第2章 指 令 系 统

李峰
fli@sdu.edu.cn
https://funglee.github.io

2.1 数据表示

2.2 寻址技术

2.3 指令格式的优化设计

2.4 指令系统的功能设计

2.5 RISC指令系统

2.6 VLIW指令系统

- 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。

本章主要内容有三大方面：

- 数据表示
- 寻址技术
- 指令系统设计

有三种类型的指令系统：

- CISC：复杂指令系统
- RISC：精简指令系统
- VLIW：超长指令字

指令系统设计：

- 指令的格式设计
- 指令系统的功能设计、
- 指令系统的性能评价

2.1 数据表示

2.1.1 数据表示与数据类型

2.1.2 浮点数的表示方法

2.1.3 浮点数格式设计

2.1.4 浮点数的舍入处理

2.1.5 警戒位的设置方法

2.1.6 自定义数据表示

2.1.1 数据表示与数据类型

- 数据类型：文件、图、表、树、阵列、队列、链表、栈、向量、串、实数、整数、布尔数、字符等
- 确定哪些数据类型用数据表示实现，是软件与硬件的取舍问题
- 数据表示的定义：
  - 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。
  - 例如：定点、逻辑、浮点、十进制、字符、字符串、堆栈和向量等

- 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

例：计算A＝A＋B，其中，A、B均为200×200的矩阵。分析采用向量数据表示的作用。

解：如果在没有向量数据表示的计算机上实现，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

如果有向量数据表示，只需要一条指令。减少访问主存（取指令）次数4×40,000次

2.1.2 浮点数的表示方法

1. 浮点数的表示方式

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

- 个数值：
  尾数m：数制(小数或整数)和码制(原码或补码)
  阶码e：整数, 移码(偏码、增码、余码)或补码
- 两两个基值：
  尾数基值r_m：2、4、8、16和10进制等
  阶码基值r_e： 通常为2进制
- 两个字长：长度和物理位置，均不包括符号位
  尾数长度p：尾数部分按基值计算的长度
  阶码长度q：阶码部分的二进制位数
- 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

- 浮点数表示方式的研究核心：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
- 在数据字长确定的前提下，研究各种浮点数表示方式的表数范围、表数精度和表数效率，以及它们之间的关系，目的是寻找到一种具有最大表数范围、最高表数精度和最优表数效率的浮点数表数方式

- 浮点数规格化
  - 目的：为了提高数据的表示精度和数据表示的唯一性
  - 基值为r_m的尾数的规格化：绝对值≥ r_m^{-1}
  - 当$r_m = 2$时，
    - 二进制原码的规格化数的表现形式：

      正数 01xxx…x

      负数 11xxx…x

    - 补码尾数的规格化的表现形式（尾数的最高位与符号位相反）

      正数 01xxx…x

      负数 10xxx…x
  - 当$r_m = 2^k$时，补码表示的正尾数的最高的k位不全为0，负尾数的最高的k位不全为1

移码

- 以二进制8位数为例
  - 补码可以表示2^8=256个数

-128	-127	-2	-1	0	1	127
1000 0000	1000 0001	1111 1110	1111 1111	0000 0000	0000 0001	0111 1111

  - 向右移动128，即加上128

-128	-127	-2	-1	0	1	127
0000 0000	0000 0001	0111 1110	0111 1111	1000 0000	1000 0001	1111 1111

- 移码表示范围】

$$-r_e^q \sim r_e^q - 1$$

- **在浮点数表示中，有些参数是隐含的**

例：已知一个32位浮点数 C1C00000，无法知道它表示的是什么数值！给出6个参数：尾数用纯小数，原码表示，r_m=16，p=6；阶码用整数，移码表示，r_e=2，q=6；浮点数格式如下图。

解：1100 0001 1100 0000 0000 0000 0000 0000

$-12/16 \times 16^1 = -12.0$

- **一种浮点数的存储式**

1 位	1 位	*q* 位	*p* 位
m_f	e_f	e	m

注：m_f 为尾数的符号位，e_f 为阶码的符号位，e 为阶码的值，m 为尾数的值。

- **在浮点数表示中，有些参数是隐含的**

例：已知一个32位浮点数 C1C00000，无法知道它表示的是什么数值！给出6个参数：尾数用纯小数，原码表示，r_m=16，p=6；阶码用整数，移码表示，r_e=2，q=6；浮点数格式如下图。

解：1100 0001 1100 0000 0000 0000 0000 0000

$-12/16 \times 16^1 = -12.0$

- **一种浮点数的存储式**

1 位	1 位	*q* 位	*p* 位
m_f	e_f	e	m

注：m_f 为尾数的符号位，e_f 为阶码的符号位，e 为阶码的值，m 为尾数的值。

2. 浮点数的表数范围

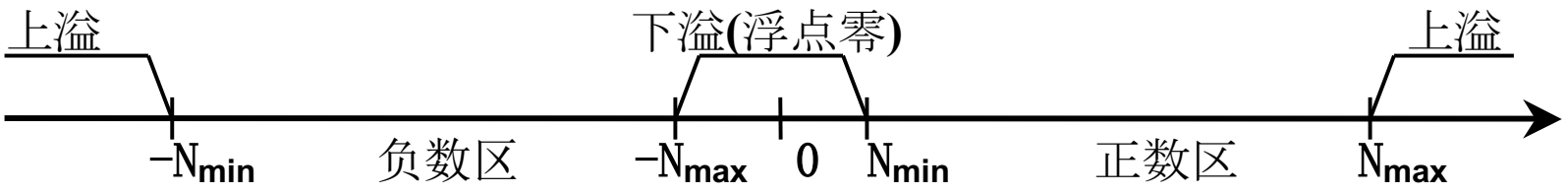
- 尾数为原码、小数，阶码用移码、整数时，规格化浮点数N的表数范围：

$$r_m^{-1} \cdot r_m^{-r_e^q} \le |N| \le \left(1 - r_m^{-p}\right) \cdot r_m^{r_e^q - 1}$$

- 尾数为补码，负数区间的表数范围为：

$$-r_m^{r_e^q - 1} \le N \le -\left(r_m^{-1} + r_m^{-p}\right) \cdot r_m^{-r_e^q}$$

- 浮点数在数轴上的分布情况



尾数用原码、纯小数时规格化浮点数的表数范围

表数范围	规格化尾数	阶码	规格化浮点数
最大正数（N_{max}）	$1-r_m^{-p}$	r_e^q-1	$\left(1-r_m^{-p}\right)\cdot r_m^{r_e^q-1}$
最小正数（N_{min}）	r_m^{-1}	----	$r_m^{-1}\cdot r_m^{-r_e^q}$
最大负数（$-N_{max}$）	$-r_m^{-1}$	----	$-r_m^{-1}\cdot r_m^{-r_e^q}$
最小负数（$-N_{min}$）	$-\left(1-r_m^{-p}\right)$	$-r_e^q$	$-\left(1-r_m^{-p}\right)\cdot r_m^{r_e^q-1}$

- 阶码用移码表示
  - 浮点0的范围：$|N| < r_m^{-1} \cdot r_m^{-r_e^q}$
  - 对于上例：$|N| < 2^{-129}$
  - 如果阶码用补码表示，浮点0为：

    0100 0000 0000 0000 0000 0000 0000 0000
  - 浮点0与机器0不同，判0困难。
- 阶码与补码的关系

十进制：	-128	-1	0	+127
补码：	1000 0000	1111 1111	0000 0000	0111 1111
移码：	0000 0000	0111 1111	1000 0000	1111 1111

例：尾数用原码、小数表示，阶码用移码、整数表示，p＝23，q＝7，$r_m=r_e=2$，求规格化浮点数N的表数范围。

解：规格化浮点数N的表数范围是：

$$\frac{1}{2}r^{-2^7} \le |N| \le \left(1-2^{-23}\right)\cdot 2^{2^7-1}$$

即：

$$2^{-129} \le |N| \le \left(1-2^{-23}\right)\cdot 2^{127}$$

1 位	1 位	7 位	23 位
m_f	e_f	e	m

注：m_f 为尾数的符号位，e_f 为阶码的符号位，e 为阶码的值，m 为尾数的值。

- 规格化最大正数：
  0111 1111 1111 1111 1111 1111 1111 1111
- 规格化最小正数：
  0000 0000 0100 0000 0000 0000 0000 0000
- 规格化最大负数：
  1000 0000 0100 0000 0000 0000 0000 0000
- 规格化最小负数：
  1111 1111 1111 1111 1111 1111 1111 1111

尾数用补码、纯小数时规格化浮点数的表数范围

表数范围	规格化尾数	阶码	规格化浮点数
最大正数（N_{max}）	$1-r_m^{-p}$	r_e^q-1	$\left(1-r_m^{-p}\right)\cdot r_m^{r_e^q-1}$
最小正数（N_{min}）	r_m^{-1}	----	$r_m^{-1}\cdot r_m^{-r_e^q}$
最大负数（$-N_{max}$）	$-(r_m^{-1}+r_m^{-p})$	----	$-(r_m^{-1}+r_m^{-p})\cdot r_m^{-r_e^q}$
最小负数（$-N_{min}$）	-1	$-r_e^q$	$-r_m^{r_e^q-1}$

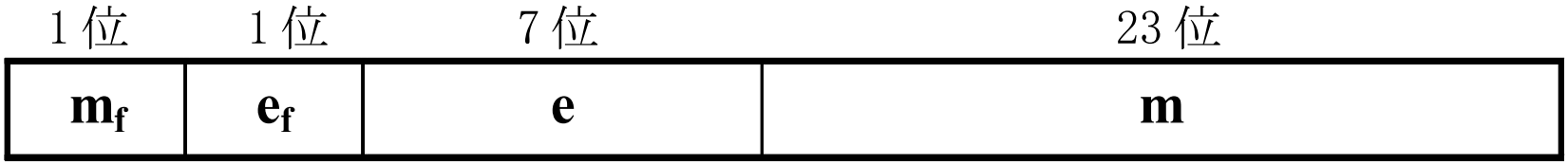
例：尾数用补码、小数表示，阶码用移码、整数表示，p＝23，q＝7，$r_m=r_e=2$，求规格化浮点数*N*的表数范围。

解：规格化浮点数*N*在正数区的表数范围：

$$\frac{1}{2}2^{-2^7} \le N \le \left(1-2^{-23}\right)\cdot 2^{2^7-1}$$

*N*在负数区的表数范围：

$$-2^{-127} \le N \le -\left(\frac{1}{2}+2^{-23}\right)\cdot 2^{-128}$$



注：m_f 为尾数的符号位，e_f 为阶码的符号位，e 为阶码的值，m 为尾数的值。

- **规格化最大正数：** 相同

  0111 1111 1111 1111 1111 1111 1111 1111

- **规格化最小正数：** 相同

  0000 0000 0100 0000 0000 0000 0000 0000

- **规格化最大负数：** 尾数 -0.100 0 ... 0 0001

  1000 0000 0011 1111 1111 1111 1111 1111

- **规格化最小负数：** 尾数 -1.0

  1111 1111 1000 0000 0000 0000 0000 0000

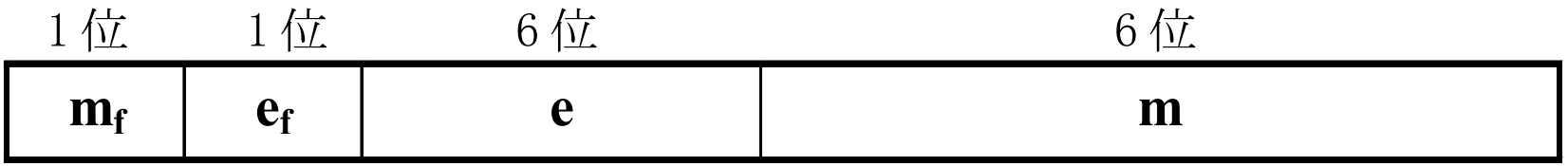
例：尾数用补码、小数表示，阶码用移码、整数表示，p＝6，q＝6，r_m＝16，r_e＝2，求规格化浮点数N表数范围。

解：规格化浮点数*N*在正数区间的表数范围：

$$16^{-65} \le N \le (1-16^{-6})\cdot 16^{63}$$

在负数区间的表数范围：

$$-16^{63} \le N \le -(\frac{1}{16}+16^{-6})\cdot 16^{-64}$$



注：m_f 为尾数的符号位，e_f 为阶码的符号位，e 为阶码的值，m 为尾数的值。

- **规格化最大正数：相同**

  0111 1111 1111 1111 1111 1111 1111 1111

- **规格化最小正数：相同**

  0000 0000 0001 0000 0000 0000 0000 0000

- **规格化最大负数：** 尾数 -0.0001 00... 0 0001

  1000 0000 1110 1111 1111 1111 1111 1111

- **规格化最小负数：** 尾数 -1.0

  1111 1111 0000 0000 0000 0000 0000 0000

- 尾数为原码、小数，阶码用移码、整数时，规格化浮点数N的表数范围：

$$r_m^{-1} \cdot r_m^{-r_e^q} \le |N| \le \left(1 - r_m^{-p}\right) \cdot r_m^{r_e^q - 1}$$

- 尾数为补码，负数区间的表数范围为：

$$-r_m^{r_e^q - 1} \le N \le -\left(r_m^{-1} + r_m^{-p}\right) \cdot r_m^{-r_e^q}$$

- 规格化浮点数的表数范围主要与阶码的长度q和尾数的基值r_m有关
- 能表示的绝对值最大的浮点数可以近似为

$$|N_{max}| = r_m^{r_e^q}$$

- 假设两种浮点数表示方式
  - **F1**：尾数基值**2**，阶码长度$q1$
  - **F2**：尾数基值r_m，阶码长度$q2$，设$k = \lceil \log_2 r_m \rceil$
- 两种浮点数表示方式的表数范围的比值

$$T = (r_m, q) = \frac{N_{max2}}{N_{max1}} = \frac{r_m^{2^{q_2}}}{r_m^{2^{q_1}}} = \frac{2^{k \cdot 2^{q_2}}}{2^{2^{q_2}}} = 2^{k \cdot 2^{q_2} - 2^{q_1}}$$

3. IEEE754浮点数国际标准

- 32位单精度浮点数格式如下：

符号 S, 1 位	阶码 **e**，**8** 位	尾数数值 ***m***，**23** 位

  - 阶码用**移-127码**表示，即阶码的0～255分别表示阶码的真值为-127～128。
  - 尾数用**原码、小数**，1位符号位、23位小数和1位隐藏的整数共25位表示。
  - 尾数和阶码的基值都是2。
- 64位双精度浮点数，阶码用11位移码表示

