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。
  - 增加相应的基址寄存器和地址加法器硬件，在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存入对应该道程序使用的基址寄存器中。
  - 程序执行时，通过地址加法器将逻辑地址加上基址寄存器内的程序基址，形成有效的物理地址
  - 可在指令中加入相应的标志位来指明指令地址是否需要加基址

2.3 指令系统的优化设计

主要目标：节省程序的存储空间

指令格式尽量规整，便于译码

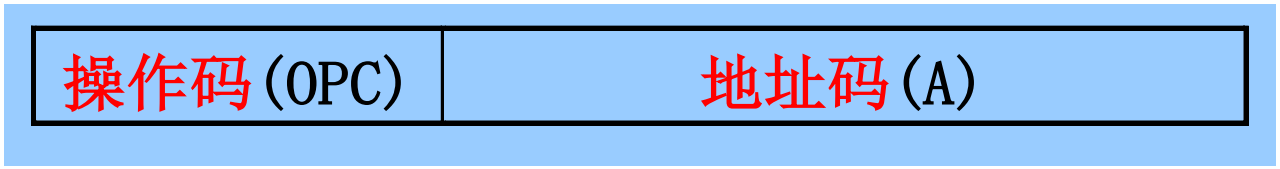
2.3.1 指令的组成

2.3.2 操作码的优化设计

2.3.3 指令字格式的优化设计

2.3.1 指令的组成

➢一般的指令主要由两部分组成：**操作码和地址码**



➢地址码通常包括三部分内容：

- **地址**：地址码、立即数、寄存器、变址寄存器
- **地址的附加信息**：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

2.3.1 指令的组成

- 操作码主要包括两部分内容：
  - **操作种类**：加、减、乘、除、数据传送、移位、转移、输入输出、程序控制、处理机控制等
  - **操作数描述**：
    - 数据的类型：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

2.3.2 操作码的优化表示

- 操作码的三种编码方法：固定长度、Huffman编码、扩展编码
- 优化操作码编码的目的：节省程序存储空间

Burroughs公司B-1700计算机

操作码编码方式	整个操作系统所用指令的操作码总位数	改进的百分比
8 位固定长编码	301,248	0
4-6-10 扩展编码	184,966	39%
Huffman 编码	172,346	43%

1. 固定长度操作码

- 定长定域：
  - IBM公司的大中型机：最左边8位为操作码
  - Intel公司的Intanium处理机：14位定长操作码
  - 许多RISC处理机采用定长操作码
- 主要优点：规整，译码简单
- 主要缺点：浪费信息量（操作码的总长位数增加）

2. Huffman编码法

- 哈夫曼压缩概念：当各种事件发生的概率不均等时，采用优化技术，对发生概率最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），使表示（处理）的平均位数（时间）缩短
- 操作码的最短平均长度可通过如下公式计算：

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

p_i表示第i种操作码在程序中出现的概率

- 固定长编码相对于最优Huffman编码的信息冗余量：

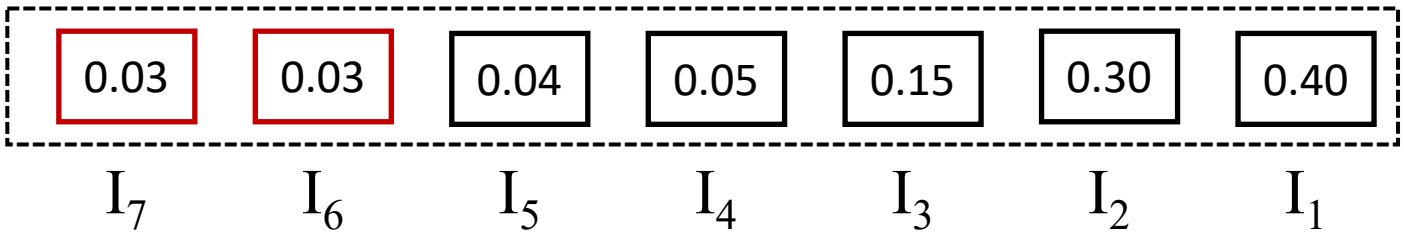
$$R = 1 - \frac{-\sum_{i=1}^{n} p_i \log_2 p_i}{\lceil \log_2 n \rceil}$$

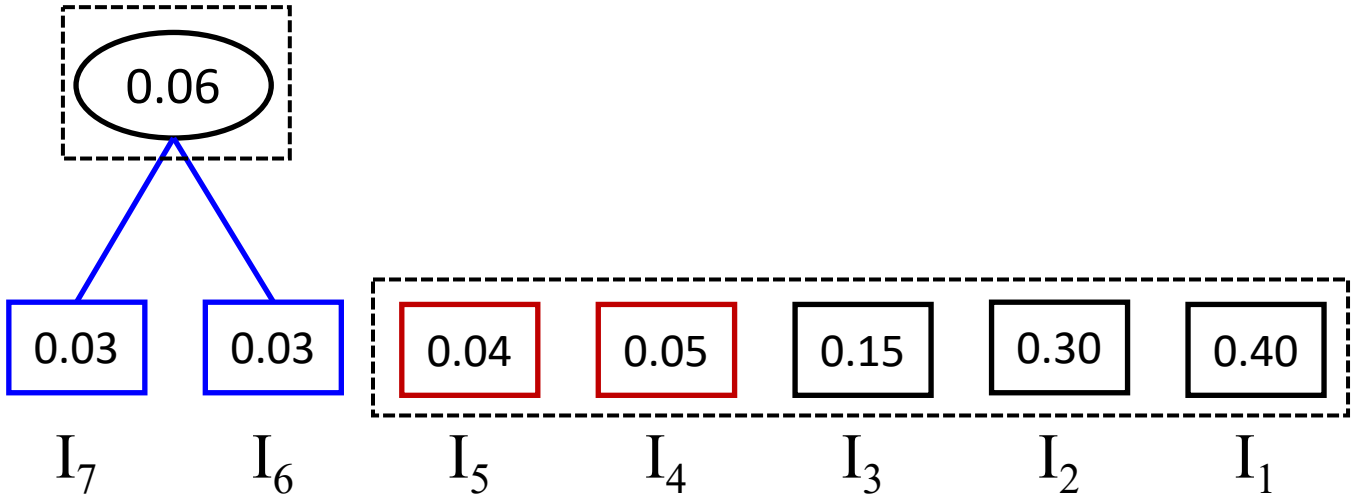
例：假设一台模型计算机共有7种不同的操作码，如果采用固定长操作码需要3位。已知各种操作码在程序中出现的概率如下表，计算采用Huffman编码法的操作码平均长度，并计算固定长操作码和Huffman操作码的信息冗余量。

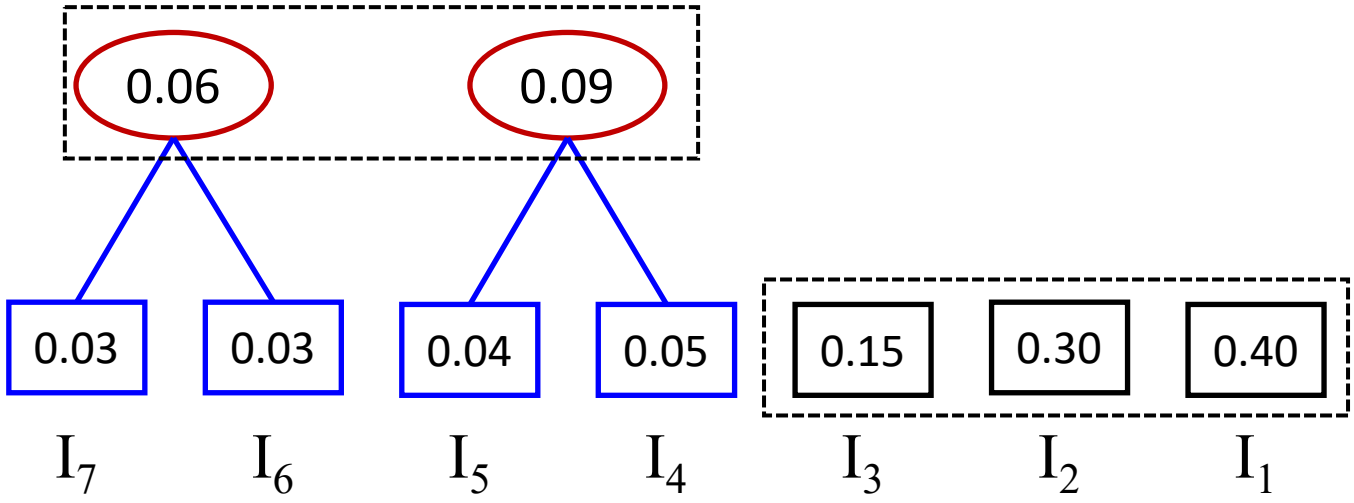
指令序号	I_1	I_2	I_3	I_4	I_5	I_6	I_7
出现的概率	0.40	0.30	0.15	0.05	0.04	0.03	0.03

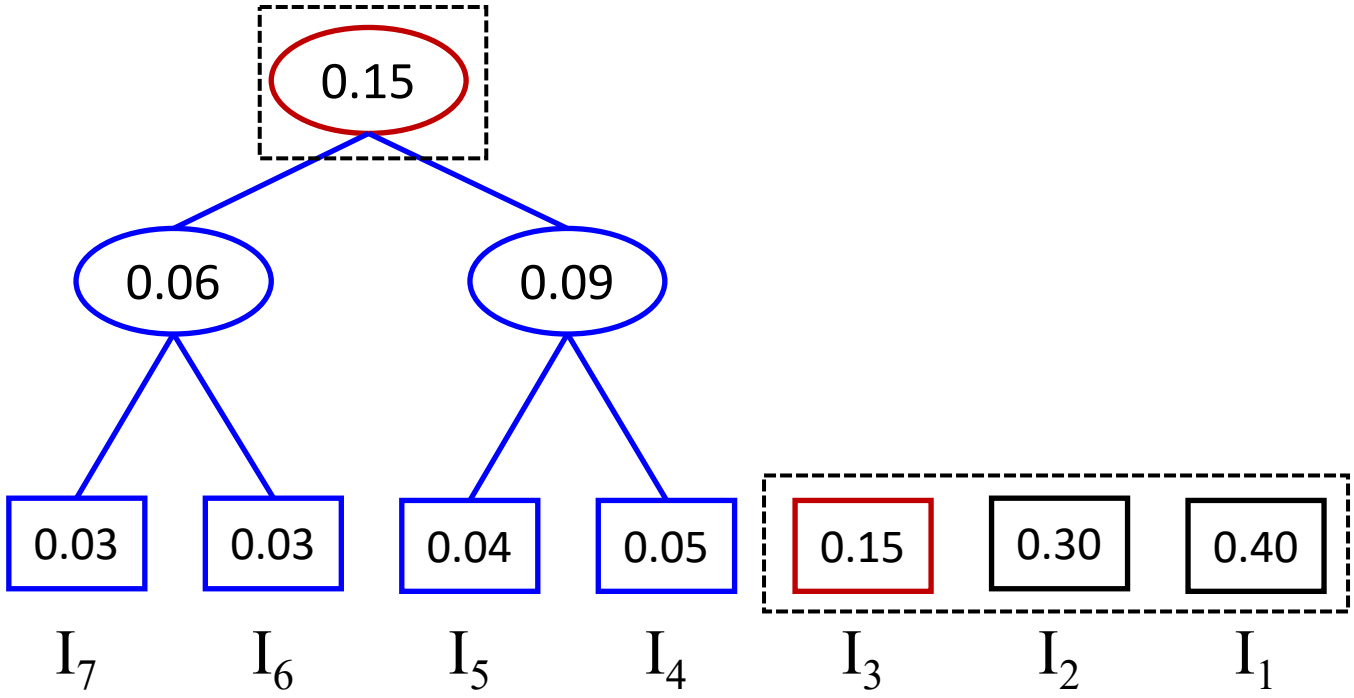
利用Huffman树进行操作码编码（又称最小概率合并法）

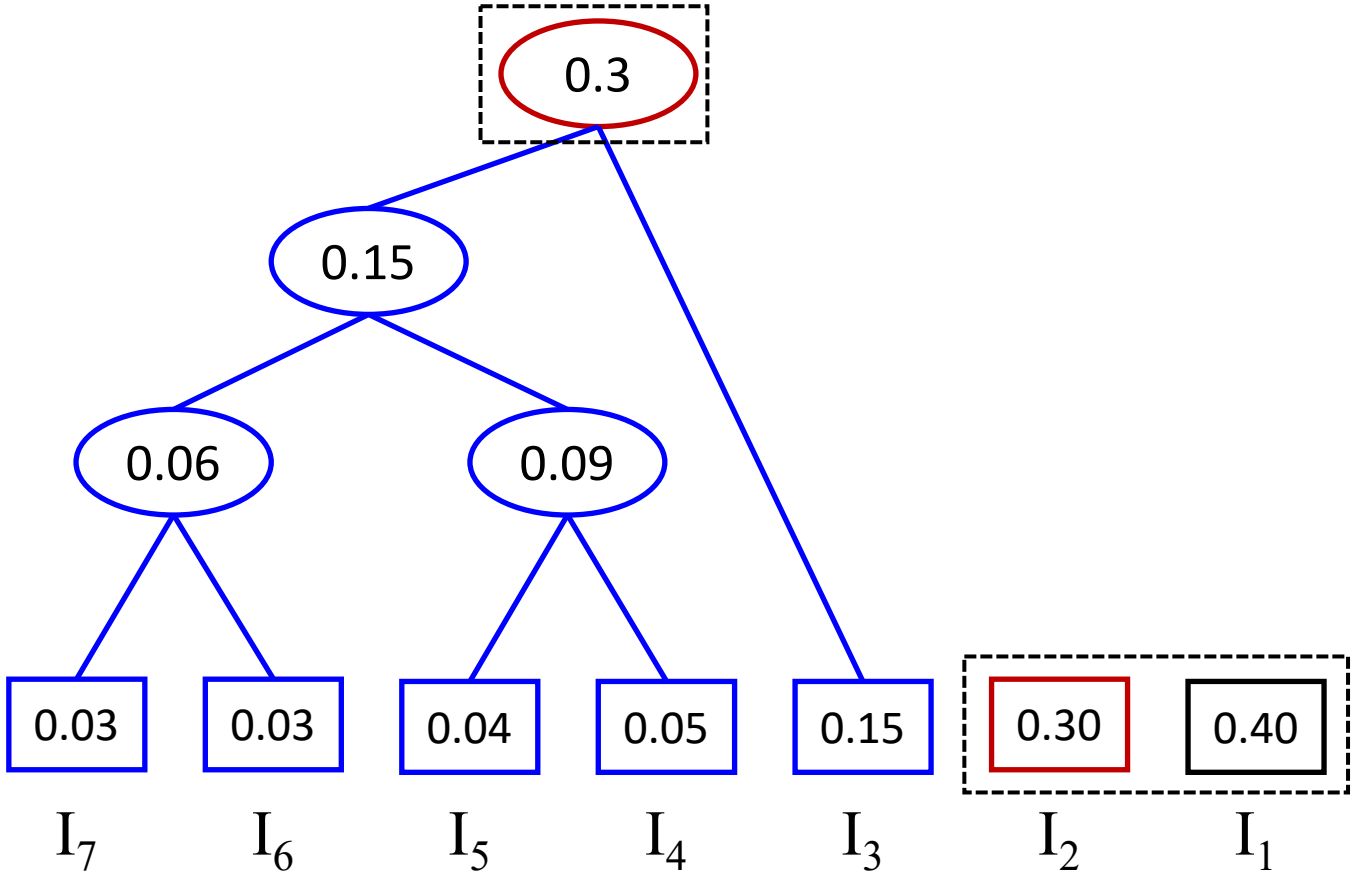
- 把所有指令按照操作码在程序中出现的概率大小，自左向右顺序排列。
- 选取两个概率最小的结点合并成一个概率值是二者之和的新结点，并把这个新结点与其它还没有合并的结点一起形成一个新的结点集合。
- 在新结点集合中选取两个概率最小的结点进行合并，如此继续进行下去，直至全部结点合并完毕。
- 最后得到的根结点的概率值为1。
- 每个新结点都有两个分支，分别用带有箭头的线表示，并分别用一位代码“0”和“1”标注。
- 从根结点开始，沿尖头所指方向寻找到达属于该指令概率结点的最短路径，把沿线所经过的代码排列起来就得到了这条指令的操作码编码。