- **IEEE754单精度浮点数特殊数据的表示**

➢ 当0＜e＜255时，表示规格化浮点数：

$N=(-1)^S\times2^{e-127}\times(1.m)$

➢ 当e＝255，且$m\neq0$时，表示一个非数NaN，NaN(Not-a-Number)可能是零除以零、求负数的平方根等情况产生的结果。

➢ 当e＝255，且m＝0时，表示一个无穷数：

$(-1)^S\times\infty$。

➢ 当e＝0，且$m\neq0$时，表示规格化浮点数：

$N=(-1)^S\times2^{-126}\times(0.m)$

➢ 当e＝0，且m＝0时，表示浮点数零：$(-1)^S\times0$

4. 浮点数的表数精度（误差）

- 产生误差的根本原因是浮点数的不连续性
- 误差产生的直接原因有两个：

1) 两个浮点数都在浮点集内，而运算结果却可能不在这个浮点集内

2) 数据从十进制转化为2、4、8、16进制，产生误差。

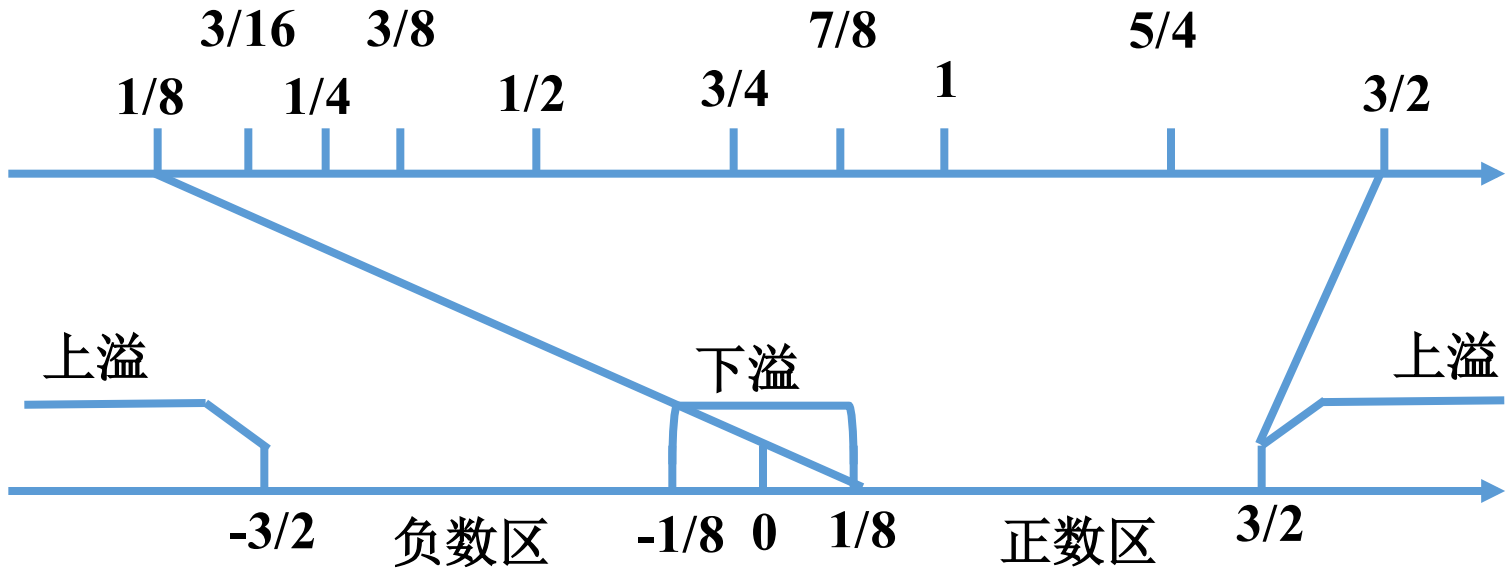
- 令N是浮点数集F内的任一给定实数，而M是F中最接近N，且被用来代替N的浮点数，则绝对表数误差（absolute representation error）为：

$$\delta = |M - N|$$

相对表数误差（relative representation error）为：

$$\delta = \left|\frac{M - N}{N}\right|$$

- 由于在同一种浮点数表示方式中，规格化浮点数的尾数有效位长度是确定的，所以规格化浮点数的相对表数误差是确定的
- 由于规格化浮点数在数轴上的分布是不均匀的，因此其绝对表数误差是不确定的
- 我们主要关注相对表数误差



		阶码（$q=2, r_m=2$）			
		1 (1.1)	0 (1.0)	-1 (0.1)	-2 (0.0)
尾数 $p=2$ $r_m=2$	0.75 (0.11)	3/2	3/4	3/8	3/16
	0.5 (0.10)	1	1/2	1/4	1/8
	-0.5 (1.10)	-1	-1/2	-1/4	-1/8
	-0.75 (1.11)	-3/2	-3/4	-3/8	-3/16

尾数用原码、纯小数表示，阶码用移码、整数表示

十进制0.1的多种浮点数表示方式

尾数基值	阶符	阶码（移码）	尾符	尾数（原码）
2	0	101	0	1100110011001100
4	0	111	0	0110011001100110
8	0	111	0	1100110011001100
16	1	000	0	0001100110011001

- 规格化浮点数的表数精度主要与尾数基值r_m和尾数长度p有关
- 最后1个有效位的精确度为一半，规格化浮点数的精度为

$$\delta(r_m, p) = \frac{1}{2} r_m^{-(p-1)}$$

证明

- 假设浮点数表示为$0.x_1x_2\ldots x_{p-1}x_px_{p+1}\ldots x_{p+n}$，$N$为要表示的浮点数
- 我们实际上是用$0.x_1x_2\ldots x_{p-1}x_px_{p+1}\ldots x_{p+n}$无限逼近$N$
- 其中，$0 \le x_{p+n} \le (r_m-1)\cdot r_m^{-(p+n)}$，则在该位的（平均）误差为

$$\frac{1}{2}(r_m-1)\cdot r_m^{-(p+n)}$$

- 则第p位的累计误差为

$$\frac{1}{2}(r_m-1)\cdot r_m^{-(p+n)}+\frac{1}{2}(r_m-1)\cdot r_m^{-(p+n-1)}+\cdots+\frac{1}{2}(r_m-1)\cdot r_m^{-p}$$

- 即

$$\delta(r_m,p)=\frac{1}{2}(r_m-1)\left(r_m^{-(p+n)}+r_m^{-(p+n-1)}+\cdots+r_m^{-p}\right)=\frac{1}{2}r_m^{-(p-1)}-\frac{1}{2}r_m^{-(p+n)}$$

- 当$n\to\infty$时，有

$$\delta(r_m,p)=\frac{1}{2}r_m^{-(p-1)}$$

- 当$r_m=2$时，有：$\delta(2,p)=\frac{1}{2}2^{-(p-1)}=2^{-p}$
- 当$r_m>2$时，假设$r_m=2^k$（ $k=2,3,\cdots$），二进制的尾数字长为$p_2=p\cdot k$

$$\delta(r_m,p)=\frac{1}{2}r_m^{-(p-1)}=2^{k-1}\cdot 2^{-p_2}$$

- 当浮点数的尾数长度（指二进制位数）相同时，尾数基值r_m取2具有最高的表数精度。当$r_m>2$时，即$r_m=2^k$，浮点数的表数精度与$r_m=2$相比将损失$k-1$倍，即相当于尾数减少了$k-1$个二进制位
- 一般性情况下， $k=\lceil\log_2 r_m\rceil$

5. 浮点数的表数效率

- 浮点数是一种冗余数制(Redundant Number System)
- 浮点数的表数效率定义为：

$$\eta = \frac{\text{可表示的规格化浮点数个数}}{\text{全部浮点数个数}} = \frac{2(r_m - 1)r_m^{p-1} \times 2r_e^q + 1}{2r_m^p \times 2r_e^q}$$

简化表示：

$$\eta(r_m) = \frac{r_m - 1}{r_m}$$

当尾数基值为2时，浮点数的表数效率为：

$$\eta(2) = \frac{2-1}{2} = 50\%$$

- 浮点数的表数效率随r_m增大

  •当尾数基值r_m＝16时，浮点数的表数效率为：

$$\eta(16) = \frac{16-1}{16} = 94\%$$

  •尾数基值r_m＝16与r_m＝2相比，浮点数的表数效率提高了：

$$T = \frac{\eta(16)}{\eta(2)} = 1.875$$

2.1.3 浮点数格式设计

1. 浮点数格式设计的主要问题

- 在表示浮点数的6个参数中，只有尾数基值r_m、尾数长度p和阶码长度q与表数范围、表数精度和表数效率有关

$$|N| < r_m^{r_e^q - 1}$$

$$\eta(r_m) = \frac{r_m - 1}{r_m}$$

$$\delta(r_m, p) = \frac{1}{2} r_m^{-(p-1)}$$

- 在字长确定的情况下，如何选择尾数基值r_m，使表数范围最大、表数精度和表数效率最高

2. 浮点数尾数基值的选择

假设有两种表示方式F1和F2，它们二进制字长相同，尾数都用原码或补码、小数表示，阶码都用移码、整数表示，阶码的基值均为2，尾数的基值不同。

- 浮点数表示方式F1：$r_{m1}=2$，p_1，q_1，

  二进制字长：$L_1=p_1+q_1+2$

- 浮点数表示方式F2：$r_{m2}=2^k$，$p2$，$q2$，

  二进制字长：$L_2=kp_2+q_2+2$

由F1与F2的二进制字长相同，即$L_1=L_2$，得：

$$p_1+q_1=kp_2+q_2 \qquad (2.1)$$

- 字长和表数范围确定时，尾数基值与表数精度的关系

F1的表数范围是：$|N_{1max}| = 2^{2^{q_1}}$

F2的表数范围是：$|N_{2max}| = (2^k)^{2^{q_2}}$

F1与F2的表数范围相同，得到：$2^{2^{q_1}} = (2^k)^{2^{q_2}}$

两边取以2为底的对数得：$q_1 = q_2 + \log_2 k$ (2.2)

(2.2)代入(2.1) 得：$p_1 + q_2 + \log_2 k = kp_2 + q_2$

化简得到：$p_1 = kp_2 - \log_2 k$ (2.3)

F1的表数精度是：$\delta_1 = \frac{1}{2} \cdot 2^{1-p_1}$ (2.4)

把(2.3)代入(2.4)得到：$\delta_1 = \frac{1}{2} \times 2^{1-kp_2+\log k}$

F2的表数精度是：$\delta_2 = \frac{1}{2} \times 2^{k(1-p_2)}$

取F2与F1表数精度的比值：

$$T = \frac{\delta_2}{\delta_1} = 2^{k-\log k-1} \quad (2.5)$$

只有$k=1$ ($r_m=2$)或$k=2$ ($r_m=4$)时，比值$T=1$

- 结论1：在字长和表数范围一定时，尾数基值r_m取2或4，浮点数具有最高的表数精度。

- 字长和表数精度一定，尾数基值r_m与表数范围的关系

由F1与F2的表数精度相同得到：

$$\frac{1}{2}\cdot 2^{1-p_1}=\frac{1}{2}\cdot\left(2^{k}\right)^{1-p_2}$$

即：$p_1=kp_2-k+1$ (2.6)

(2.6)代入(2.1)得：$kp_2-k+1+q_1=kp_2+q_2$

即：$q_1=q_2+k-1$ (2.7)

F1的表数范围：$2^{2^{q_1}}=2^{2^{q_2+k-1}}=2^{2^{q_2}\cdot 2^{k-1}}$

F2的表数范围：$\left(2^{k}\right)^{2^{q_2}}=2^{2^{q_2}\cdot k}$

假设F2的表数范围大于F1的表数范围，则应该有F2阶码的最大值要大于F1阶码的最大值：

$$2^{q_2} \cdot k > 2^{q_2} \cdot 2^{k-1} \text{ 即：} k > 2^{k-1}$$

这个不等式在正整数定义域内没有解

只有$k=1$（$r_m=2$）或 $k=2$（$r_m=4$）时，F2阶码的最大值等于F1阶码的最大值。

- 结论2：在字长和表数精度一定时，尾数基值r_m取2或4，浮点数具有最大的表数范围。
- 推论：在字长确定之后，尾数基值r_m取2或4，浮点数具有最大表数范围和最高表数精度。

例：IBM 370系列机的短浮点数表示方式，r_m＝16，p＝6，r_e＝2，q＝6，尾数用原码、小数表示，阶码用移码、整数表示。求表数范围和表数精度，并与r_m＝2时进行比较。

解：表数精度为：$\delta = \frac{1}{2} \cdot 16^{-(6-1)} = 2^{-21}$

表数范围是：$|N_{max}| = 16^{2^6} = 2^{256}$

若尾数基值r_m＝2，则有：$\frac{1}{2} \cdot 2^{-(p-1)} = 2^{-21}$

解得p＝21，则q＝9，它的表数范围是：

$$|N_{max}| = 2^{2^9} = 2^{512}$$

- 表数效率：
  - 当r_m＝2时：1/2＝50％
  - 当r_m＝4时：3/4＝75％
  - 当r_m＝2时，规格化浮点数可以采用隐藏位方法表示:如果尾数用原码表示，最高位一定为1;如果尾数用补码表示，最高位一定与符号位相反,这时表数效率为100％
- 结论：浮点数的尾数基值r_m取2，并采用隐藏位表数方法是最佳的浮点数表示方式。这种浮点数表示方式能做到表数范围最大、表数精度最高、表数效率最好。

主要的浮点数表示方法

- **IBM公司的IBM360、370、4300系列机等，尾数基值r_m＝16**
- **Burroughs公司的B6700、B7700等大型机，尾数基值r_m＝8**
- **DEC公司的PDP-11、VAX-11和Alpha小型机；CDC公司的CDC6600、CYBER70等大型机；HP公司的PR-RISC；Intel公司的x86系列机；IEEE754浮点数国际标准；……均采用尾数基值r_m＝2**

3. 浮点数格式设计

定义浮点数格式的6个参数，确定原则如下：

- 尾数：多数机器用原码、小数表示
  - 采用原码表示：加减法比补码表示复杂，乘除法比补码简单，而且非常直观。
  - 采用小数表示能简化运算，特别是乘法和除法运算。
- 阶码：一般机器用整数、移码表示
  - 采用移码表示的主要原因是：浮点0与机器0一致。阶码进行加减运算时，移码的加减法运算要比补码复杂

- 基值：
  - 尾数的基值r_m＝2
  - 阶码的基值r_e＝2，
- 采用隐藏位表示方式能够使规格化浮点数的表数效率达到100％（当r_m＝2时）
- 浮点数格式设计的关键问题是：

在表数范围和表数精度给定的情况下，如何确定最短的尾数字长p和阶码字长q，并根据总字长的要求，恰当分配p与q

4. 浮点数格式设计举例

例：请设计一种浮点数格式，要求表数范围不小于10^{37}，正、负数对称，表数精度不低于10^{-16}。

解：根据表数范围的要求：$2^{2^q-1} \geq 10^{37}$

解这个不等式

$$2^q - 1 \geq \frac{\log 10^{37}}{\log 2}\text{，即 } 2^q \geq \frac{\log 10^{37}}{\log 2} + 1$$

两边取以10为底的对数：

$$q \geq \frac{\log(\log 10^{37}/\log 2+1)}{\log 2}\text{, 即 } q \geq 6.95$$

- 取阶码字长$q=7$
- 根据表数精度的要求：$2^{-p} \leq 10^{-16}$，即$p > 53.2$
- 由于浮点数的字长通常为 8 的倍数，因此，取$p=55$
- 总的字长为：$7+55+1+1=64$
- 浮点数格式如下：

- 尾数用原码、小数表示，阶码用移码、整数表示，$r_m=2$，$p=56$(最高有效位隐藏)，$r_e=2$，$q=7$。
- 上例所设计浮点数格式的性能如下：