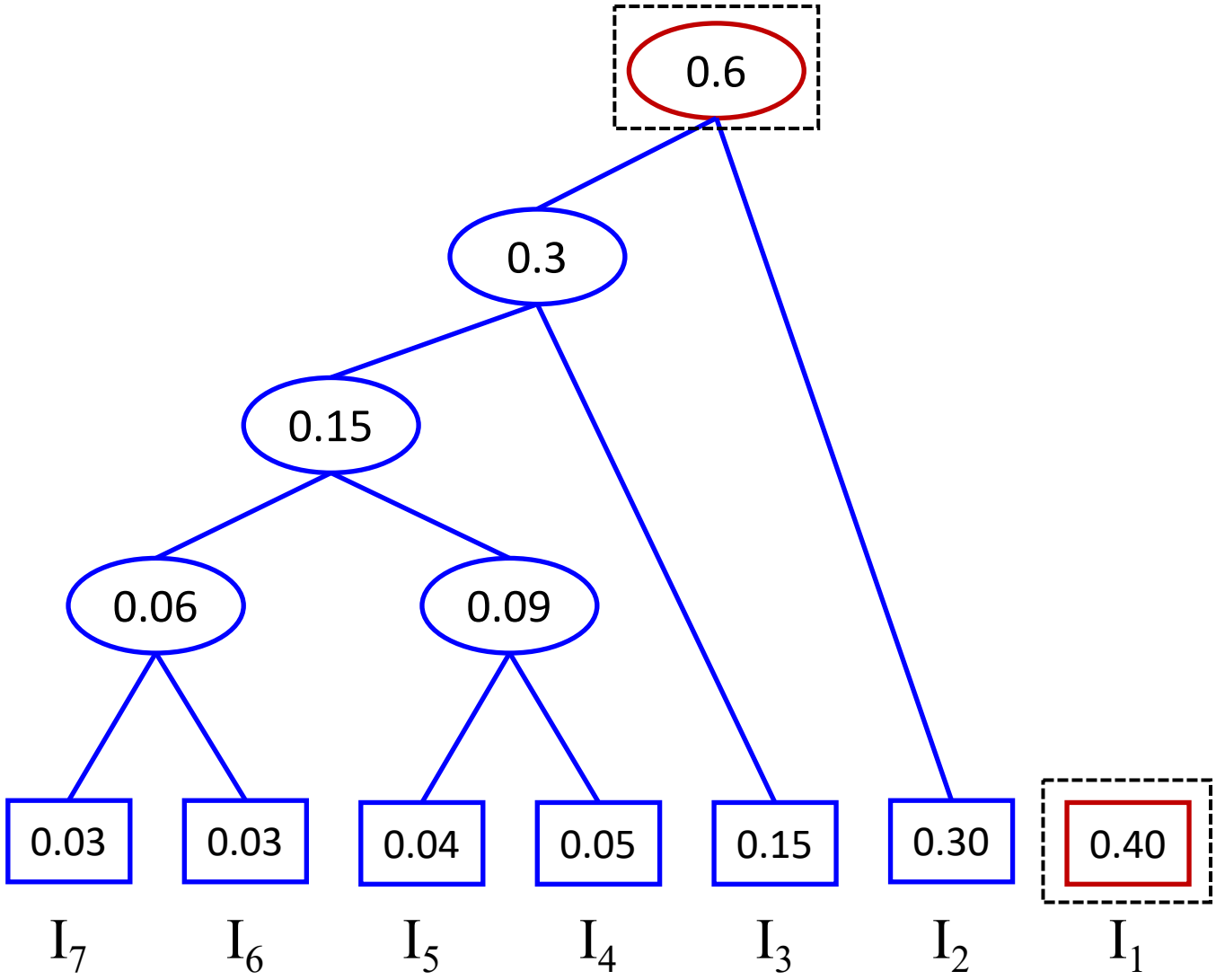

0.03
0.03
0.04
0.05
0.15
0.30
0.40
I_7
I_6
I_5
I_4
I_3
I_2
I_1

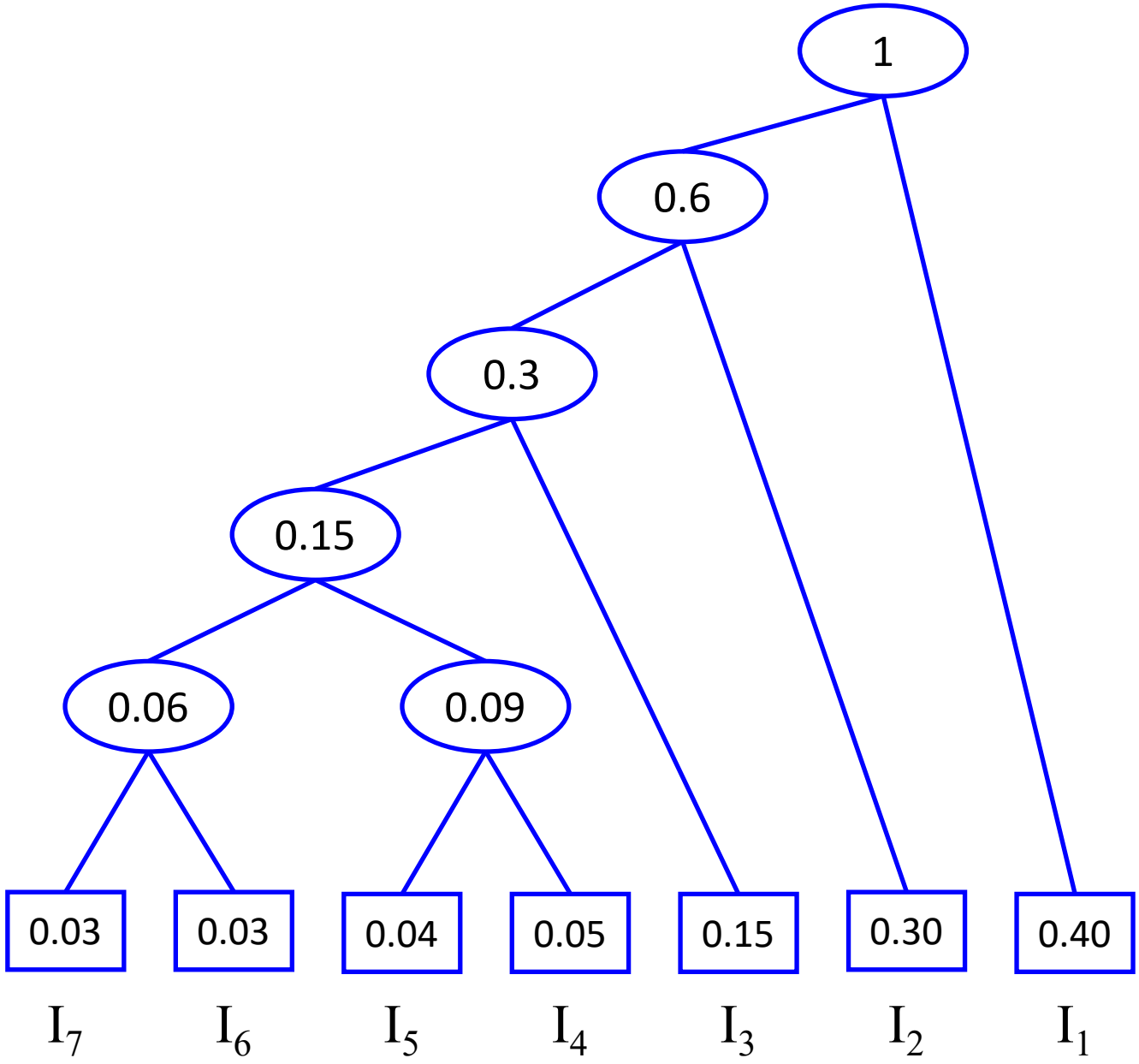

0.06
0.03
0.03
0.04
0.05
0.15
0.30
0.40
I7
I6
I5
I4
I3
I2
I1


0.06
0.09
0.03
0.03
0.04
0.05
0.15
0.30
0.40
I7
I6
I5
I4
I3
I2
I1


0.15
0.06
0.09
0.03
0.03
0.04
0.05
0.15
0.30
0.40
I7
I6
I5
I4
I3
I2
I1

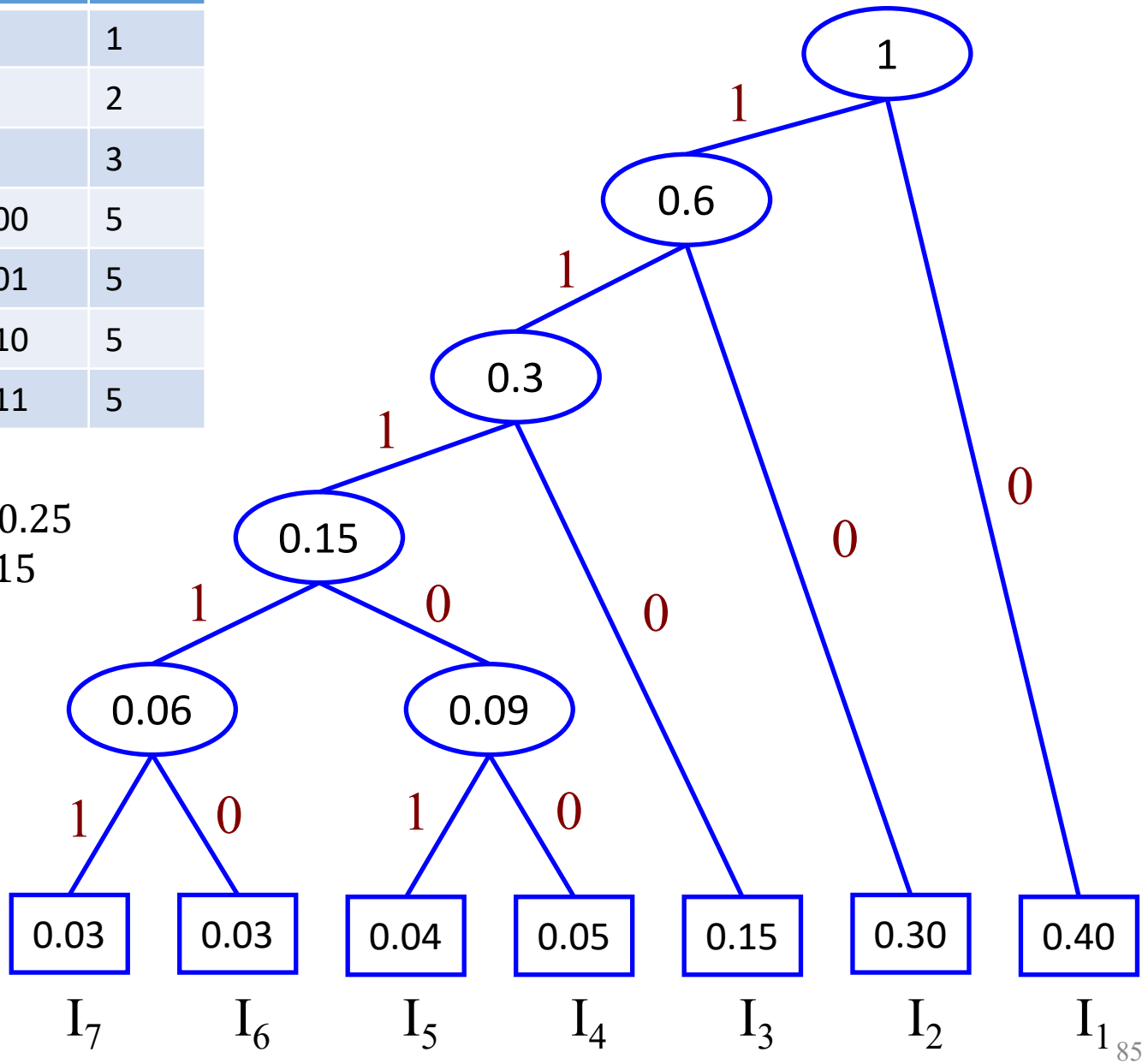

0.3
0.15
0.06
0.09
0.03
0.03
0.04
0.05
0.15
0.30
0.40
I7
I6
I5
I4
I3
I2
I1


0.6
0.3
0.15
0.06
0.09
0.03
0.03
0.04
0.05
0.15
0.30
0.40
I7
I6
I5
I4
I3
I2
I1


1
0.6
0.3
0.15
0.06
0.09
0.03
0.03
0.04
0.05
0.15
0.30
0.40
I7
I6
I5
I4
I3
I2
I1

指令	频度p_i	哈夫曼编码	码长
1	0.40	0	1
2	0.30	10	2
3	0.15	110	3
4	0.05	11100	5
5	0.04	11101	5
6	0.03	11110	5
7	0.03	11111	5

$$0.4 + 0.6 + 0.45 + 0.25 + 0.2 + 0.15 + 0.15 = 2.2$$



- 采用Huffman编码法的操作码平均长度为：

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

- 操作码的最短平均长度为：

$$H_{opt} = -\sum_{i=1}^{7} p_i \log_2 p_i = 2.17$$

- 采用3位固定长操作码的信息冗余量为：

$$R = 1 - \frac{H_{opt}}{\lceil \log_2 7 \rceil} = 1 - \frac{2.17}{3} \approx 27.7\%$$

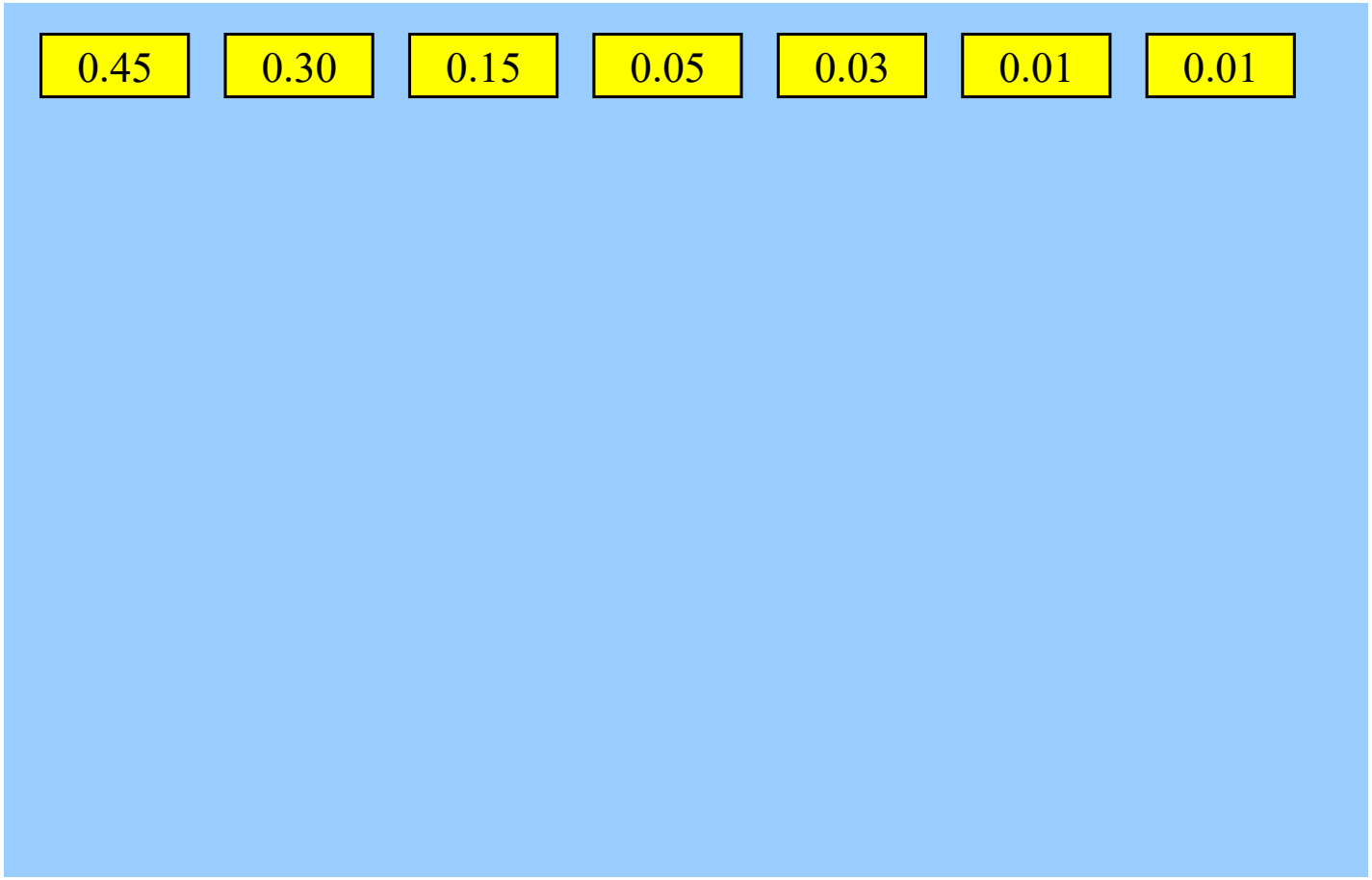
- Huffman编码法的信息冗余量仅为：

$$R = 1 - \frac{2.17}{2.20} \approx 1.36\%$$

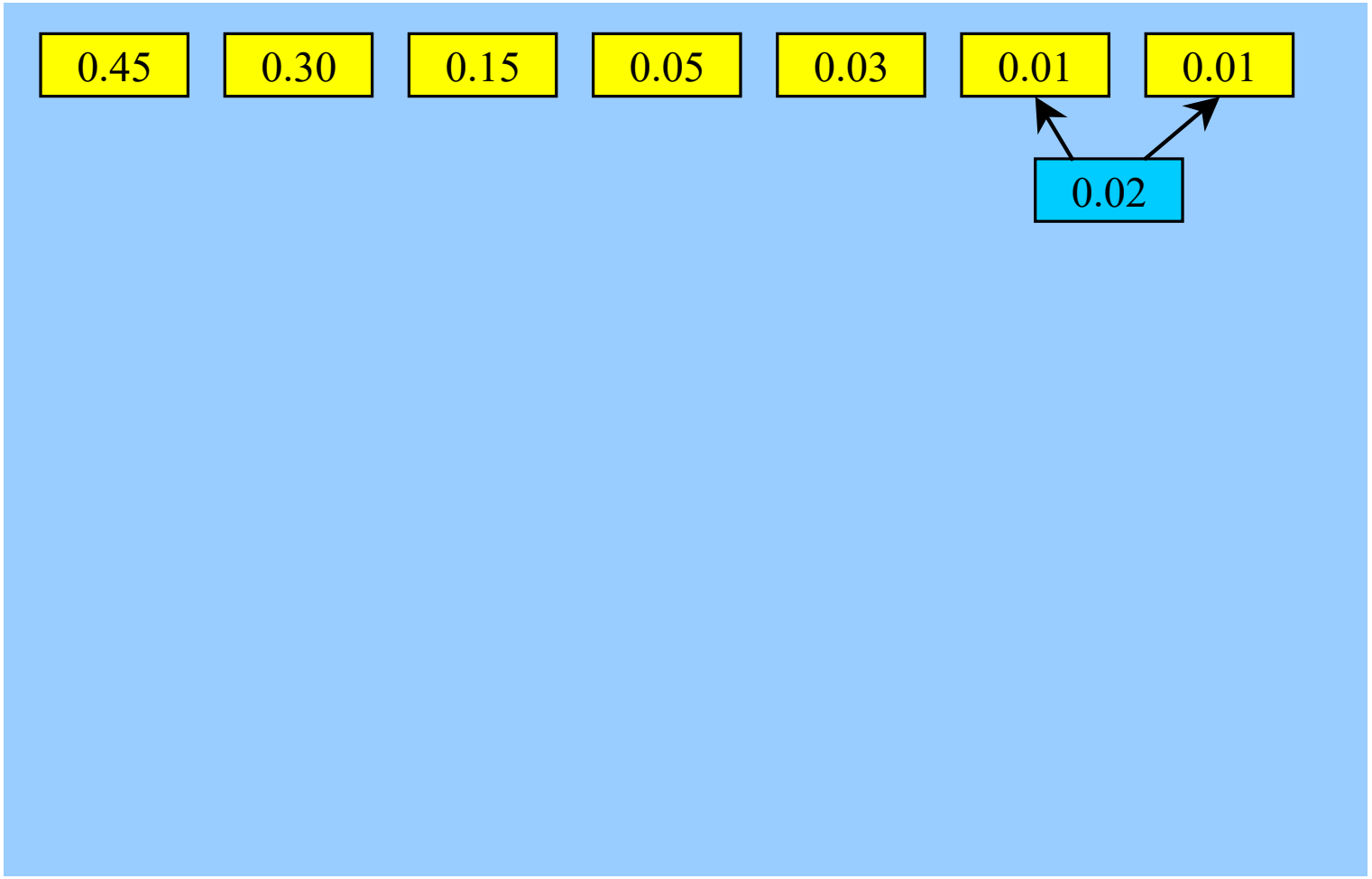
例：假设一台模型计算机共有7种不同的操作码，如果采用固定长操作码需要3位。已知各种操作码在程序中出现的概率如下表，计算采用Huffman编码法的操作码平均长度，并计算固定长操作码和Huffman操作码的信息冗余量。