绝对值最大的尾数：$1 - r_m^{-p} = 1 - 2^{-56}$

绝对值最小的尾数：$1/r_m = 1/2$

最大阶码：$r_e{}^q - 1 = 127$

最小阶码：$-r_e{}^q = -128$

最大正数：$(1-2^{-56}) \times 2^{127} = 1.70 \times 10^{38}$

最小正数：$1/2 \times 2^{-128} = 1.47 \times 10^{-39}$

最大负数：$-1/2 \times 2^{-128} = -1.47 \times 10^{-39}$

最小负数：$-(1-2^{-56}) \times 2^{127} = -1.70 \times 10^{38}$

表数精度：$2^{-56} = 1.39 \times 10^{-17}$

表数效率：100%

浮点零：与机器零相同，64位全0

2.1.4 浮点数的舍入处理

- 浮点数要进行舍入处理的原因是：
  1) 十进制数转化为浮点数时，有效位长度超过给定的尾数字长。
  2) 两个浮点数的加减乘除结果，尾数长度超过给定的尾数字长。
- 舍入处理要解决的问题是：把规格化尾数的p＋g位处理成只有p位
  - 其中：p是浮点数表示方式给定的尾数字长，g是超过给定尾数字长的部分。

- 舍入方法的主要性能标准是：绝对误差小，积累误差小，容易实现。
- 进行舍入处理时要注意的问题是：
  - 必须先规格化，然后再舍入，否则舍入是没有意义的。
  - 在计算积累误差时，要同时考虑到正数区和负数区的情况。

方法1：恒舍法, 又称截断法、必舍法等

- 实现非常容易。
- 误差大：正负区差相反，但同一区误差积累较大。

		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		$\sigma=+2^{-p-1}(2^{g}-1)$

方法2：恒置法（恒置r/2法、冯诺依曼法）

- 规则：把有效字长的最低一位置成r/2。
- 优缺点：实现比较容易，积累误差较小，正负区误差平衡。精度比较低。

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$\sigma=2^{-p}$

方法3：下舍上入法（4舍5入法、0舍1入法等）

- 优缺点：精度高，积累误差小，正负区误差完全平衡。实现起来比较困难。

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$\sigma=+2^{-p-1}$

方法4：R*舍入法 （只有少数巨型机采用）

- **优缺点：没有积累误差，精度很高。实现很复杂。判断g是否为10...0，采用下舍上入法或恒置法，如果溢出，可能要再次右规格化。**

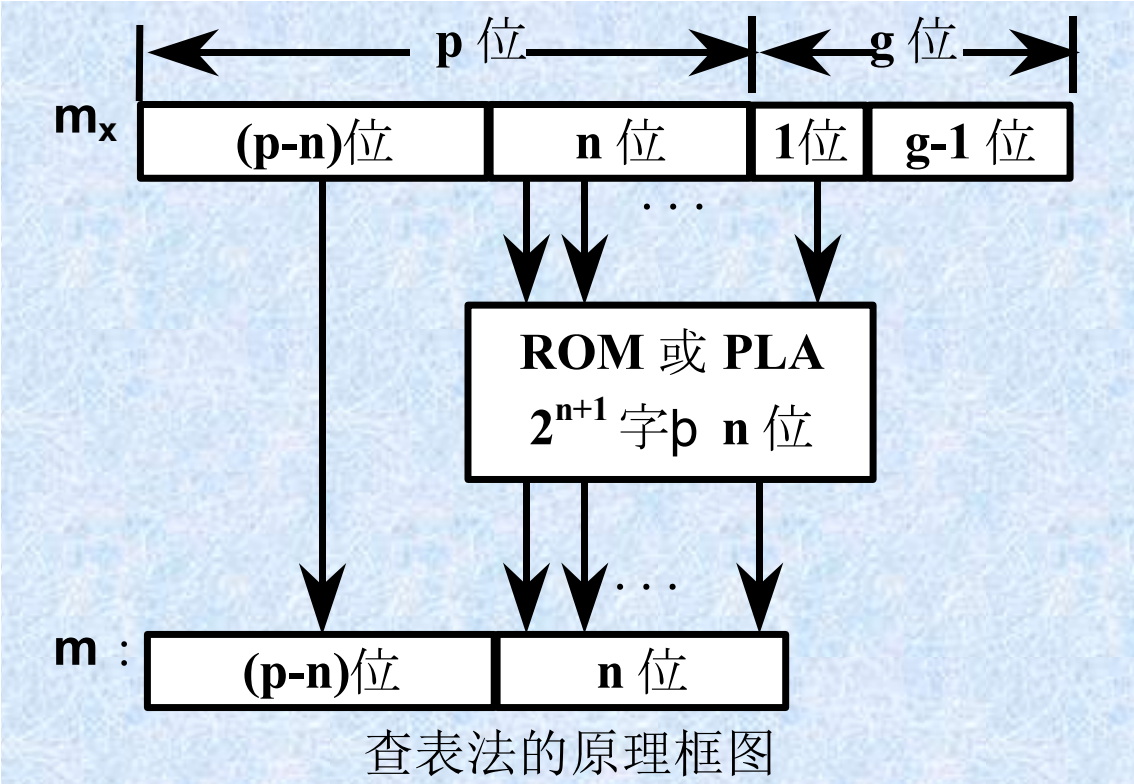
舍入方法	舍入前(p+g 位)	舍入后(p 位)	误差情况
下舍上入法	0.xxx...xx|00...00	0.xxx...xx	0
下舍上入法	0.xxx...xx|00...01	0.xxx...xx	-2^{-p-g}
下舍上入法	0.xxx...xx|0...010	0.xxx...xx	-2^{-p-g+1}
…	…	…	…
下舍上入法	0.xxx...xx|01...11	0.xxx...xx	$-2^{-p-1}(1-2^{-g+1})$
恒置 1 法	0.xxx...x0|10...00	0.xxx...x1	$+2^{-p-1}$
恒置 1 法	0.xxx...x1|10...00	0.xxx...x1	-2^{-p-1}
下舍上入法	0.xxx...xx|10...01	0.xxx...xx$+2^{-p}$	$+2^{-p-1}(1-2^{-g+1})$
…	…	…	…
下舍上入法	0.xxx...xx|11...10	0.xxx...xx$+2^{-p}$	$+2^{-p-g+1}$
下舍上入法	0.xxx...xx|11...11	0.xxx...xx$+2^{-p}$	$+2^{-p-g}$

方法5：查表法（ROM舍入法，PLA舍入法等）

- 优缺点：通过修改ROM或PLA，使积累误差达到平衡。继承了下舍上入法精度高、积累误差小的优点，同时又克服了实现困难的缺点。

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	$-2^{-p}(1-2^{-g})\sim -2^{-p-1}$	

Type equation here.



查表法的原理框图

- 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。

五种舍入方法的主要性能比较

舍入方法	正数区的误差范围	正数区积累误差	实现难易程度
恒舍法	$-2^{-p}(1-2^{-g})\sim 0$	$2^{-p-1}(2^{g}-1)$	最简单
恒置法	$-2^{-p}(1-2^{-g})\sim 2^{-p}$	2^{-p}	很简单
下舍上入法	$-2^{-p-1}(1-2^{-g+1})\sim 2^{-p-1}$	2^{-p-1}	很复杂
R*舍入法	$-2^{-p-1}\sim 2^{-p-1}$	0	最复杂
查表法	$-2^{-p}(1-2^{-g})\sim 2^{-p-1}$	$2^{-p-1}(2^{n}-2^{g})$	一般

关于舍入方法的主要结论：

- 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。
- R*舍入法是唯一积累误差能达到完全平衡的舍入方法，但由于实现非常复杂，仅在少数对误差要求非常高的机器中采用。
- 下舍上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。

- 补码制中的舍入方法：
  - 恒舍法在负数区和正数区的积累误差都是负误差，因此积累误差非常大。为使负数区为正误差，则要改用“恒入法”，实现起来非常困难。
  - 其余4种舍入方法的舍入规则保持不变。
- 反码制中的舍入方法：
  - 恒舍法保持与原码制中相同的舍入方法；
  - 恒置法在负数区要改为恒置反码法；
  - 下舍上入法在负数区要改为上舍下入法，如二进制中改为1舍0入法，且“入”是做减法；
  - 查表法和R*舍入法，负数区的舍入规则由正数区的规则经0、1交换和加、减交换后得到。

2.1.5 警戒位的设置方法

- 在规定的尾数字长之外，运算器中的累加器需要另外增加的长度称为警戒位（Guard Bit）
- **不设置警戒位，可能出现很大的误差**

例2.8：$0.10000000\times2^{0}-0.11111101\times2^{-1}$

	0.10000000×2^{0}
对阶：	$+1.10000001\times2^{0}$
求和：	0.00000001×2^{0}
左规：	0.10000000×2^{-7}

	0.10000000	$\times2^{0}$
对阶：	$+1.10000001$	1×2^{0}
求和：	0.00000001	1×2^{0}
左规：	0.11000000	$\times2^{-7}$

- 不设置警戒位
  可能造成完全错误的运算结果

例2.9：$0.1000\times2^{125}-0.1111\times2^{124}$

	0.1000	$\times2^{125}$
对阶：	$+1.1000$	1×2^{125}
求和：	0.0000	1×2^{125}
左规：	0.1000	$\times2^{121}$

	0.1000×2^{125}
对阶：	$+1.1000\times2^{125}$
求和：	0.0000×2^{125}

- **警戒位的用处只有两个：**

  (1) 用于左规格化时移入尾数有效字长内。

  (2) 用于舍入。

- **警戒位的来源有以下几个方面：**

  (1) 做加、减法时，因对阶从有效字长内移出去的部分。

  (2) 做乘法时，双倍字长乘积的低字长部分。

  (3) 做除法时，因没有除尽而多上商的几位。

  (4) 右规格化时移出有效字长的那部分。

  (5) 从十进制转换成二进制时，尾数超出有效字长的部分。

- **加减法运算对警戒位的需要**

1) 同号相加或异号相减，浮点数的尾数之和不需要左规格化，**因此不必设置警戒位。**

$$\frac{1}{r_m} \leq |m_A| \leq 1, 0 \leq |m_B| \leq 1$$

两数之和为：$1/r_m \leq |m_A + m_B| < 2$

2) 同号相减或异号相加，阶差为0,

例2.10：$0.10001 \times 2^0 - 0.10000 \times 2^0 = 0.10000 \times 2^{-4}$

- 两个尾数之和最多左规格化p-1位，但是对阶时没有移出任何代码。**警戒位有需要，但没有来源，因此不必设置警戒位。**

(3) 同号相减或异号相加，阶差为1，只需要设置一位警戒位。

- **例2.11**：$0.10000\times2^{0}-0.11111\times2^{-1}$

  $=0.10000\times2^{-5}$

  两个尾数之和最多左规格化p位，但是对阶时只移出1位代码

	p=5	g=1
	0. 10000	0×2^{0}
对阶：	+1. 10000	1×2^{0}
求和：	0. 00000	1×2^{0}
左规：	0. 10000	0×2^{-5}

(4) 同号相减或异号相加，阶差大于等于2，只需一位警戒位。

- **例2.12**：$0.10000\times2^{0}-0.11111\times2^{-2}$

  $=0.10000\times2^{-1}$

对阶时移出去很多位，但是两个尾数之和最多只左规格化1位

	p=5	g=2
	0. 10000	00×2^{0}
对阶：	+1. 11000	01×2^{0}
求和：	0. 01000	01×2^{0}
左规：	0. 10000	10×2^{-1}
舍入：	0. 10001	$\times2^{-1}$

- **乘法运算：只需要设置一个警戒位。**

$1/r_m \le |m_A, m_B| < 1$,

两数乘积：$1/r^2_m \le |m_A \times m_B| < 1$,

两个规格化尾数的乘积最多左规格化一位。

- **除法运算：不必设置警戒位。**

$1/r_m \le |m_A, m_B| < 1$,

尾数的商：$1/r_m \le |m_A / m_B| < r_m$,

两个规格化尾数的商不需要左规格化。

2.1.6 自定义数据表示

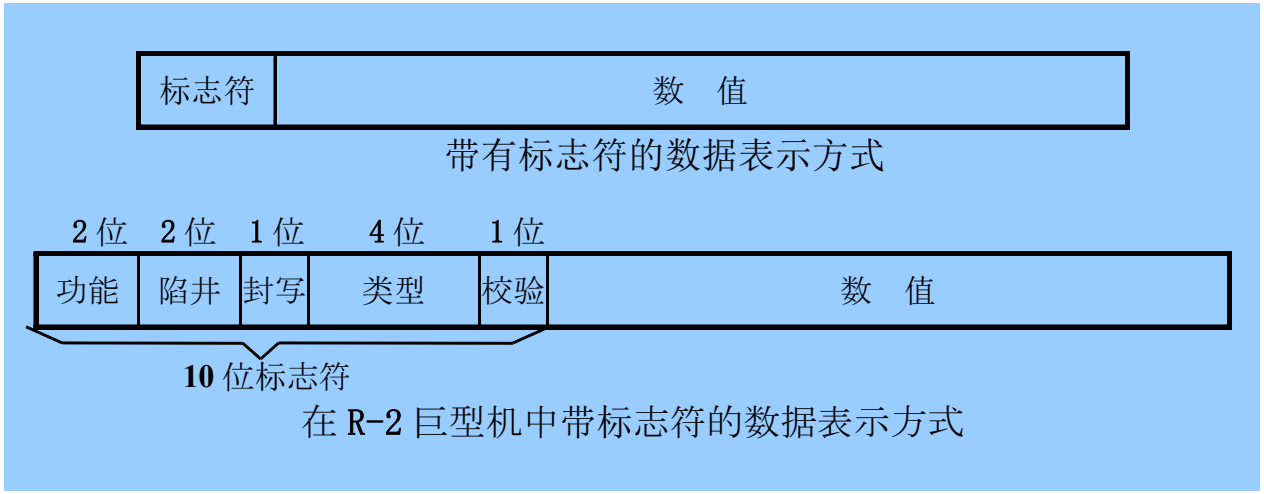
一般处理机中的数据表示方法

- 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；
  - 寻址方式，如直接寻址、间接寻址、相对寻址、寄存器寻址等；
  - 数据的功能，如地址、地址偏移量、数值、控制字、标志等；
  - 同一种操作(如加法)通常有很多条指令。

- **在高级语言和应用软件中**
  - 数据的属性由数据自己定义；
  - 在高级语言与机器语言之间的语义差距，要靠编译器等填补。
- Burroughs公司在大型机中引入**自定义数据表示方式和带标志符的数据表示方式**

1. 带标志符的数据表示法

- 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



- **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。
- **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。**

例2.13：假设X处理机的数据不带标志符，其指令字长和数据字长均为32位；Y处理机的数据带标志符，数据字长增加至35位，其中3位是标志符，其指令字长由32位减少至30位。并假设一条指令平均访问两个操作数，每个操作数平均被访问R次。分别计算这两种不同类型的处理机中程序所占用的存储空间。

解：X和Y处理机程序占用的存储空间总和分别为：$B_X = 32I + \frac{2\times 32I}{R}$ 和 $B_Y = 30I + \frac{2\times 35I}{R}$

- 程序占用存储空间的比值：

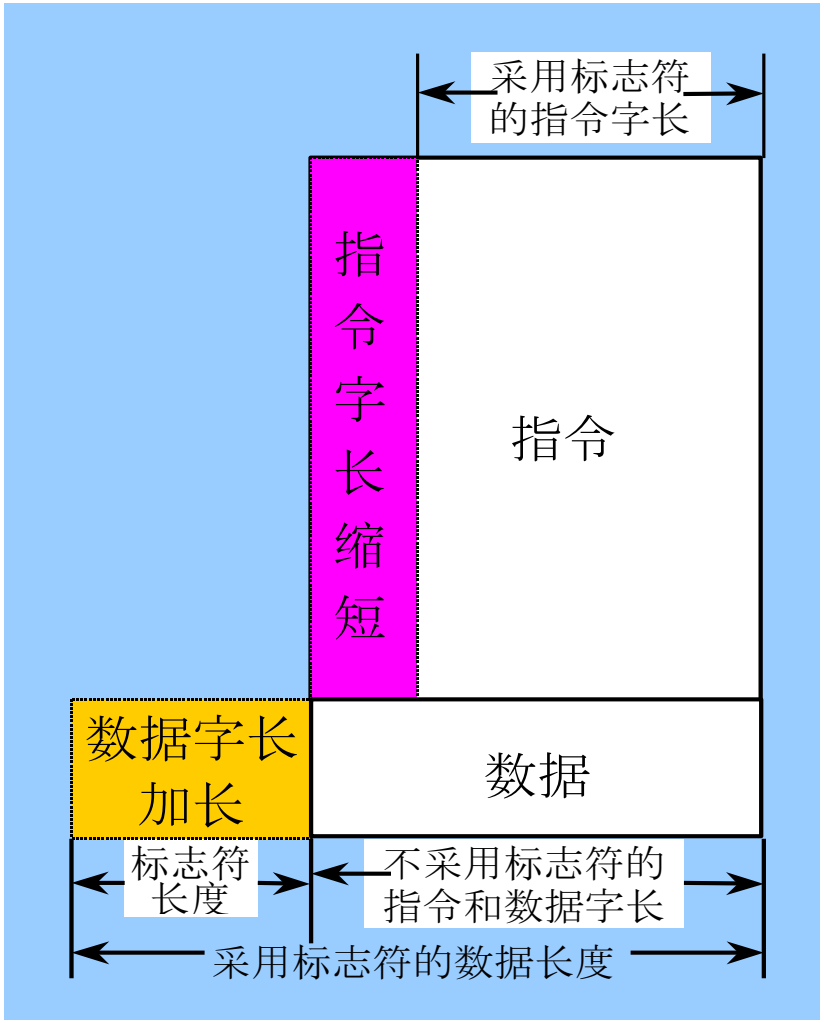
$$\frac{B_Y}{B_X}=\frac{30I+\frac{2\times35I}{R}}{32I+\frac{2\times32I}{R}}=\frac{15R+35}{16R+32}$$

当R＞3时，有 $\frac{B_Y}{B_X}<1$ ，

在实际应用中经常是 R＞10，

- 即带标志符的处理机所占用的存储空间通常要小。

常规数据表示方法与带标志符数据表示方法的比较



例：在IBM370中执行A＝A＋B 运算。

- 若A和B都是十进制数，只需要一条指令，共6个字节，在IBM370/145上执行时间是13微秒。
- 若A与B中有一个是定点二进制数，由于要进行数据类型的一致性检查和转换，在PL/I语言中的编译结果为13条指令，共64个字节，在IBM370/145机上执行时间增加到408微秒。
- 两者相比，存储空间节省5倍，运算速度快30多倍。
- 如果采用自定义数据表示方法，由硬件自动实现一致性检查和数据类型的转换。

- **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) 简化程序设计，缩小了人与计算机之间的语义差距。
4) 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。
5) 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。
6) 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

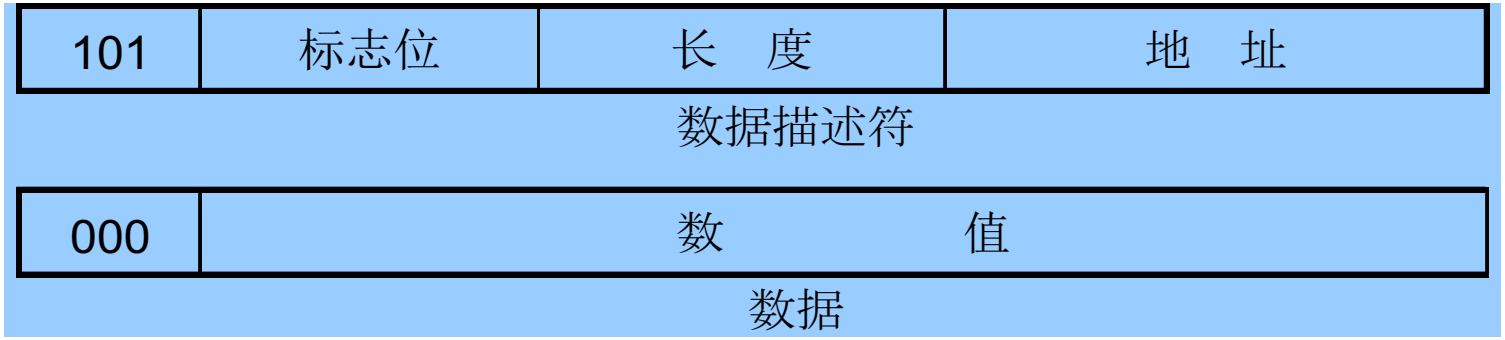
- 但是，程序的运行时间是由设计时间、编译时间和调试时间共同组成的。
- 采用标志符数据表示方法，程序的设计时间、编译时间和调试时间可以缩短。

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。

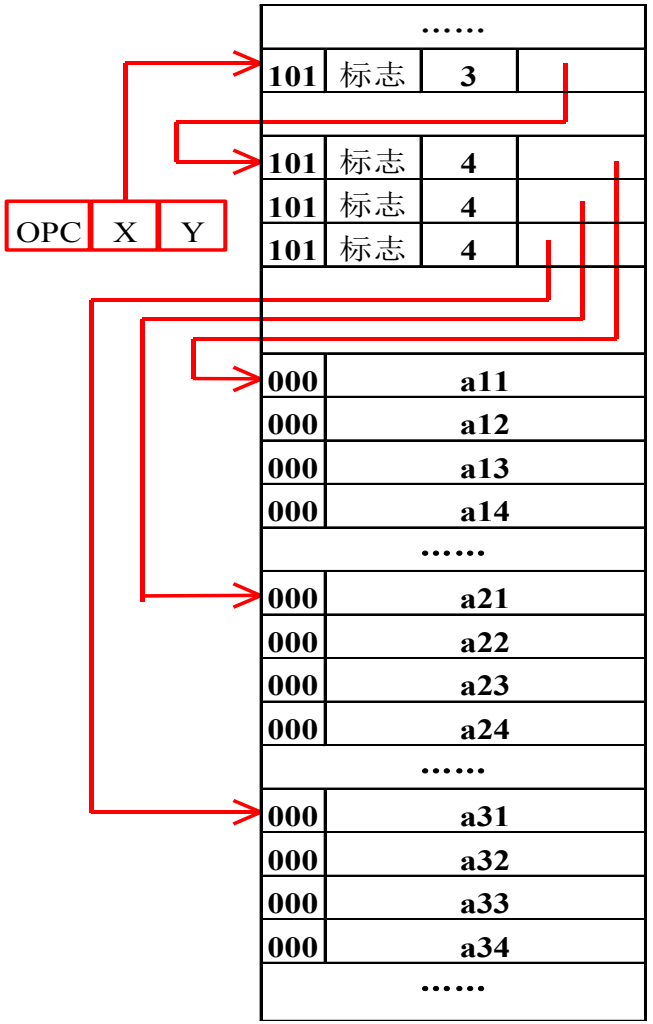
2. 数据描述符表示法

- 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。



例：用数据描述符表示方法表示一个3×4的矩阵：

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



2.2 寻址技术

- 寻找操作数及其地址的技术称为寻址技术

2.2.1 编址方式

2.2.2 寻址方式

2.2.3 定位方式

重点：寻址方式的选择

2.2.1 编址方式

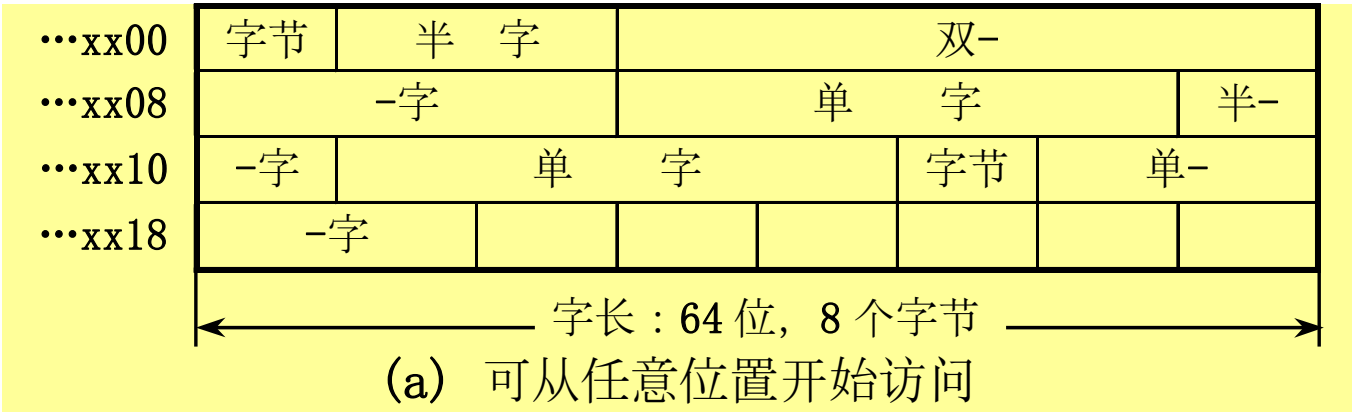
- 对各种存储设备进行编码的方法。
- 主要内容：编址单位、零地址空间个数、并行存储器的编址、输入输出设备的编址

1. 编址单位

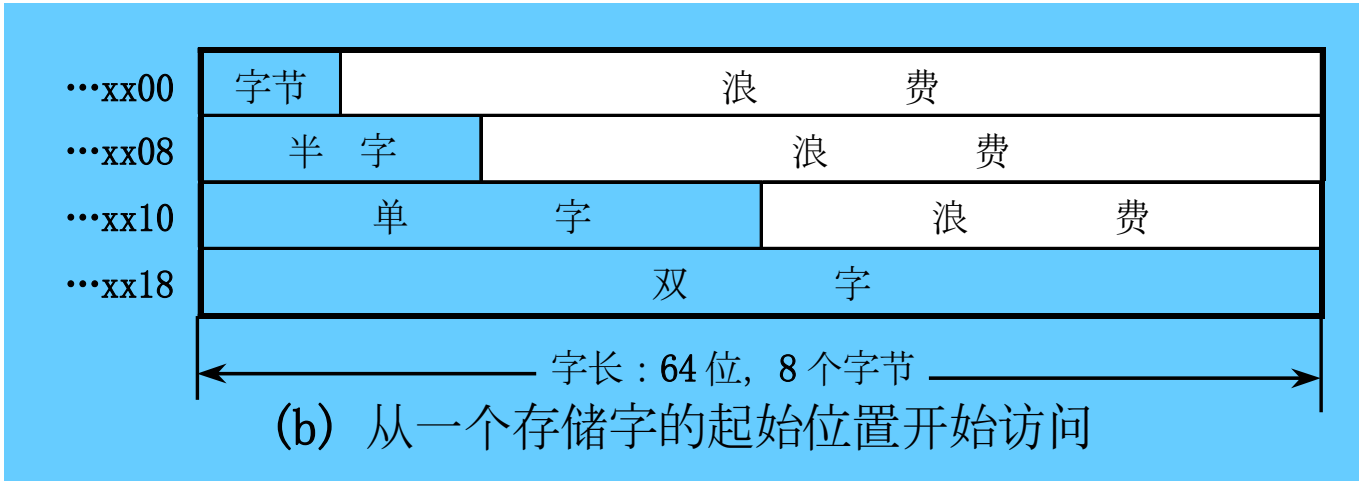
- 常用的编址单位：字编址、字节编址、位编址、块编址等
- 编址单位与访问字长
  - 一般：字节编址，字访问
  - 部分机器：位编址，字访问
  - 辅助存储器：块编址，位访问

- 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) 地址信息浪费：对于32位机器，浪费2位地址(最低2位地址)； 对于64位机器，浪费3位地址
  2) 存储器空间浪费
  3) 读写逻辑复杂
  4) 大端(Big Endian)与小端(Little Endian)问题