指令序号	I_1	I_2	I_3	I_4	I_5	I_6	I_7
出现的概率	0.45	0.30	0.15	0.05	0.03	0.01	0.01

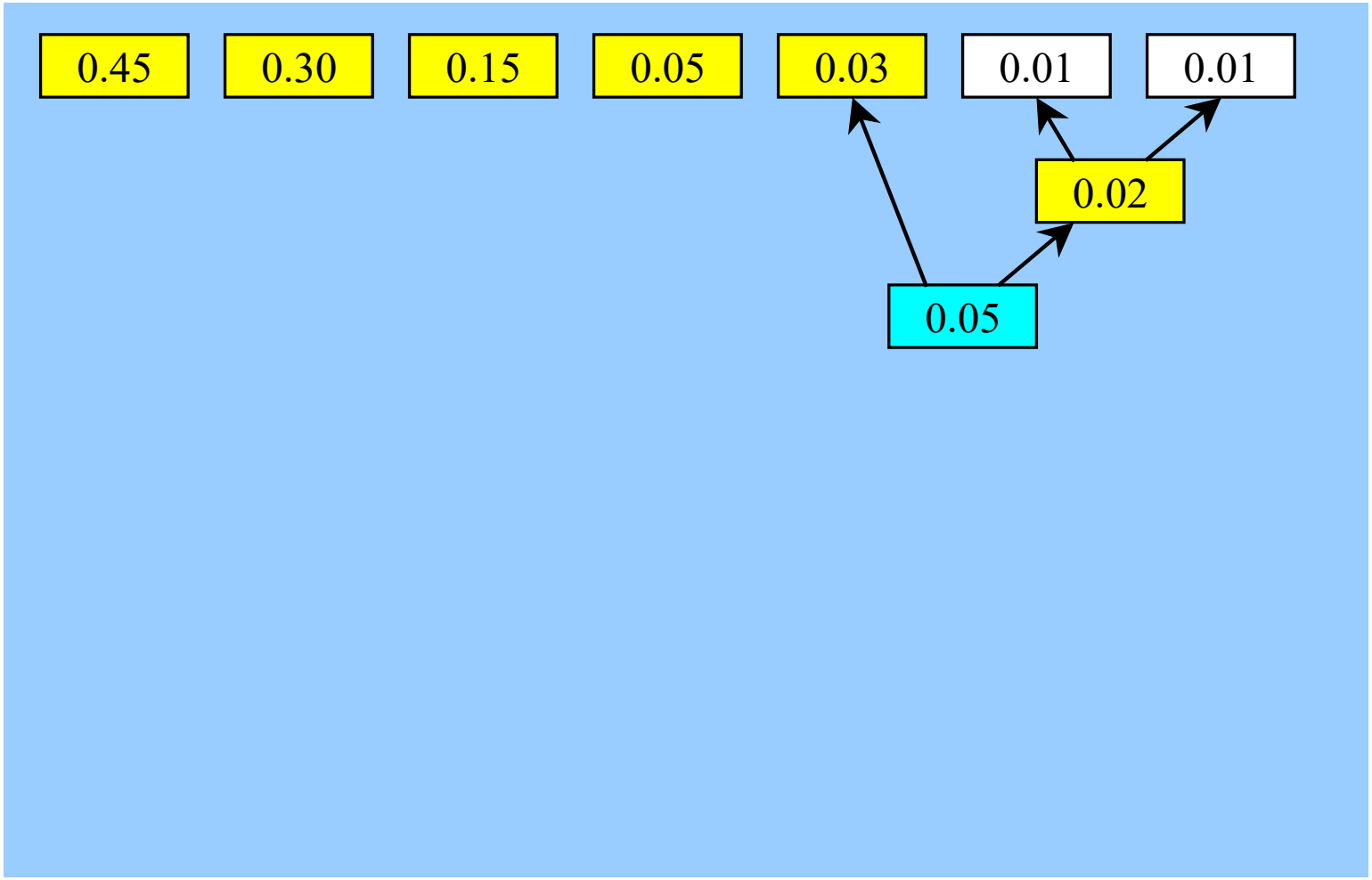
利用Huffman树进行操作码编码



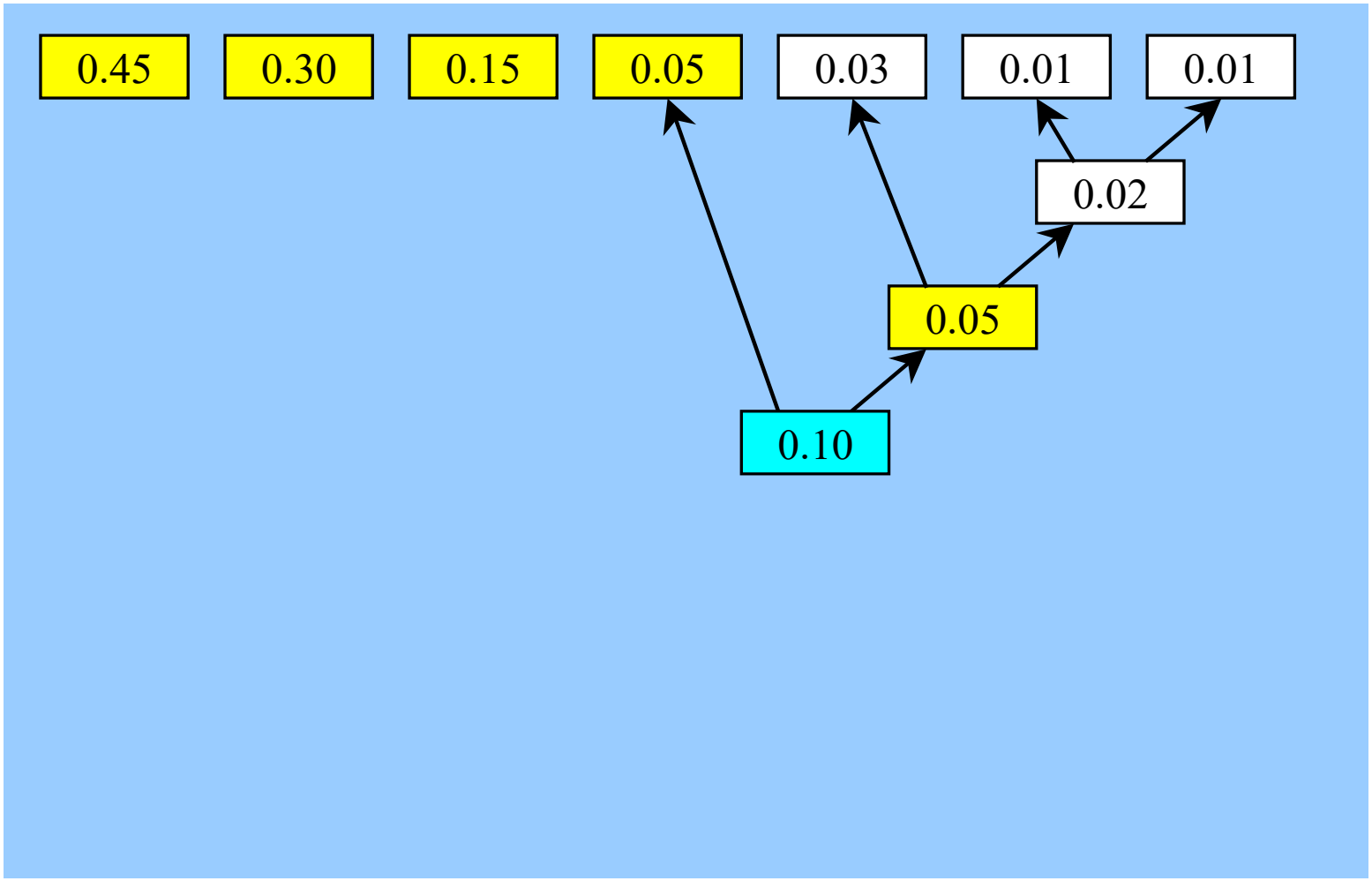
利用Huffman树进行操作码编码



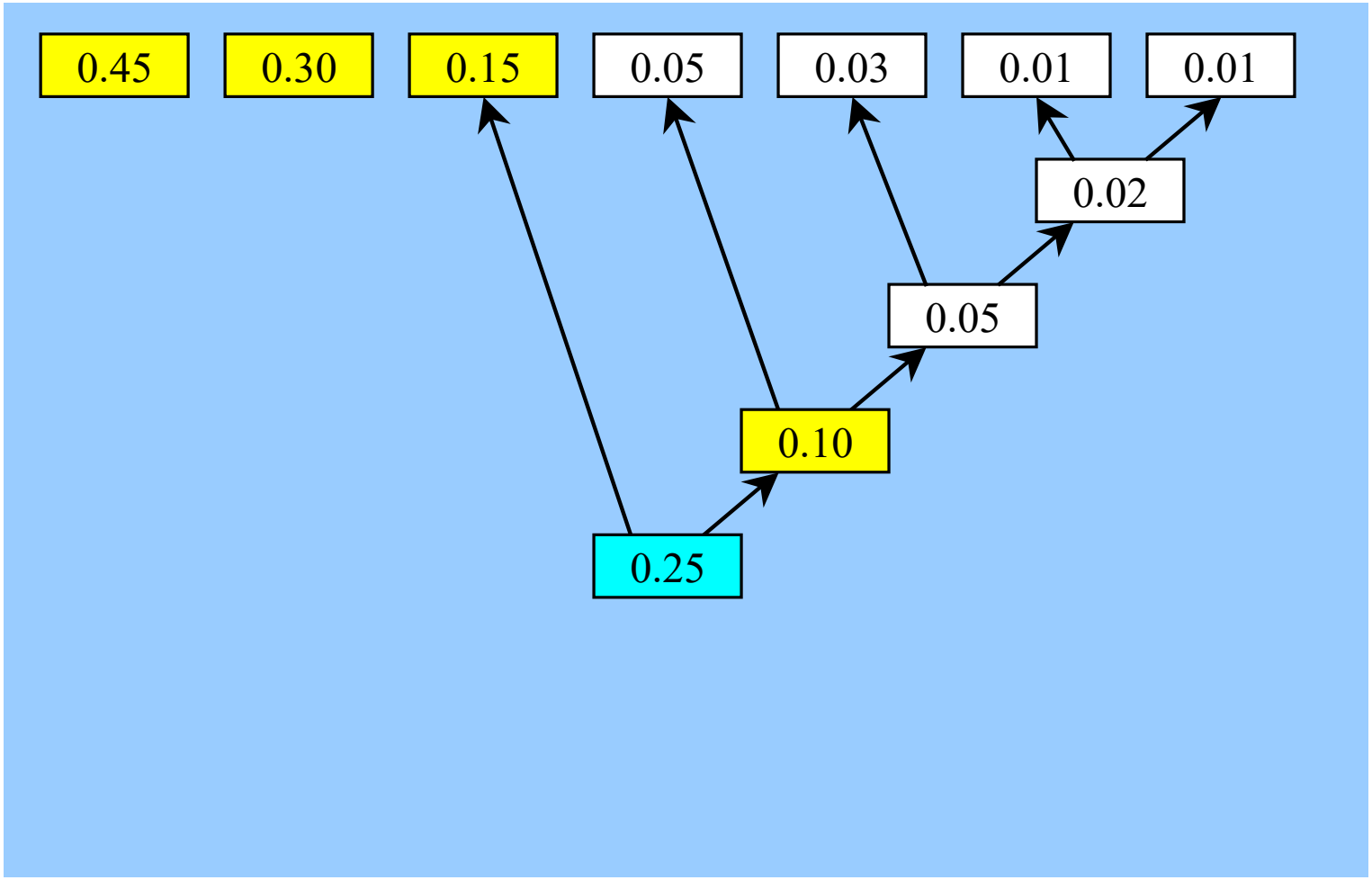
利用Huffman树进行操作码编码



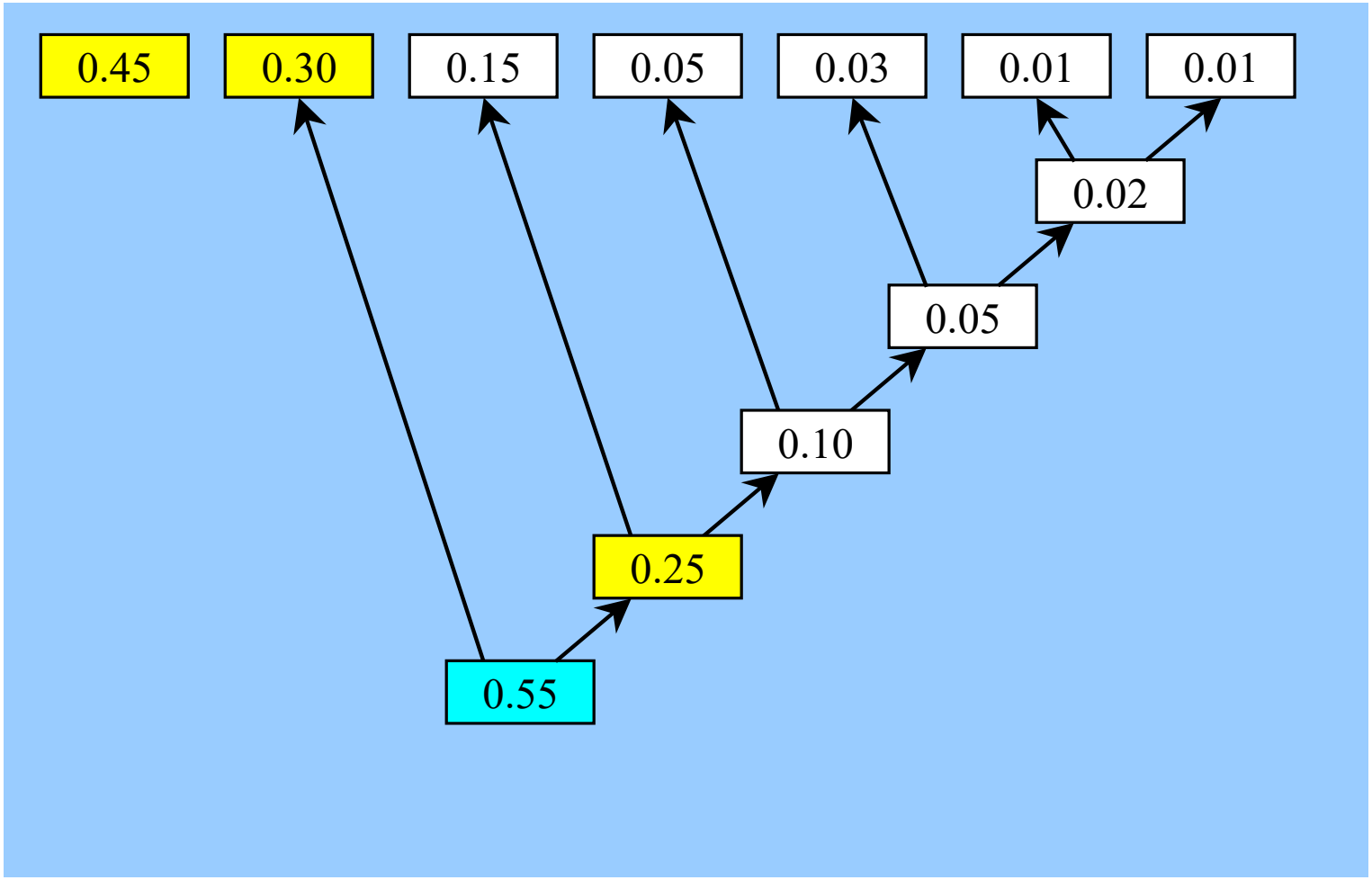
利用Huffman树进行操作码编码



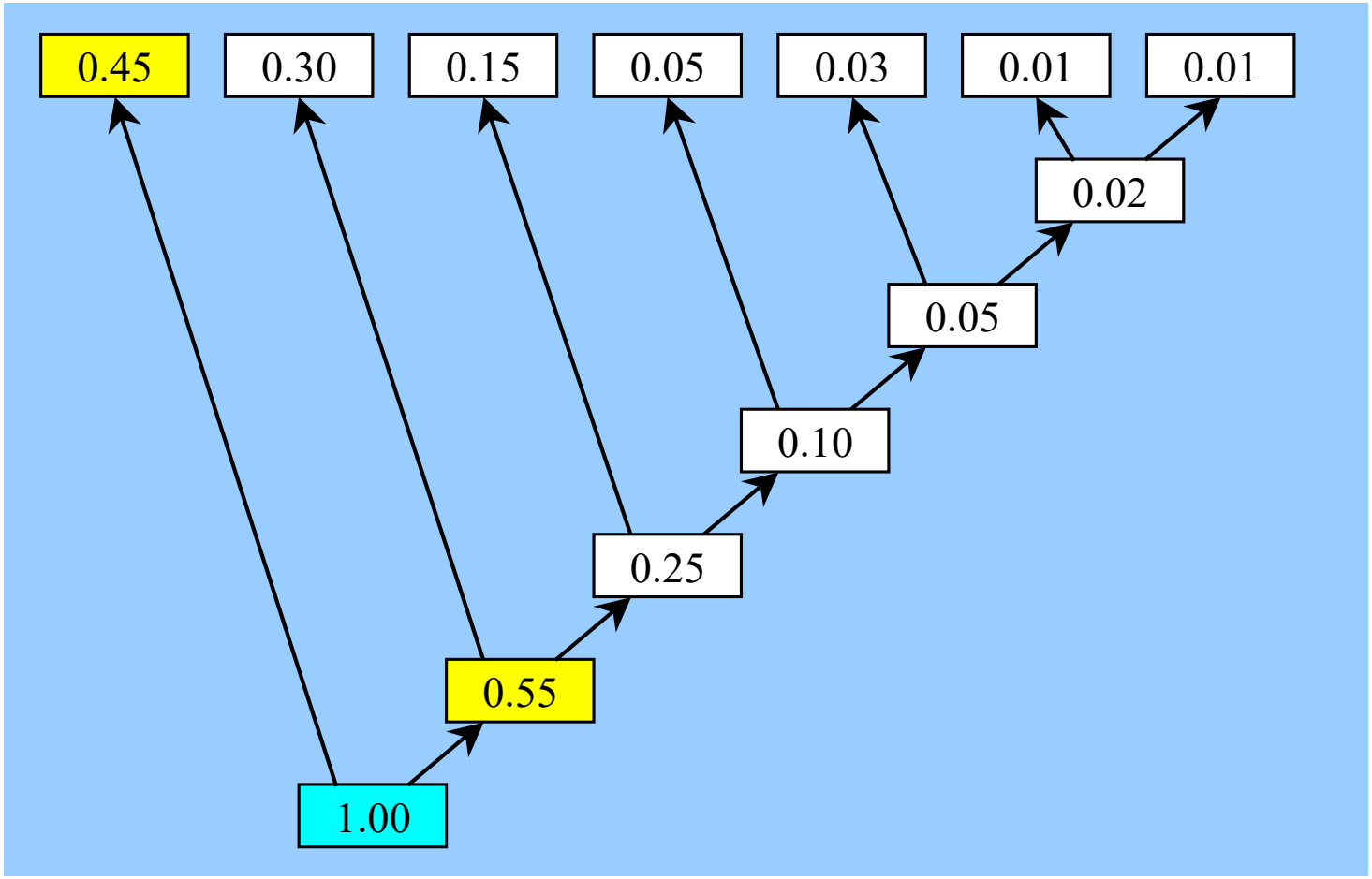
利用Huffman树进行操作码编码



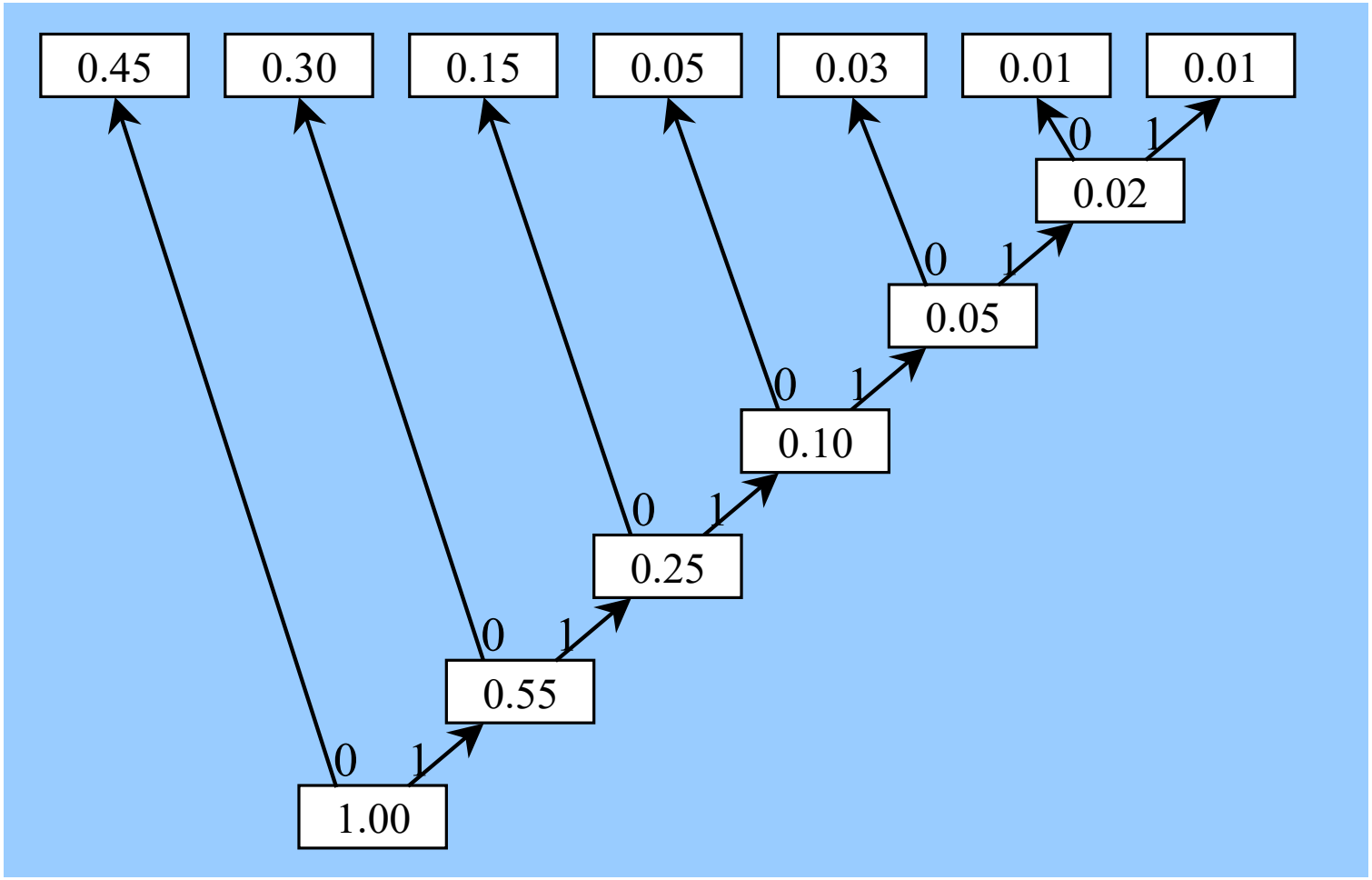
利用Huffman树进行操作码编码



利用Huffman树进行操作码编码



利用Huffman树进行操作码编码



Huffman操作码编码

指令序号	出现的概率	Huffman 编码法	操作码长度
I_1	0.45	0	1 位
I_2	0.30	1 0	2 位
I_3	0.15	1 1 0	3 位
I_4	0.05	1 1 1 0	4 位
I_5	0.03	1 1 1 1 0	5 位
I_6	0.01	1 1 1 1 1 0	6 位
I_7	0.01	1 1 1 1 1 1	6 位

解：采用Huffman编码法的操作码平均长度为：

$$H = \sum_{i=1}^{7} p_i \cdot l_i = 1.97$$

$0.45 + 0.6 + 0.45 + 0.2 + 0.15 + 0.06 + 0.06 = 1.97$

操作码的最短平均长度为：

$$H_{opt} = -\sum_{i=1}^{7} p_i \log_2 p_i = 1.95$$

- 采用3位固定长操作码的信息冗余量为：

$$R = 1 - \frac{H}{\lceil \log_2 7 \rceil} = 1 - \frac{1.95}{3} = 35\%$$

- Huffman编码法的信息冗余量仅为：

$$R = 1 - \frac{1.95}{1.97} \approx 1.0\%$$

与3位固定长操作码的信息冗余量35%相比要小得多

3. 扩展编码法

- Huffman操作码的主要缺点：
  - 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