(1) 地址信息浪费



(a) 可从任意位置开始访问



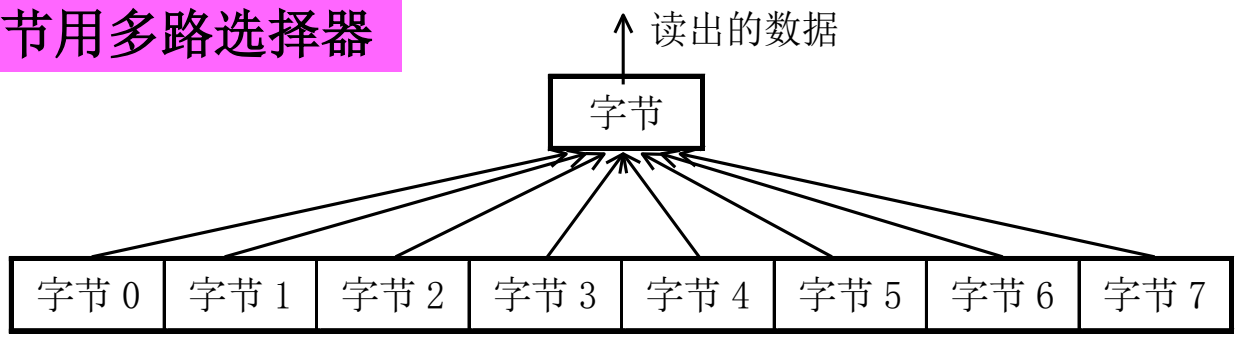
(b) 从一个存储字的起始位置开始访问

(2) 存储器空间浪费

(c) 从地址的整倍数位置开始访问

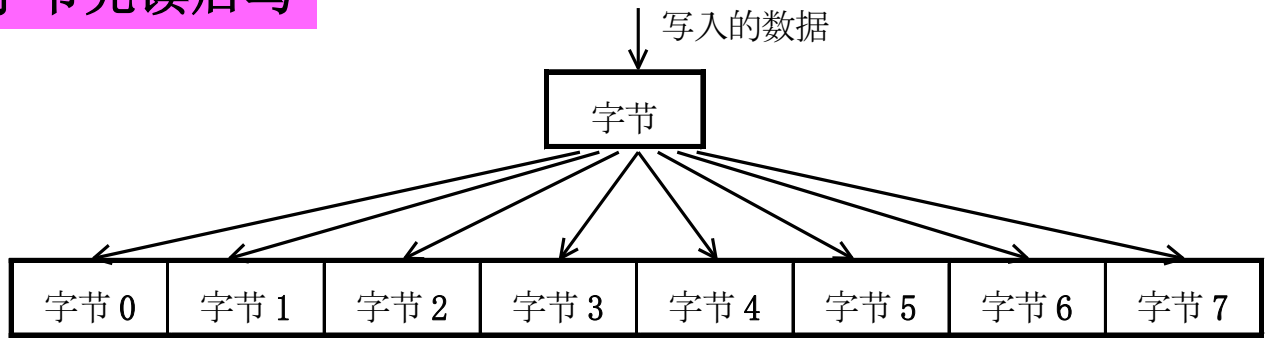
(3) 读写逻辑复杂

读一个字节用多路选择器



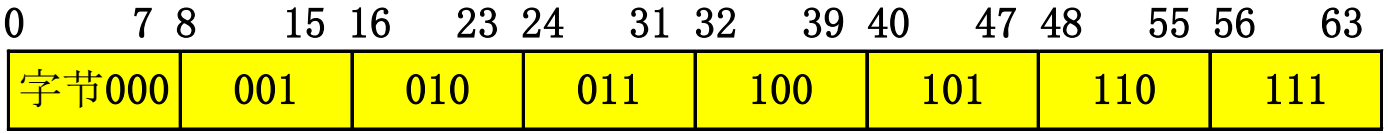
(a) 从存储器里读一个字节

写一个字节先读后写

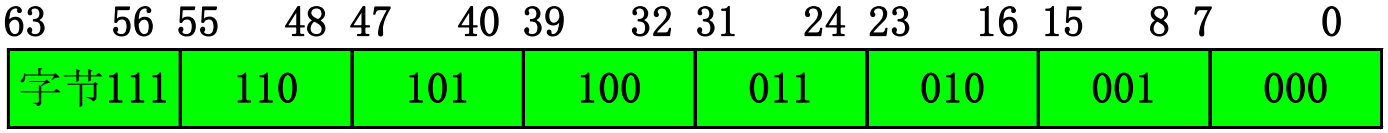


(b) 写一个字节到存储器

(4)大端(Big Endin)与小端(Little Endian)问题

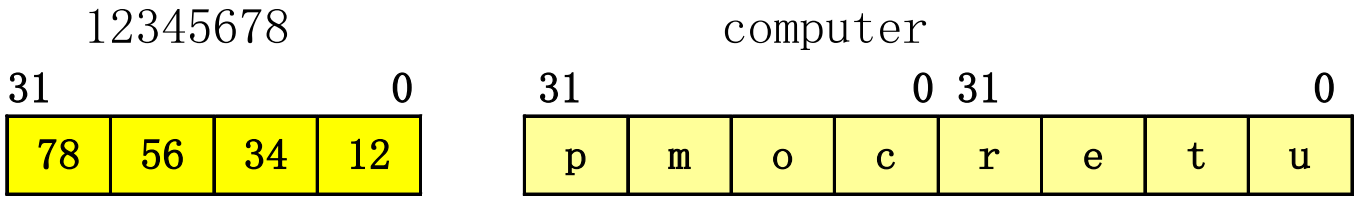


(a) 从左边开始编址



(b) 从右边开始编址

一个存储字的两种编址方式



一种数据存储方式

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

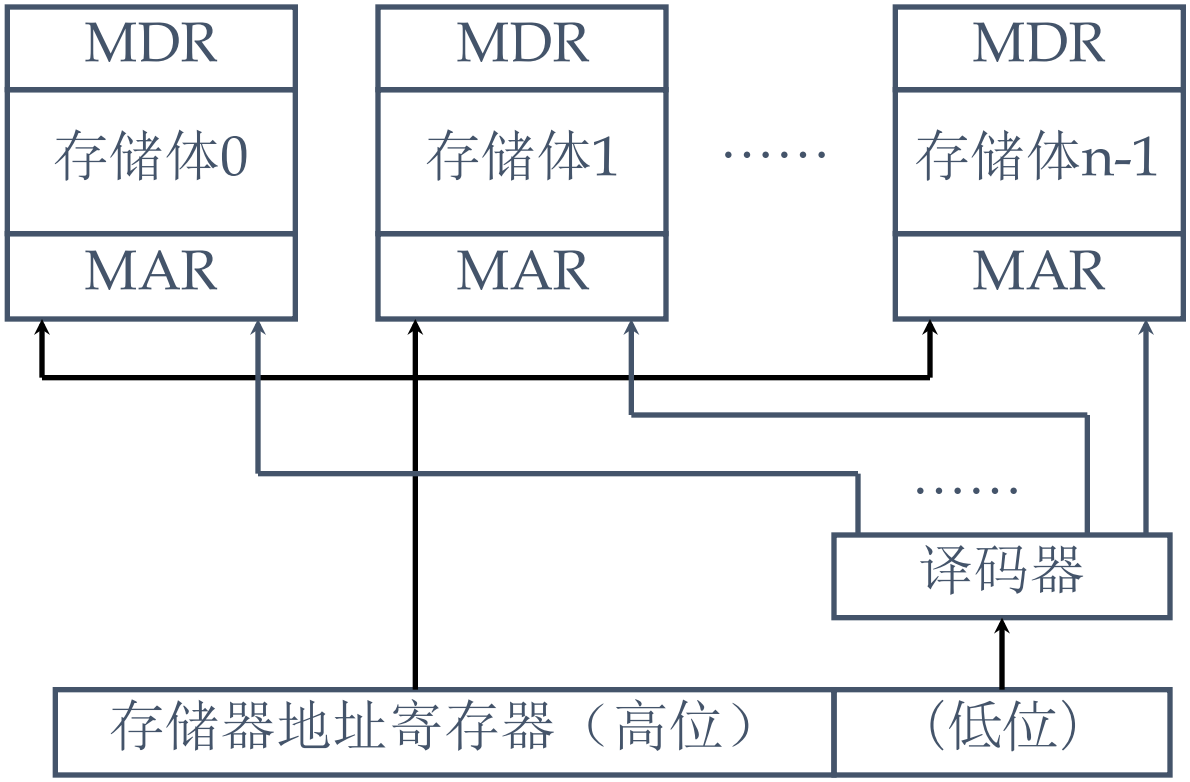
- 高位交叉编址：主要用来扩大存储器容量。
- 低位交叉编址：主要是提高存储器速度。

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$j=A\ mod\ m$
  - 体内地址$i=A/m$
    - M_0: $0,m,2m,\cdots,m(l-1)$+0
    - M_i: $i,m+i,2m+i,\cdots,m(l-1)+i$
- 适合于单处理机内的数据存取和带Cache的主存

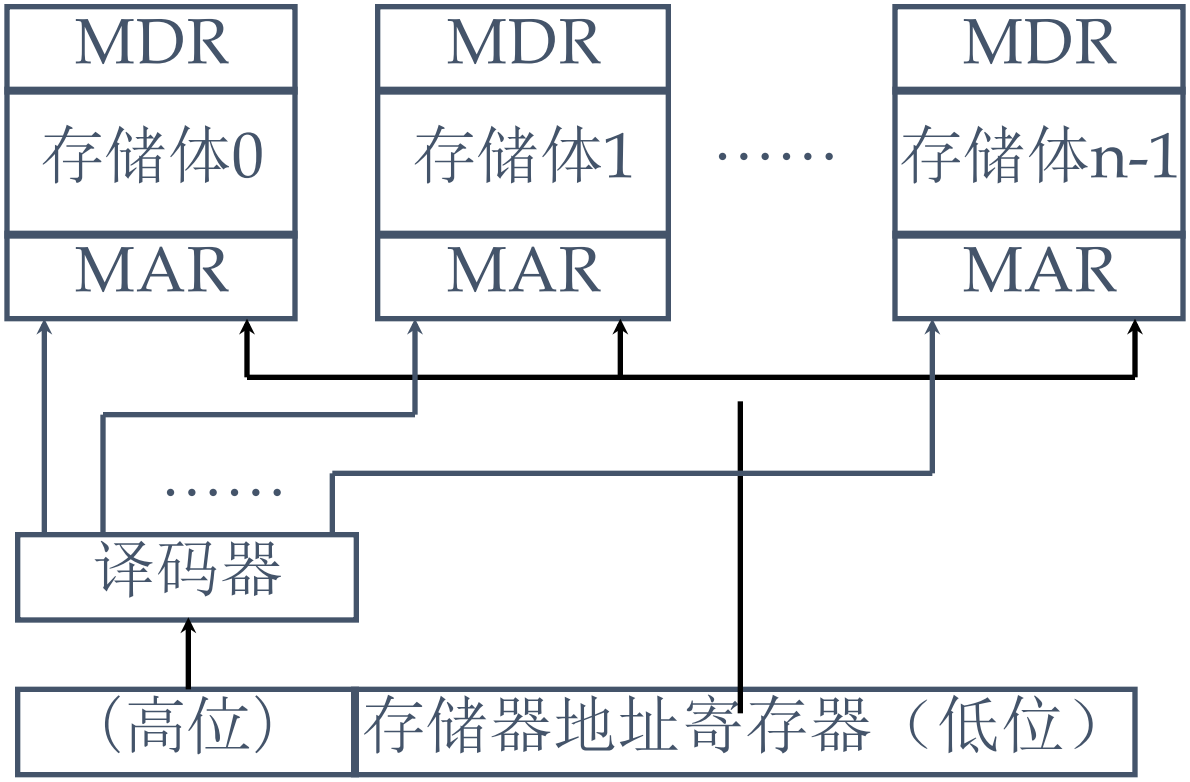
模4低位交叉编址

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

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$m\times j+i$，其中$i=0,1,2,\cdots,l-1$，$j=0,1,2,\cdots,m-1$
- 寻址规则
  - 体地址$j=A/m$
  - 体内地址$i=A\ mod\ m$
    - M_0: $0,1,2,\cdots,l-1$
    - M_i: $il,il+1,\cdots,(i+1)l-1$
- 适合于共享存储器的多机系统，适用于指令和数据分别存于不同分体中


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

4. 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：增加编程的困难

2.2.2 寻址方式

寻找操作数及数据存放地址的方法

1. 寻址方式的设计思想

- 立即数寻址方式
  - 用于数据比较短，且为源操作数的场合
- 面向寄存器的寻址方式

OPC　　R

OPC　　R, R

OPC　　R, R, R

OPC　　R, M

- 面向主存储器的寻址方式：
  - 直接寻址、间接寻址、变址寻址 （相对寻址、基址寻址）

    OPC　M

    OPC　M, M

    OPC　M, M, M
- 面向堆栈的寻址方式：

  OPC　　;运算型指令

  OPC　M　　;数据传送型指令

2. 寄存器寻址

- 主要优点：指令字长短，指令执行速度快，支持向量和矩阵等运算
- 主要缺点：不利于优化编译，现场切换困难,硬件复杂

3. 堆站寻址方式

- 主要优点：支持高级语言，有利与编译程序，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，栈顶部分设计成一个高速的寄存器堆

4. 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
- 原则上，一种处理机中只需设置间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？

例2.15：一个由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重叠问题。为了编程简单，采用一般的两地址指令编写程序。

- 用间接寻址方式编写程序如下：

```
START: MOVE ASR, ASI    ;保存源起始地址
       MOVE ADR, ADI    ;保存目标起始地址
       MOVE NUM, CNT    ;保存数据的个数
LOOP:  MOVE @ASI,@ADI   ;传送一个数据
       INC  ASI         ;源数组的地址增量
       INC  ADI         ;目标数组地址增量
       DEC  CNT         ;个数减1
       BGT  LOOP        ;测试数据传送完?
       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                ;当前正在传送的目标
                        ;数组地址
CNT:   0                ;剩余数据的个数
```

- 用变址寻址方式编写程序如下：

```
START: MOVE  AS,   X          ;将数组起始地址送入变址寄存器
       MOVE  NUM,  CNT        ;保存数据个数
LOOP:  MOVE  (X),AD-AS(X)     ;传送一个数据
       INC   X                ;增量变址寄存器
       DEC   CNT              ;个数减1
       BGT   LOOP             ;测试数据传送完成
       HALT                   ;停机
NUM:   N                      ;传送的数据个数
CNT:   0                      ;剩余数据的个数
```

- 主要优缺点比较：

  1) 采用变址寻址方式编写的程序简单、易读。

  2) 对于程序员，两种寻址方式的主要差别是：

     - 间址寻址：间接地址在主存中，没有偏移量
     - 变址寻址：基地址在变址寄存器中, 有偏移量

  3) 实现的难易程度：间址寻址方式容易实现

  4) 指令的执行速度：间址寻址方式慢

  5) 对数组运算的支持：变址寻址方式比较好

- 自动变址：在访问间接地址时，地址自动增减
- 前变址与后变址：变址与间址混合时
  - 前变址寻址方式：EA＝((X)＋A)
  - 后变址寻址方式：EA＝(X)＋(A)

2.2.3 定位方式

- 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

- 主要的定位方式

➢ 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。

2.3 指令格式的优化设计

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

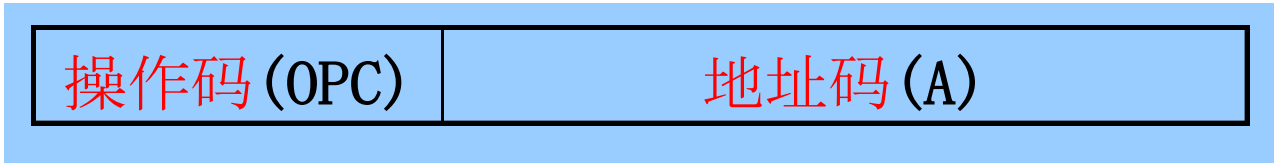
2.3.3 地址码的优化设计

2.3.4 指令格式设计举例

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

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

例如：Burroughs公司的B-1700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

- 1952年由Huffman首先提出
- 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：

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

- 必须知道每种操作码在程序中出现的概率

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

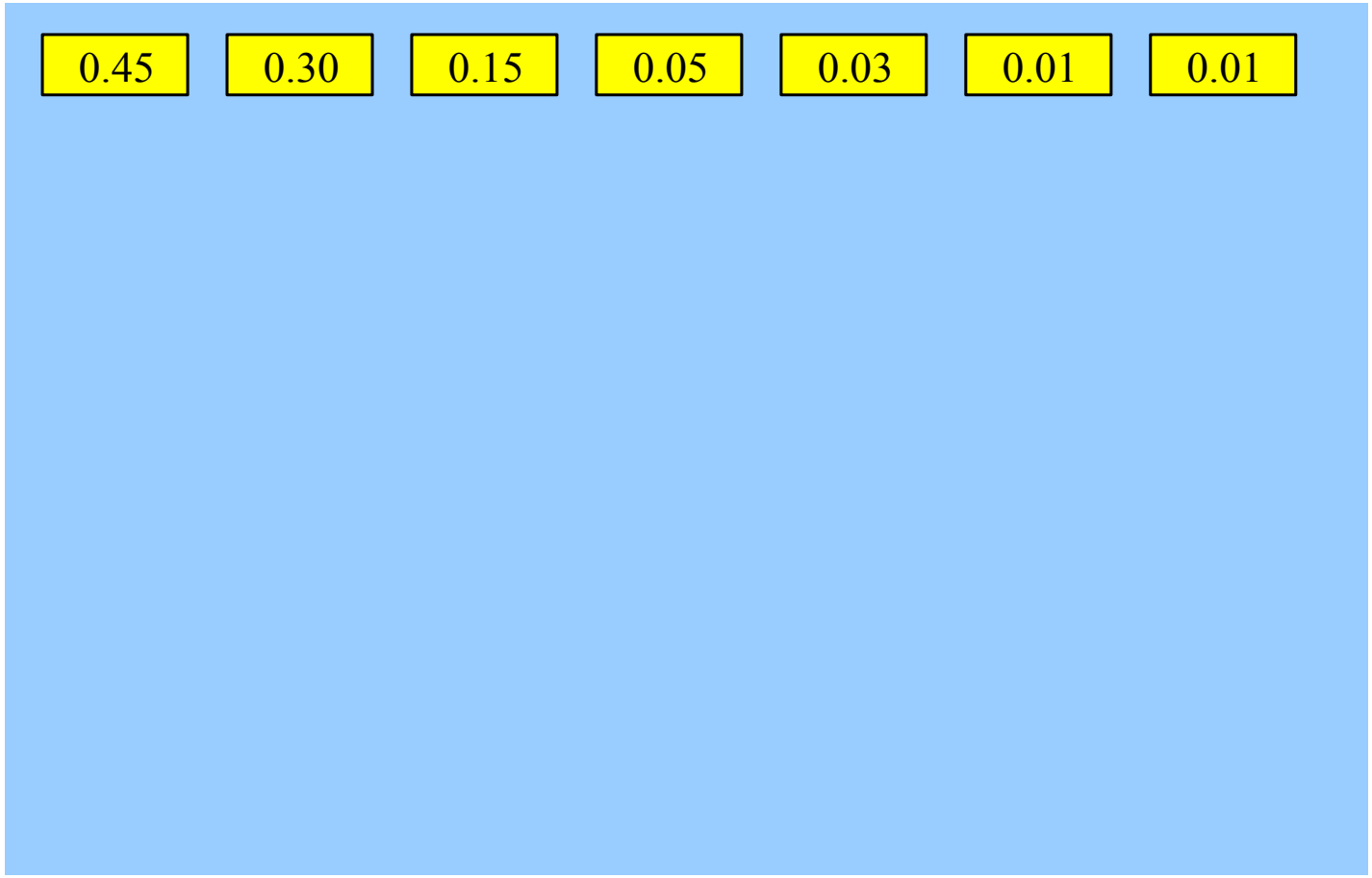
解：

利用Huffman树进行操作码编码（又称最小概率合并法）

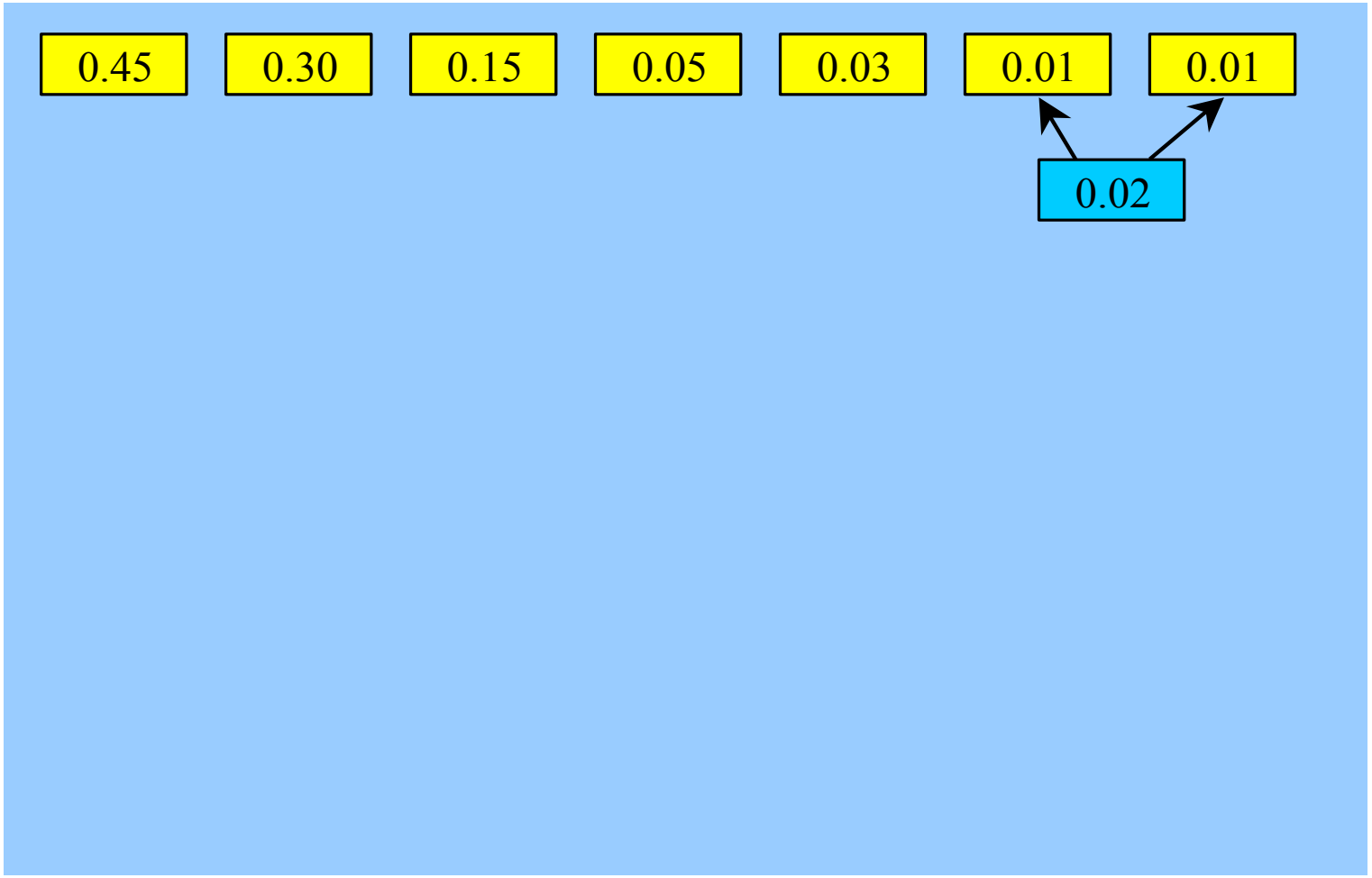
- 把所有指令按照操作码在程序中出现的概率大小，自左向右顺序排列。
- 选取两个概率最小的结点合并成一个概率值是二者之和的新结点，并把这个新结点与其它还没有合并的结点一起形成一个新的结点集合。