指令序号	出现的概率	Huffman 编码法	操作码长度
I_1	0.45	0	1 位
I_2	0.30	1 0	2 位
I_3	0.15	1 1 0	3 位
I_4	0.05	1 1 1 0	4 位
I_5	0.03	1 1 1 1 0	5 位
I_6	0.01	1 1 1 1 1 0	6 位
I_7	0.01	1 1 1 1 1 1	6 位

- 扩展编码法：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，操作码的长度不是定长的，但是只有有限集中码长。仍然采用高概率指令用短码、低概率指令用长码的哈夫曼编码思想

指令	频度p_i	哈夫曼编码	OP长度	扩展操作码	OP长度
1	0.40	0	1	00	2
2	0.30	10	2	01	2
3	0.15	110	3	10	2
4	0.05	11100	5	1100	4
5	0.04	11101	5	1101	4
6	0.03	11110	5	1110	4
7	0.03	11111	5	1111	4

- 哈夫曼编码：平均长度=2.2，信息冗余度1.36%
- 扩展操作码：平均长度=2.3，信息冗余度5.65%

$$0.8 + 0.6 + 0.3 + 0.2 + 0.16 + 0.12 + 0.12 = 2.3$$

$$1 - \frac{2.17}{2.3} \approx 5.65\%$$

- 等长扩展法：4-8-12扩展法，3-6-9扩展法
- 4-8-12扩展法：不同的扩展编码适应于不同的指令分布

操作码 4-8-12 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 15 种

等长 15/15/15……扩展法

保留1111

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

保留1xxx

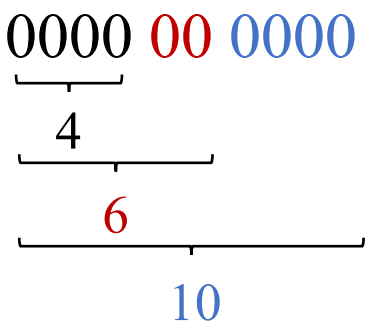
前缀数：$2^4-8=8$

前缀数：

$2^8-64=64$

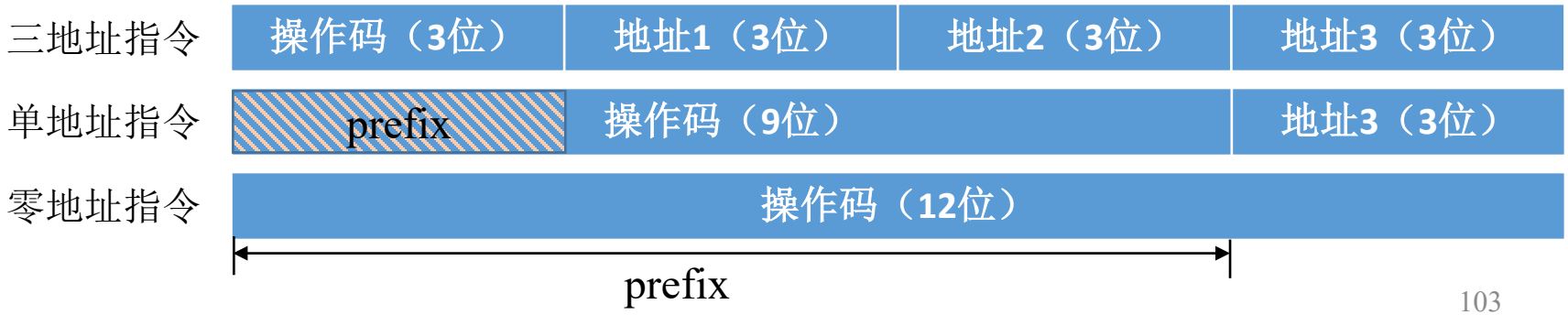
不等长操作码扩展编码法(4-6-10 扩展编码)

编码方法	各种不同长度操作码的指令			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



若某机有要求：三地址指令4条，单地址指令255条，零地址指令16条，设指令字长为12位，每个地址码长为3位。能否以扩展操作码为其编码？如果单地址指令改为254条呢?

- 三地址指令操作码最短，零地址指令操作码最长
- 三地址指令操作码为3位，可表示8个码，其中4个码（如000、001、010、011）用来表示4条三地址指令，剩余4个码（如100、101、110、111）用来扩展。
- 若操作码9位，利用剩余4个扩展码可以最多可以表示256个操作码。
- 若单地址指令用掉其中255个码，则无法表示剩余的16条零地址指令。



2.3.3 指令字格式的优化

1) 采用扩展码，并根据指令的频度分布选择合适的编码方式，以缩短操作码平均码长

2) 采用多种寻址方式（如基址、变址、相对、寄存器、寄存器间接、段式存放、隐式指明等），以缩短地址码长度，并在有限的地址长度内提供更多的地址信息

3) 采用多种地址制，增强指令功能、从宏观上缩短指令长度、并加快指令执行速度

4) 在同种地址制内采用多种地址形式，如寄存器-寄存器、寄存器-主存、主存-主存等，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组配

5) 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度

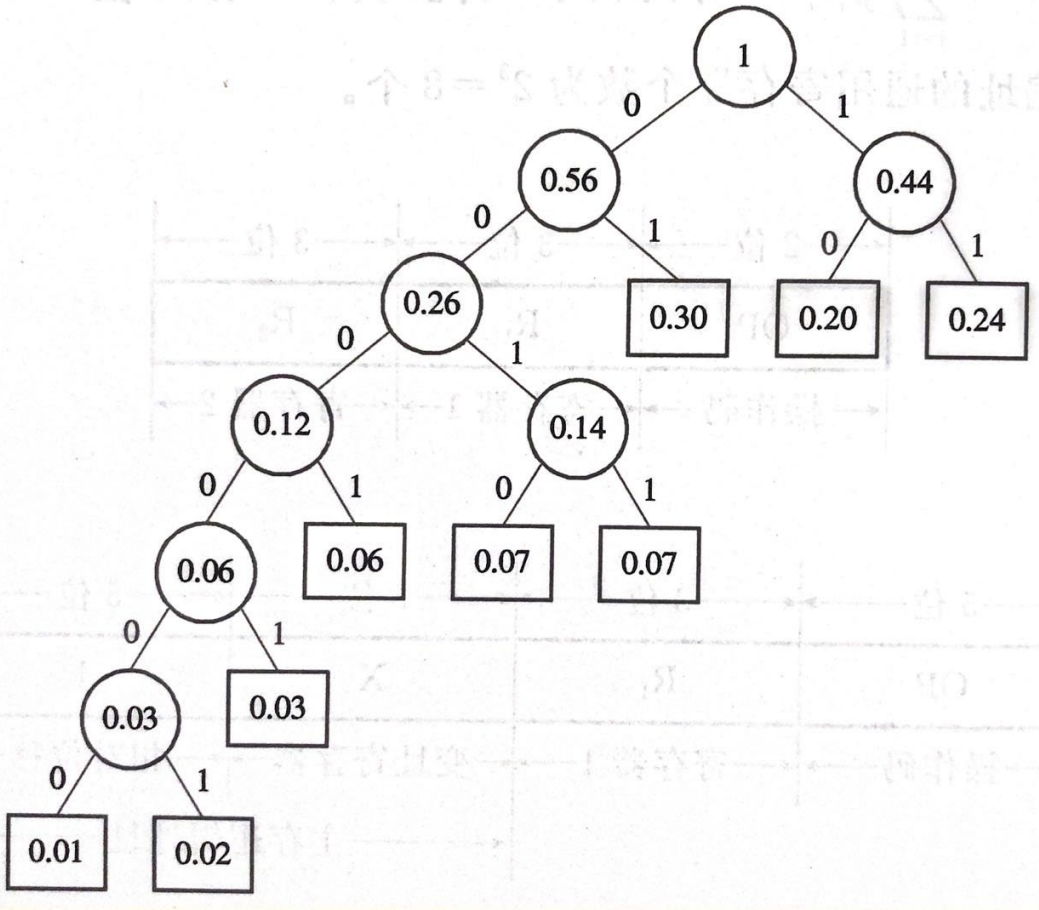
某模型机9条指令的使用频度如下表所示。要求有两种指令字长，都按双操作数指令格式编排，采用扩展操作码，并限制只能有两种操作码码长。设该机有8个通用寄存器，主存为16位宽，按字节编址，采用按整数边界存储，任何指令都在一个主存周期中取得，短指令为寄存器-寄存器型，长指令为寄存器-主存型，主存地址应能使用通用寄存器变址寻址。

3位

16位

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%

0.01(STP) **0.02**(SHR) **0.03**(CIL) **0.06**(JOM) **0.07**(JMP) **0.07**(STO) **0.2**(CLA) **0.24**(SUB) **0.3**(ADD)


1
0
1
0.56
0.44
0
1
0
1
0.26
0.30
0.20
0.24
0
1
0.12
0.14
0
1
0
1
0.06
0.06
0.07
0.07
0
1
0.03
0.03
0
1
0.01
0.02

指令	频度	哈夫曼编码	扩展编码
ADD（加）	30%	01	00
SUB（减）	24%	11	01
CLA（清加）	20%	10	10
JOM（按负转移）	6%	0001	11000
STO（存）	7%	0011	11001
JMP（转移）	7%	0010	11010
SHR（右移）	2%	000001	11011
CIL（循环左移）	3%	00001	11100
STP（停机）	1%	000000	11101

- 哈夫曼编码平均码长：2.61
- 2-5扩展编码平均码长：2.78

操作码（2位）	寄存器1（3位）	寄存器1（3位）

操作码（5位）	寄存器1（3位）	变址寄存器（3位）	相对位移（5位）

2. 一台模型机共有 7 条指令，各指令的使用频度分别为 35%、25%、20%、10%、5%、3%、2%，有 8 个通用数据寄存器，2 个变址寄存器。

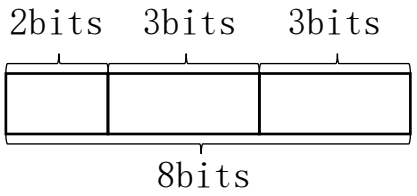
1） 要求操作码的平均长度最短，请设计操作码的编码，并计算所设计操作码的平均长度

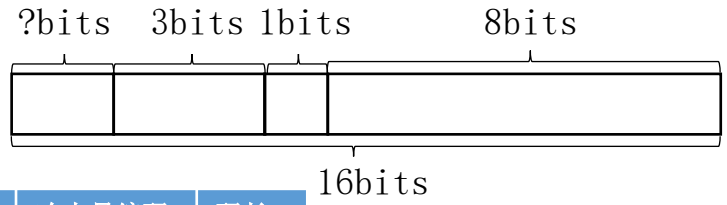
2） 设计 8 位字长的寄存器-寄存器型指令 3 条、16 位字长的寄存器-存储器型变址寻址方式指令 4 条，变址范围不小于正、负 127。请设计指令格式，并给出各字段的长度和操作码的编码。

2. 一台模型机共有 7 条指令，各指令的使用频度分别为 35%、25%、20%、10%、5%、3%、2%，有 8 个通用数据寄存器，2 个变址寄存器。

1） 要求操作码的平均长度最短，请设计操作码的编码，并计算所设计操作码的平均长度

2） 设计 8 位字长的寄存器-寄存器型指令 3 条、16 位字长的寄存器-存储器型变址寻址方式指令 4 条，变址范围不小于正、负 127。请设计指令格式，并给出各字段的长度和操作码的编码。





频度	哈夫曼编码	码长
35%	00	2
25%	01	2
20%	10	2
10%	110	3
5%	1110	4
3%	11110	5
2%	11111	5

哈夫曼编码平均码长：2.61

频度	哈夫曼编码	码长
35%	00	2
25%	01	2
20%	10	2
10%	1100	4
5%	1101	4
3%	1110	4
2%	1111	4

2.4 指令系统的发展和改进

2.4.1 两种途径和方法

2.4.2 按CISC方向发展和改进指令系统

2.4.3 按RISC方向发展和改进指令系统

2.4.4 RISC关键技术

2.4.1 两种途径和方向

➢两种途径和方向

- 按CISC方向：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
- 按RISC方向：通过减少指令种类和简化指令功能来降低硬件设计的复杂度，提高指令执行速度

2.4.2 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

2.4.3 按RISC方向发展和改进指令系统

1. CISC面临的问题

   1）20%与80%规律

   2）超大规模集成电路技术的发展

   3）软硬件分配问题

2. RISC的定义与特点

3. RISC思想的精华

- 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思想的计算机IBM 801
- 1986年，IBM正式推出采用RISC体系结构的工作站IBM RT PC
- CISC指令系统存在的问题：
  - 1979年，美国加洲伯克利分校 David Patterson 提出：

1. CISC面临的问题

1）20%与80%规律

- 在CISC中，大约20%的指令占据了80%的处理机执行时间
- 例如：8088处理机的指令种类大约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%的指令：使用频度只有8.9%,2.28%的处理机运行时间

Intel8088 处理机指令系统使用频度和执行时间统计
（C 语言编译程序和 PROLOG 解释程序）

使用频度				执行时间			
序号	指令	%	累计%	序号	指令	%	累%
1	MOV	24.85	24.85	1	IMUL	19.55	19.55
2	PUSH	10.36	35.21	2	MOV	17.44	36.99
3	CMP	10.28	45.49	3	PUSH	11.11	48.10
4	JMPcc	9.03	54.52	4	JMPcc	10.55	58.65
5	ADD	6.80	61.32	5	CMP	7.80	66.45
6	POP	4.14	65.46	6	CALL	7.27	73.72
7	RET	3.92	69.38	7	RET	4.85	78.57
8	CALL	3.89	73.27	8	ADD	3.27	81.84
9	JUMP	2.70	75.97	9	JMP	3.26	85.10
10	SUB	2.43	78.40	10	LES	2.83	87.93
11	INC	2.37	80.77	11	POP	2.61	90.54
12	LES	1.98	82.75	12	DEC	1.49	92.03
13	REPN	1.92	84.67	13	SUB	1.18	93.21
14	IMUL	1.69	86.36	14	XOR	1.04	94.25
15	DEC	1.37	87.73	15	INC	0.99	95.24
16	XOR	1.13	88.86	16	LOOPcc	0.64	95.88
17	REPNZ	0.78	89.64	17	LDS	0.64	96.52
18	CLD	0.54	90.18	18	CMPS	0.44	96.96
19	LOOPcc	0.52	90.70	19	MOVS	0.39	97.35
20	TEST	0.40	91.10	20	JCXZ	0.37	97.72

1. CISC面临的问题

2）超大规模集成电路（VLSI）技术的发展引起的问题

- VLSI工艺要求规整性
  - CISC为了实现大量复杂指令，控制逻辑极不规整
  - RISC控制逻辑简单，正好适应了VLSI工艺的要求
- 主存与控存的速度相当
  - 70年代之前，一般使用磁芯做主存储器，用半导体做控制存储器，两者速度相差5-10倍，因此CISC大量使用微程序技术以实现复杂的指令系统
  - 70年代之后，开始使用DRAM（动态随机存储器）做主存储器，使得主存和控存速度相当，因此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
- 由于VLSI的集成度迅速提高，使得生产单芯片处理机成为可能。单芯片处理机希望采用规整的硬布线逻辑，不希望采用微程序

1. CISC面临的问题

3）软硬件分配问题

- CISC中，通过增强指令系统的功能，简化了软件，增加了硬件复杂度
- 由于指令的复杂性，指令的执行时间必然加长，从而使整个程序的执行时间反而增加
- 1981年，Patterson等人研制了32位的RISC I微处理器，总共31种指令，3种数据类型，两种寻址方式，研制周期10个月，比当时最先进的MC68000和Z8002快3至4倍
- 1983年，又研制了RISC II，指令种类扩充到39种，单一变址寻址方式，通用寄存器138个

2. RISC的定义与特点

1) **减少指令和寻址方式种类**：指令系统只选择高频指令，减少指令数量，一般不超过100条；减少寻址方式，一般不超过两种

2) **固定指令格式**：精减指令格式限制在两种以内，并使全部指令都在相同的长度

3) **大多数指令在单周期内完成**：让所有指令都在一个周期内完成

4) **采用LOAD/STORE结构**：扩大通用寄存器数量（一般不少于32个），尽量减少访存，所有指令只有存（STORE）/取（LOAD）指令可以访存，其他指令只对寄存器操作