- 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。

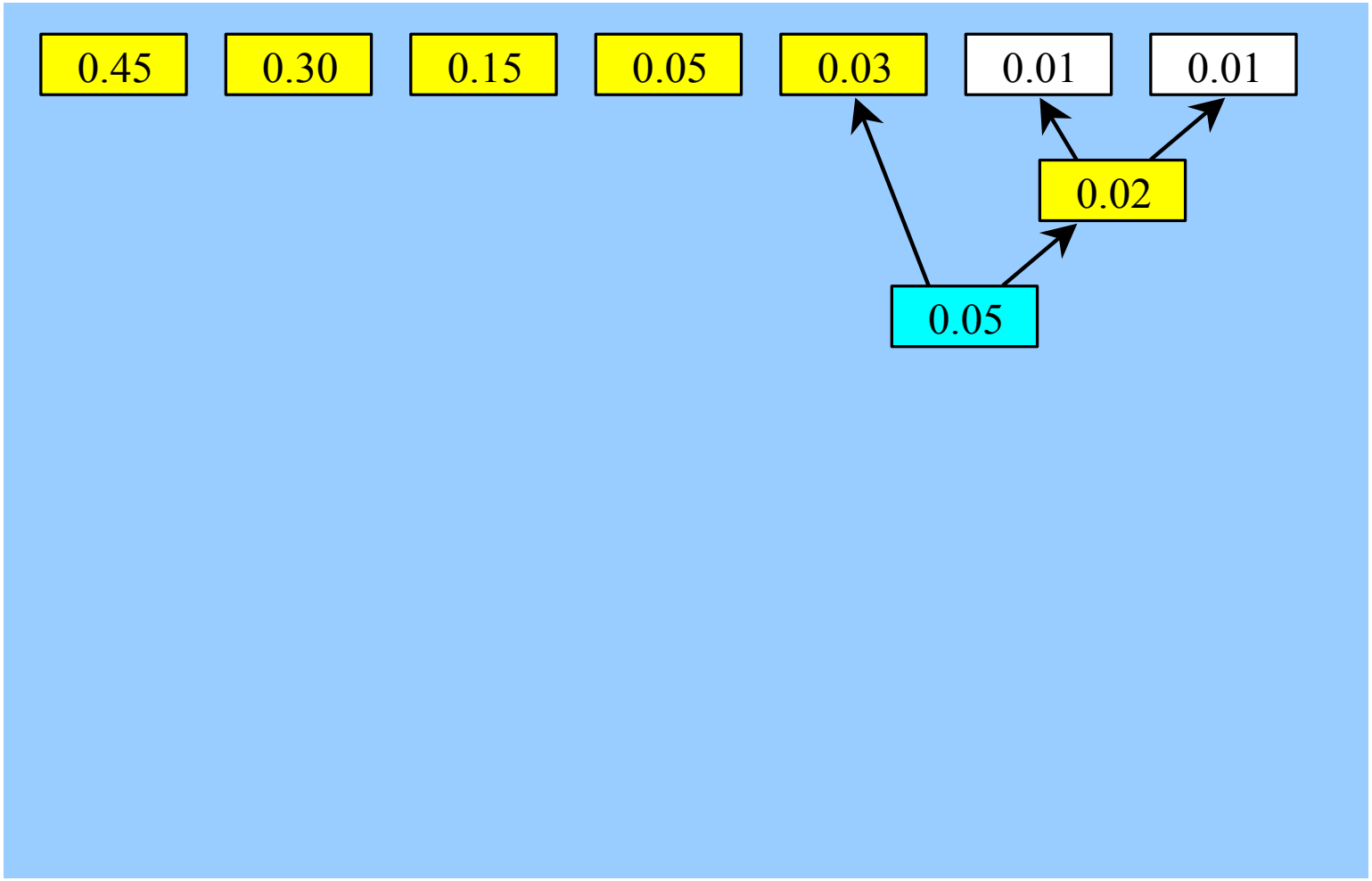
利用Huffman树进行操作码编码



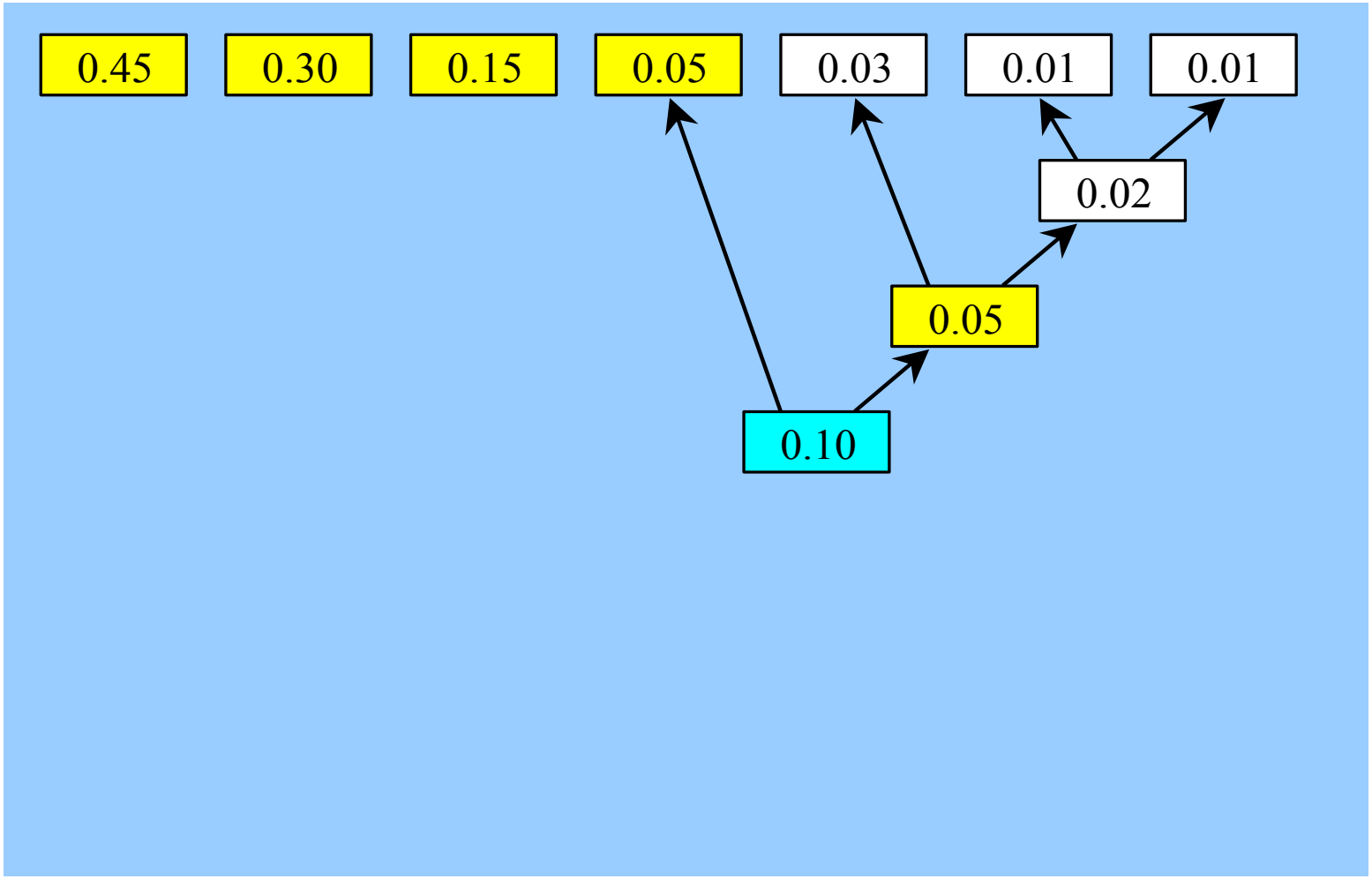
利用Huffman树进行操作码编码



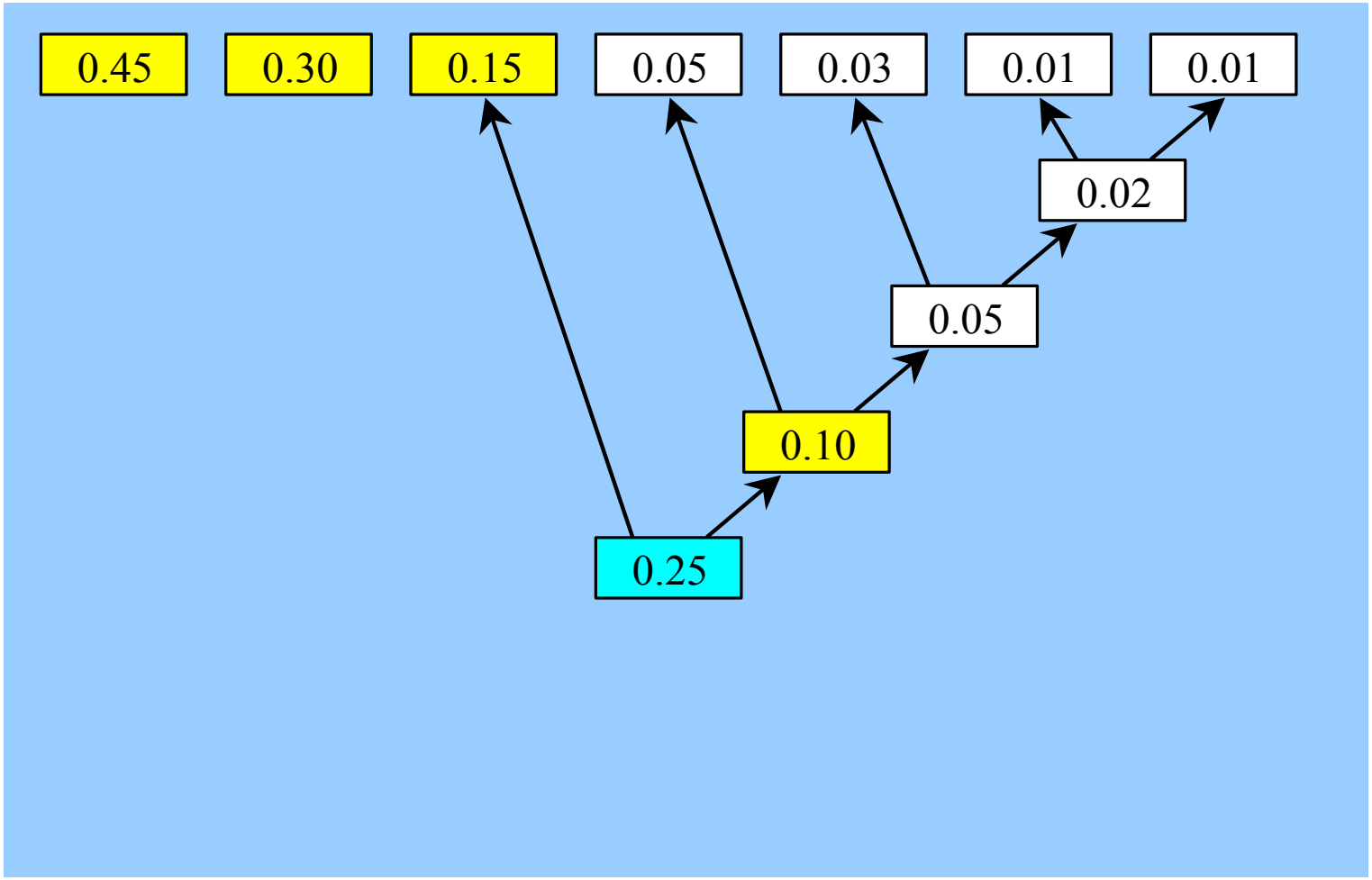
利用Huffman树进行操作码编码



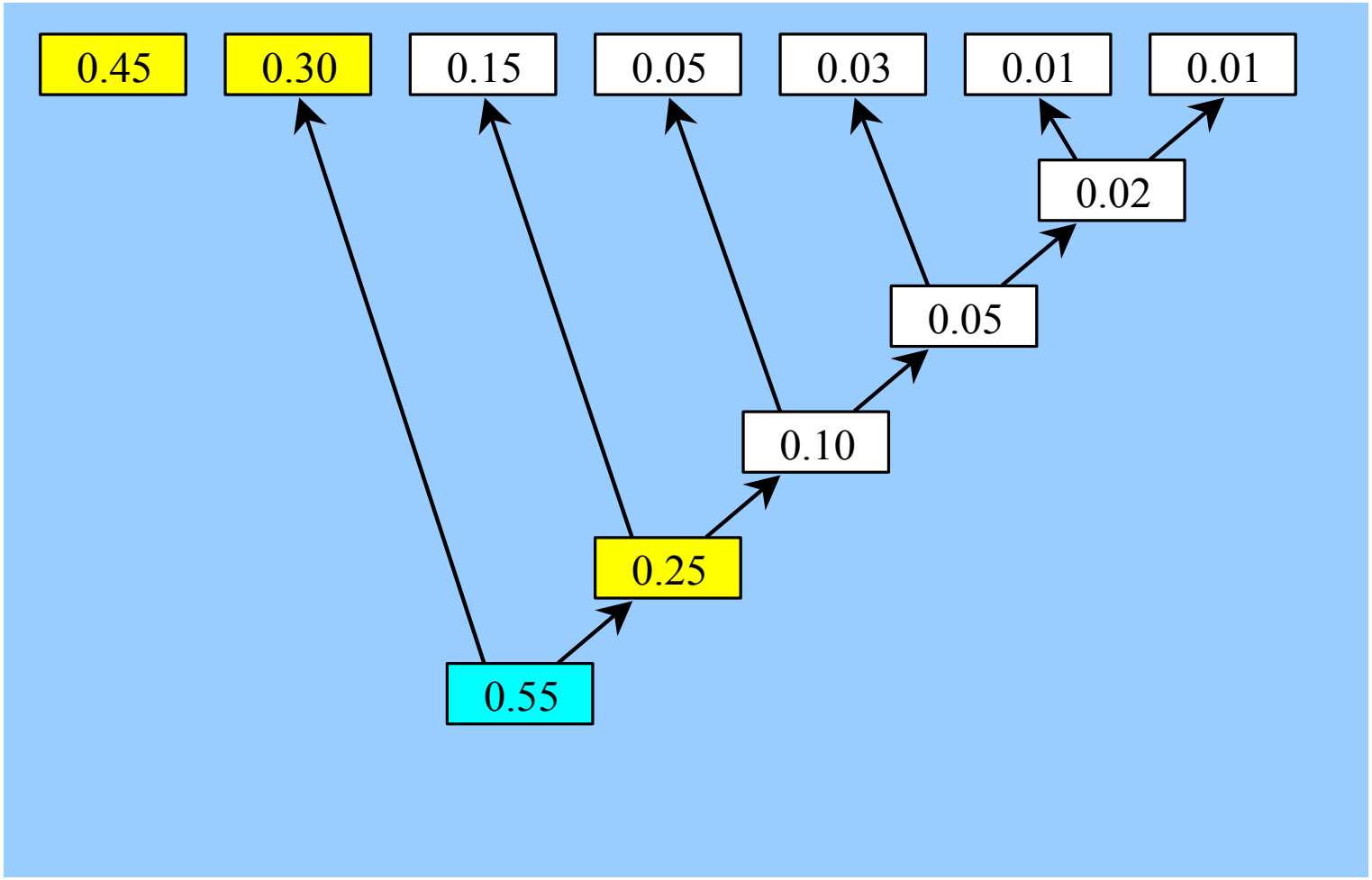
利用Huffman树进行操作码编码



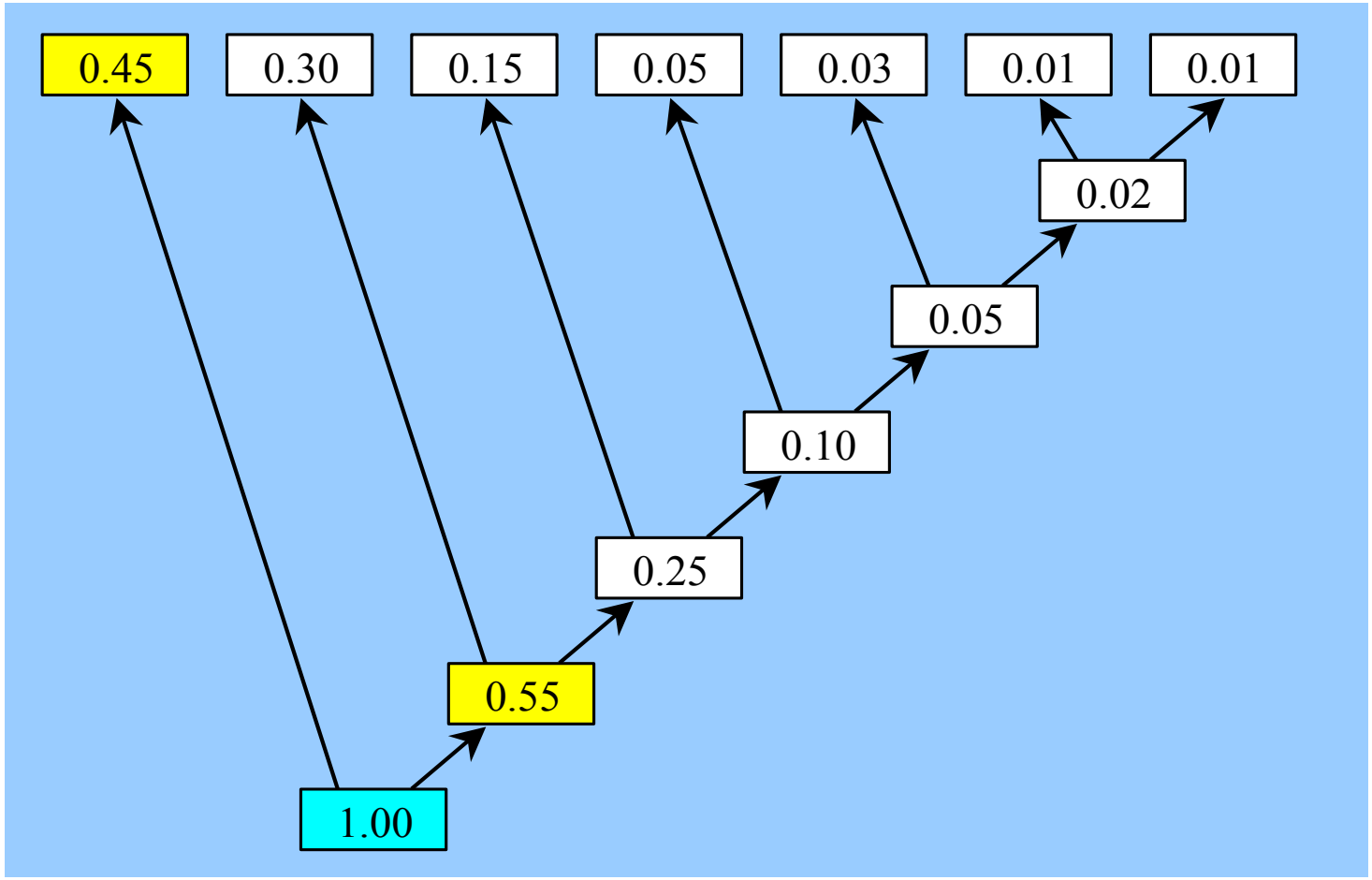
利用Huffman树进行操作码编码



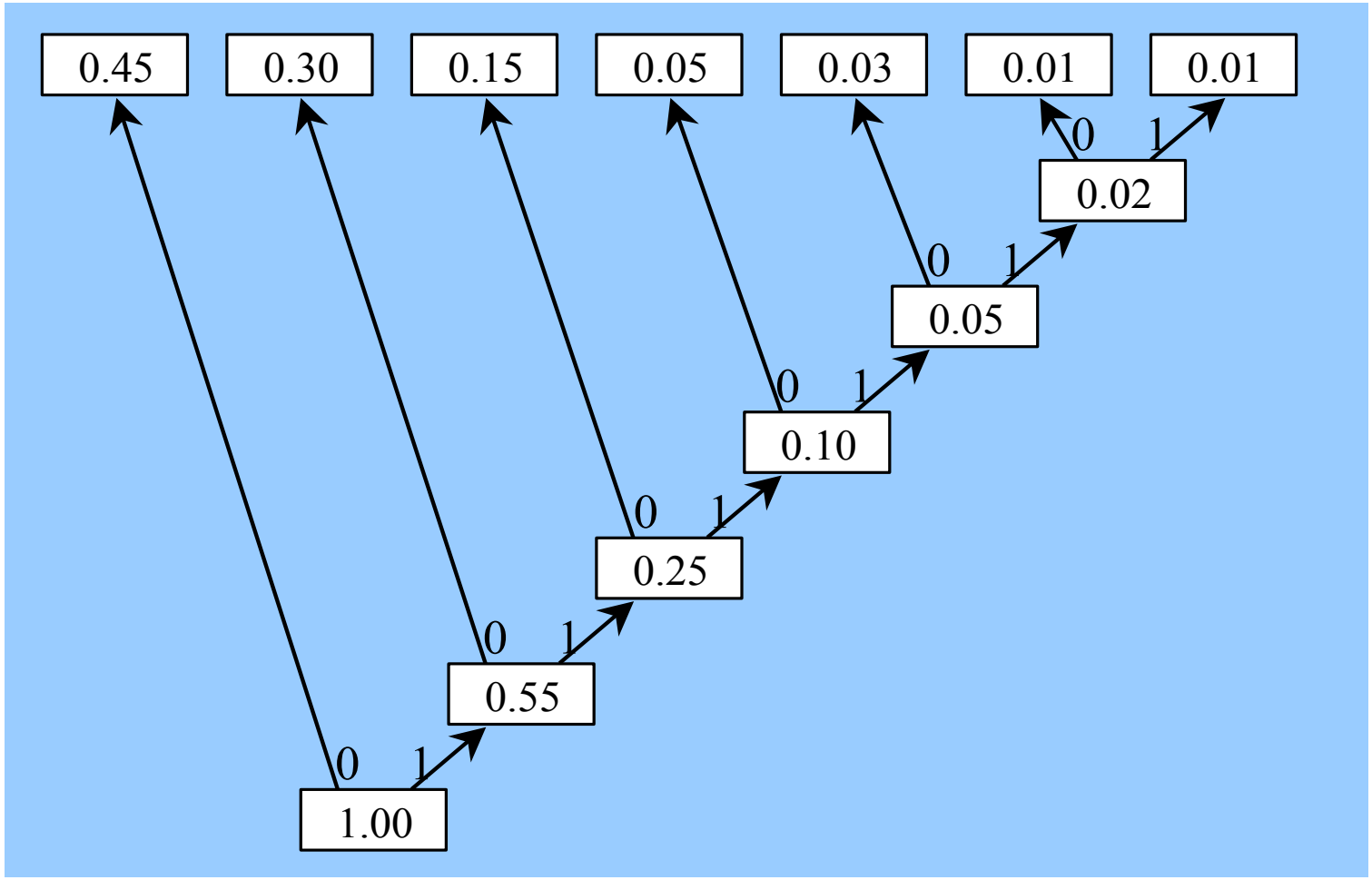
利用Huffman树进行操作码编码



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：

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

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：

$$R = 1 - \frac{H}{\lceil \log_2 7 \rceil} = 1 - \frac{1.97}{3} \approx 35\%$$

- 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**
- **扩展编码法：由固定长操作码与Huffman编码法相结合形成**

例：将例2.17改为1-2-3-5扩展编码法，操作码最短平均长度为：

H＝0.45×1＋0.30×2＋0.15×3＋(0.05＋0.03＋0.01＋0.01)×5

＝2.00

信息冗余量为：$R = 1 - \frac{1.95}{2.00} = 2.5\%$

例：将例2.17改为2-4等长扩展编码法，操作码最短平均长度为：

H＝(0.45＋0.30＋0.15)×2＋(0.05＋0.03＋0.01＋ 0.01)×4＝2.20

2-4等长扩展编码法信息冗余量为：

$$R = 1 - \frac{1.95}{2.20} = 11.4\%$$

7 条指令的操作码扩展编码法

指令序号	出现的概率	1-2-3-5 扩展编码	2-4 等长扩展编码
I_1	0.45	0	0 0
I_2	0.30	1 0	0 1
I_3	0.15	1 1 0	1 0
I_4	0.05	1 1 1 0 0	1 1 0 0
I_5	0.03	1 1 1 0 1	1 1 0 1
I_6	0.01	1 1 1 1 0	1 1 1 0
I_7	0.01	1 1 1 1 1	1 1 1 1
平均长度		2.0	2.2
信息冗余量		2.5%	11.4%