5) **硬布线逻辑**：大多数指令采用硬联逻辑，少数指令采用微程序实现，提高指令执行速度

6) **优化编译**：通过精减指令和优化设计编译程序，简单有效地支持高级语言实现

3 RISC思想的精华

- 减少CPI是RISC思想的精华
- 程序执行时间的计算公式：P＝I • CPI • T
  - P是执行这个程序所使用的总的时间；
  - I是这个程序所需执行的总的指令条数；
  - CPI(Cycles Per Instruction)是每条指令执行的平均周期
  - T是一个周期的时间长度。

CISC 与 RISC 的运算速度比较

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

- 同类问题的程序长度,RISC比CISC长30%~40%
- CPI，RISC比CISC少2倍~10倍
- RISC的速度要比CISC快3倍左右,关键是RISC的CPI减小了

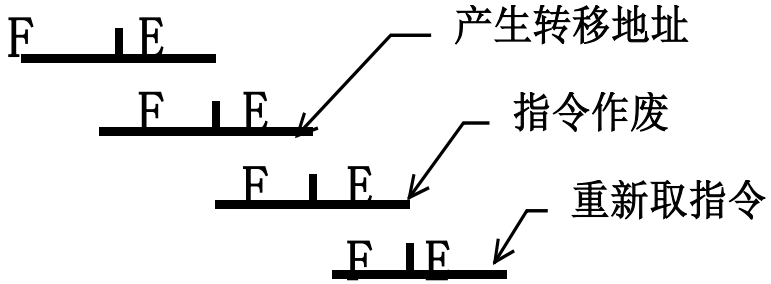
2.4.4 RISC的关键技术

1. 延时转移技术

- 为了使指令流水线不断流，在转移指令之后插入一条没有数据相关和控制相关的有效指令，而转移指令被延迟执行，这种技术称为延迟转移技术。
- 采用指令延迟转移技术时，指令序列的调整由编译器自动进行，用户不必干预。

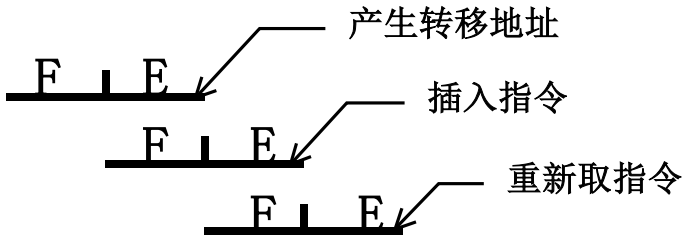
- 无条件转移指令的延迟执行

```
1:          ADD   R1, R2      1:
2:          JMP   NEXT2       2:
3: NEXT1:   SUB   R3, R4      3:
          ……                  n:
n: NEXT2:   MOVE  R4, A
```



因转移指令引起的流水线断流

```
1:          JMP   NEXT2       1:
2:          ADD   R1, R2      2:
3: NEXT1:   SUB   R3, R4      n:
          ……
n: NEXT2:   MOVE  R4, A
```



采用延时转移技术的指令流水线

- 条件转移指令的延迟执行

➢ 调整前的指令序列：

```
1：MOVE R1, R2
2：CMP  R3, R4      ;(R3)与(R4)比较
3：BEQ  NEXT        ;如果(R3)=(R4)则转移
.....
NEXT: MOVE R4, A
```

➢调整后的指令序列：

```
1：CMP R3, R4       ;(R3)与(R4)比较
2：BEQ NEXT         ;如果(R3)=(R4)则转移
3：MOVE R1,R2       ;被插入的指令
    .....
NEXT: MOVE  R4, A
```

2.4.4 RISC的关键技术

1. 延时转移技术

- 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没有数据相关
  - 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
- 如果找不到符合上述条件的指令，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
  - 插入1条指令成功的概率比较大，插入2条或2条以上指令成功的概率明显下降

2.4.4 RISC的关键技术

2. 指令取消技术

- 采用指令延时技术，经常找不到可以用来调整的指令，
- 可考虑采用另一种方法：指令取消技术
- 分为两种情况：

  (1) 向后转移（适用于循环程序）：成功则不取消

  (2) 向前转移(IF THEN)：成功则取消

（1）向后转移（适用于循环程序）

- 实现方法：循环体的第一条指令安放在两个位置，分别在循环体的前面和后面。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；否则取消循环体后面的指令

效果：

- 能够使指令流水线在绝大多数情况下不断流。
- 对于循环程序，由于绝大多数情况下，转移是成功的。
- 只有最后一次出循环时，转移不成功。

```
LOOP:  X X X
       Y Y Y
       ……
       Z Z Z
       COMP  R1, R2, LOOP
       W W W
```

(a) 调整前的程序

```
       X X X
LOOP:  Y Y Y
       ……
       Z Z Z
       COMP  R1, R2, LOOP
       X X X
       W W W
```

(b) 调整后的程序

（2）向前转移(IF THEN　)

实现方法：如果转移不成功，执行转移指令之后的下条指令，否则取消下条指令。

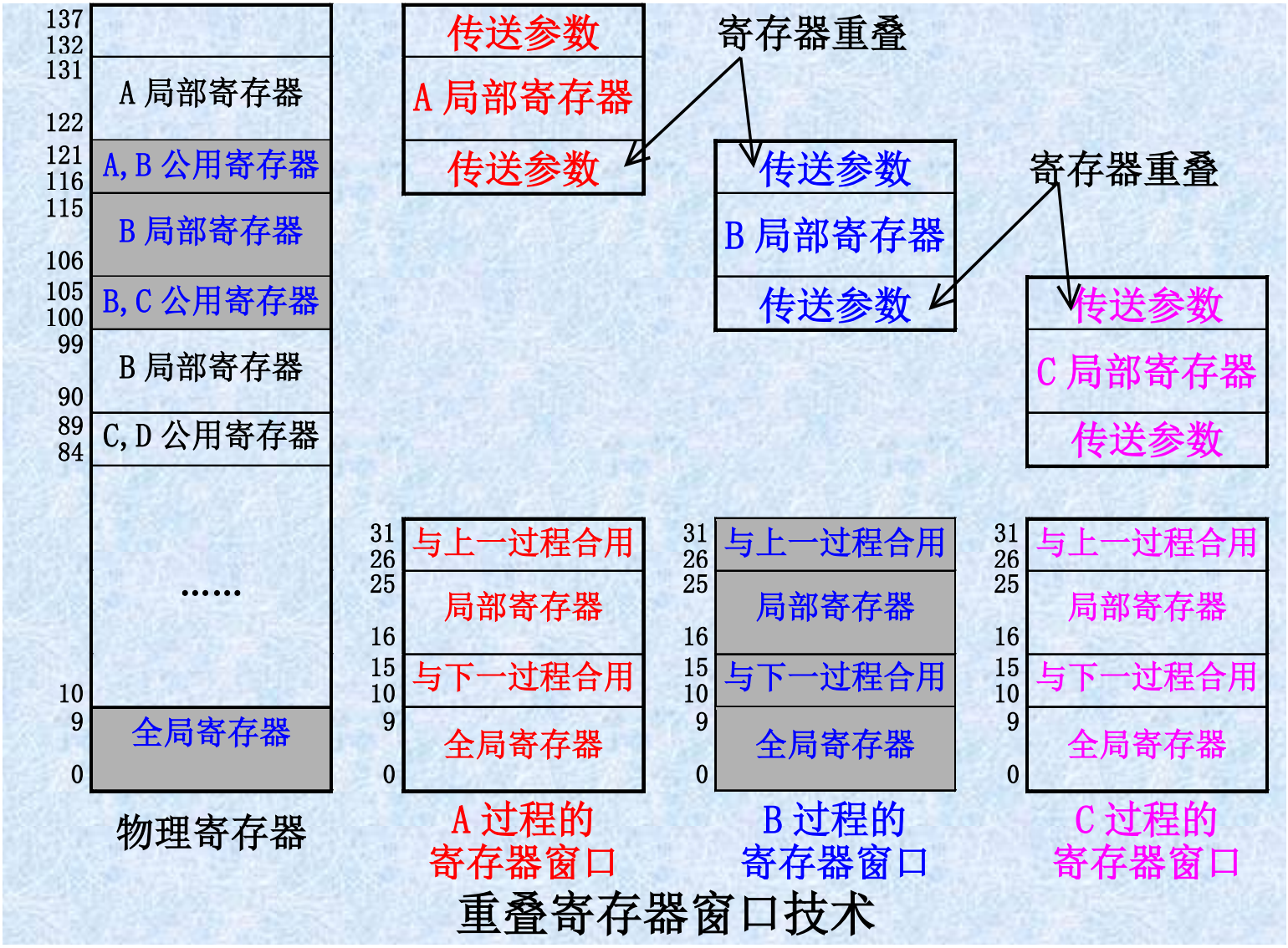
效果：转移成功与不成功的概率，通常各50％

主要优点：不必进行指令流调整

2.4.4 RISC的关键技术

3. 重叠寄存器窗口(Overlapping Register Window)技术

- 原因：在RISC中，子程序比CISC中多，因为传送参数而访问存储器的信息量很大
- 实现方法：设置一个数量比较大的寄存器堆，并把它划分成很多个窗口。在每个过程使用的几个窗口中：
  - 有一个窗口是与前一个过程共用
  - 有个窗口是与下一个过程共用


137
132
131
122
121
116
115
106
105
100
99
90
89
84
10
9
0
A 局部寄存器
A,B 公用寄存器
B 局部寄存器
B,C 公用寄存器
B 局部寄存器
C,D 公用寄存器
……
全局寄存器
物理寄存器
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
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
重叠寄存器窗口技术

寄存器窗口技术的效果

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

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(微秒)	访问存储器次数
VAX-11	5	26	10
PDP-11	19	22	15
MC68000	9	19	12
RISC II	6	2	0.2

注：Quicksort 程序的调用的次数多，深度不大，Puzzle 程序正好相反

2.4.4 RISC的关键技术

4. 指令流调整技术

目标：通过变量重新命名消除数据相关，提高流水线效率

例子：调整后的指令序列比原指令序列的执行速度快一倍

ADD R1,R2,R3; (R1)+(R2)→ R3	ADD R1,R2,R3
ADD R3,R4,R5; (R3)+(R4)→ R5	MUL R6,R7,R0
MUL R6,R7,R3; (R6)×(R7)→ R3	ADD R3,R4,R5
MUL R3,R8,R9; (R3) ×(R8)→ R9	MUL R0,R8,R9
调整前的指令序列	调整后的指令序列

2.4.4 RISC的关键技术

5. 采用高速缓冲存储器Cache

- 设置指令Cache和数据Cache分别存放指令和数据，保证向指令流水线不间断地输送指令和存取数据，提高流水线效率

6. 优化设计编译系统

- RISC由于使用了大量寄存器，因此编译程序必须尽量优化寄存器分配，提高其使用效率，减少访存次数。
- 优化编译器要做数据和控制相关性分析，要调整指令的执行序列，并与硬件相配合实现指令延迟技术和指令取消技术
- 由于CISC中一条指令在RISC中需要一段子程序来实现，所以要设计复杂的子程序库

本章重点：

1. 浮点数的表示方法及性质
2. 浮点数的设计方法
3. 浮点数的舍入方法
4. 自定义数据表示方法的原理
5. 指令格式的优化设计
6. RISC思想
7. RISC关键技术