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

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

2.3.3 地址码的优化表示

1. 地址码个数的选择

- 地址码个数通常有3个、2个、1个及 0 个等4种情况
- 价指令中地址码个数应该取多少的标准主要有两个：
  - 程序存储容量，包括操作码和地址码
  - 程序执行速度，以程序执行过程中访问主存的信息量代表

- 三地址指令

- 二地址指令

- 一地址指令

- 零地址指令

例如：计算一个典型的算术表达式：

$$x = \frac{a \times b + c - d}{e + f}$$

➢用三地址指令编写的程序如下：

```
MUL   X, A, B      ;X←(A)×(B)
ADD   X, X, C      ;X←(X)+(C)
SUB   X, X, D      ;分子的计算结果在中
ADD   Y, E, F      ;计算分母,存入Y
DIV   X, X, Y      ;最后结果在X单元中
```

$$x = \frac{a \times b + c - d}{e + f}$$

➢用普通二地址指令编写的程序：

```
MOVE  X,  A      ;复制临时变量到X中
MUL   X,  B
ADD   X,  C
SUB   X,  D      ;X中存放分子运算结果
MOVE  Y,  E      ;复制临时变量到Y中
ADD   Y,  F      ;Y中存放分母运算结果
DIV   X,  Y      ;最后结果在X单元中
```

$$x = \frac{a \times b + c - d}{e + f}$$

➢用多寄存器结构的二地址指令编写程序：

```
MOVE R1, A     ;操作数a取到寄存器R1中
MUL  R1, B
ADD  R1, C
SUB  R1, D     ;R1中存放分子运算结果
MOVE R2, E
ADD  R2, F     ;R2中存放分母运算结果
DIV  R1, R2    ;最后结果在R1中
MOVE X, R1     ;最后结果存入X中
```

$$x = \frac{a \times b + c - d}{e + f}$$

➢用一地址指令编写的程序：

```
LOAD   E     ;先计算分母，
             ;取一个操作数到累加器中
ADD    F     ;分母运算结果在累加器中
STORE  X     ;保存分母运算结果到X中
LOAD   A     ;开始计算分子
MUL    B
ADD    C
SUB    D     ;累加器中是分子运算结果
DIV    X     ;最后运算结果在累加器中
STORE  X     ;保存最后运算结果到X中
```

$$x = \frac{a \times b + c - d}{e + f}$$

➢用0地址指令编写程序： ab*c+d-ef+/

```
PUSH   A     ;操作数a压入堆栈
PUSH   B     ;操作数b压入堆栈
MUL          ;栈顶两数相乘,结果压回堆顶
PUSH   C
ADD
PUSH   D
SUB          ;栈顶是分子运算的结果
PUSH   E
PUSH   F
ADD
DIV          ;栈顶是最后运算的结果
POP    X     ;保存最后运算结果
```

用不同地址个数指令编写的程序的存储容量和执行速度

地址数目	指令条数	访存次数	程序存储量	执行速度(访存信息量)
三地址	5	20	5P＋15A＝65B	5P＋15A＋15D＝185B
二地址	7	26	7P＋14A＝63B	7P＋14A＋19D＝215B
一地址	9	18	9P＋ 9A＝45B	9P＋ 9A ＋ 9D＝117B
零地址	12	41	12P＋7A＝40B	12P＋7A＋29D＝272B
二地址寄存器型	8	15	8P+7A+9R＝40B	8P+7A+9R+7D＝96B

P 表示操作码长度，A 表示地址码长度，D 表示数据长度，R 表示通用寄存器的地址码长度，B 表示字节数。并取：D＝2A＝8P＝16R＝8B

不同地址个数指令的特点及适用场合

地址数目	程序的长度	程序存储量	程序执行速度	适用场合
三地址	最短	最大	一般	向量，矩阵运算为主
二地址	较短	很大	很低	一般不宜采用
一地址	较长	较大	较快	连续运算，硬件结构简单
零地址	最长	最小	最低	嵌套，递归，变量较多
二地址寄存器型	一般	最小	最快	多累加器，数据传送较多

- 关于地址码个数结论：

➢ 对于一般商用处理机，采用多寄存器结构的二地址指令是最理想的。

➢ 如果强调硬件结构简单，并且以连续运算（如求累加和等）为主，宜采用一地址结构。

➢ 对于以向量、矩阵运算为主的处理机，最好采用三地址结构。部分RISC处理机也采用三地址指令。

➢ 对于解决递归问题为主的处理机，宜采用零地址结构。编程容易、节省程序存储量。

2. 缩短地址码长度的方法

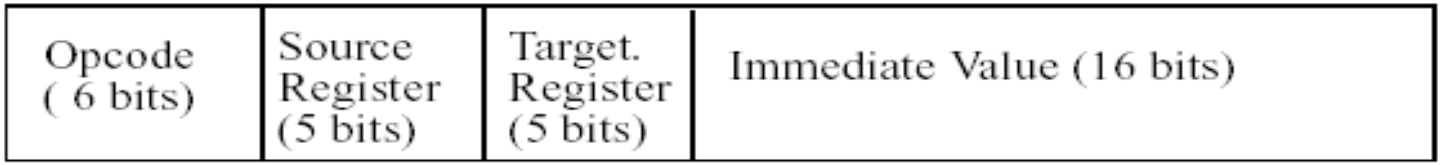
用一个短地址码表示一个大地址空间

- 用间址寻址方式缩短地址码长度
  - 方法：在主存储器的低端开辟一个专门存放间接地址的区域
- 用变址寻址方式缩短地址码长度
  - 变址寻址方式中的地址偏移量比较短，
- 用寄存器间接寻址方式缩短地址码长度
  - 例如：16个间址寄存器，用4位地址码就能表示很长的逻辑地址空间。

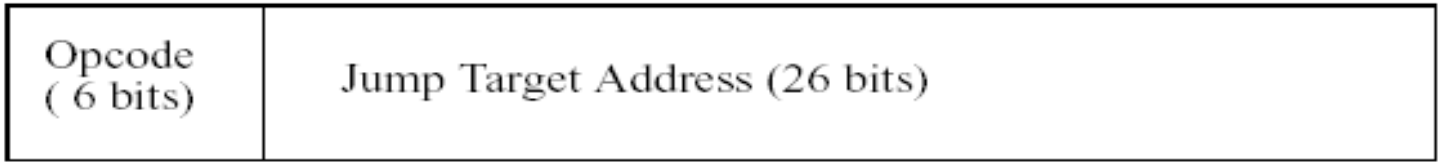
2.3.4 指令格式设计举例

1. MIPS32-4K的指令格式

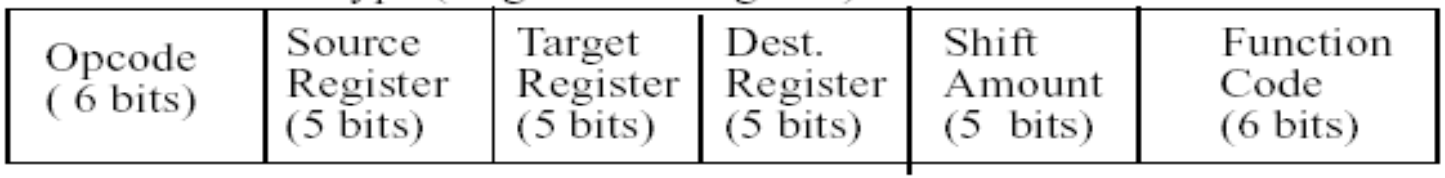
I-Type (Immediate)



J-Type (Jump)



R-Type (Register-to-Register)



- **MIPS32-4K的指令格式**

Figure 4-1 Immediate (I-Type) CPU Instruction Format

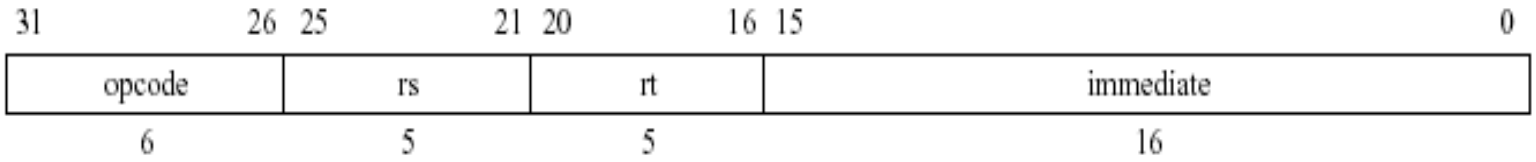


Figure 4-2 Jump (J-Type) CPU Instruction Format

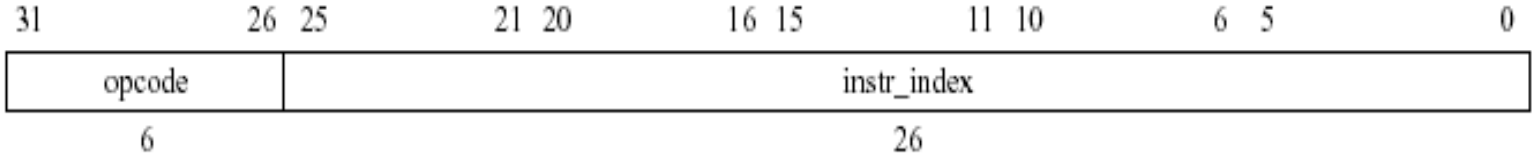
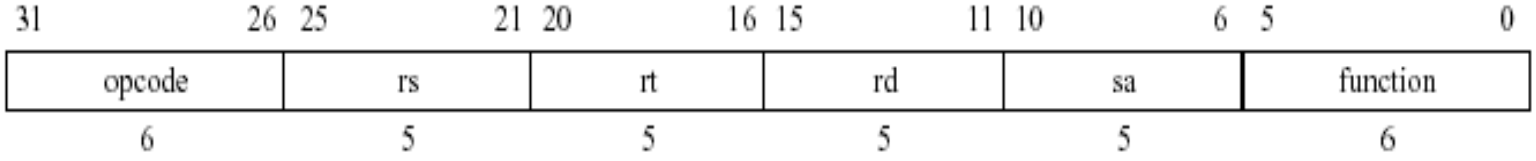


Figure 4-3 Register (R-Type) CPU Instruction Format



- **MIPS32-4K的指令格式**

Table 4-23 CPU Instruction Format Fields

Field	Description
opcode	6-bit primary operation code
rd	5-bit specifier for the destination register
rs	5-bit specifier for the source register
rt	5-bit specifier for the target (source/destination) register or used to specify functions within the primary *opcode* REGIMM
immediate	16-bit signed *immediate* used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement
instr_index	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address
sa	5-bit shift amount
function	6-bit function field used to specify functions within the primary *opcode* SPECIAL

- **MIPS32-4KC指令Opcode段编码**

Opcode		Bits 28 .. 26							
Bits 31..29		0	1	2	3	4	5	6	7
		000	001	010	011	100	101	110	111
0	000	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XOIT	LUI
2	010	COP0	保留	MIPS 保留	MIPS 保留	BEQL	BNEL	BLEZL	BGTZL
3	011	保留	保留	保留	保留	SPECIAL2	保留	保留	可扩展
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	保留
5	101	SB	SH	SWL	SW	保留	保留	SWR	CACHE
6	110	LL	保留	MIPS 保留	PREF	保留	保留	MIPS 保留	保留
7	111	SC	保留	MIPS 保留	可扩展	保留	保留	MIPS 保留	保留

- **SPECIAL操作码对应Function段编码**

function		Bits 2 .. 0							
		0	1	2	3	4	5	6	7
Bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL	保留	SRL	SRA	SLLV	可扩展	SRLV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	可扩展	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	保留	可扩展	保留	保留
3	011	MULT	MULTU	DIV	DIVU	保留	保留	保留	保留
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	可扩展	可扩展	SLT	SLTU	保留	保留	保留	保留
6	110	TGE	TGEU	TLT	TLTU	TEQ	可扩展	TNE	可扩展
7	111	保留	可扩展	保留	保留	保留	可扩展	保留	保留

- **SPECIAL2操作码对应Function段编码**

Function		Bits 2 .. 0							
		0	1	2	3	4	5	6	7
Bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	MIPS 保留	MSUB	MSUBU	MIPS 保留	MIPS 保留
1	001	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留
2	010	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留
3	011	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留
4	100	CLZ	CLO	MIPS 保留	MIPS 保留	保留	保留	MIPS 保留	MIPS 保留
5	101	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留
6	110	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留
7	111	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	SDBBP

- **REGIMM操作码对应rt段编码**

rt		Bits 18 .. 16							
Bits		0	1	2	3	4	5	6	7
20..19		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	BLTZL	BGEZL	可扩展	可扩展	可扩展	可扩展
1	01	TGEI	TGEIU	TLT	TLTIU	TEQI	可扩展	TNEI	可扩展
2	10	BLTZAL	BGEZAL	BLTZALL	BGEZALL	可扩展	可扩展	可扩展	可扩展
3	11	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展

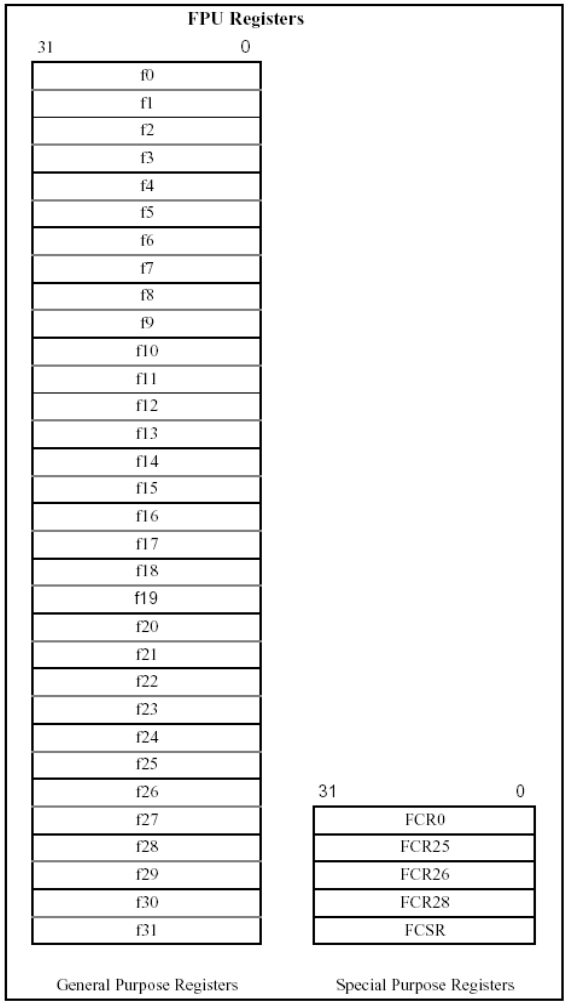
- **COP0 中 rs 段编码**

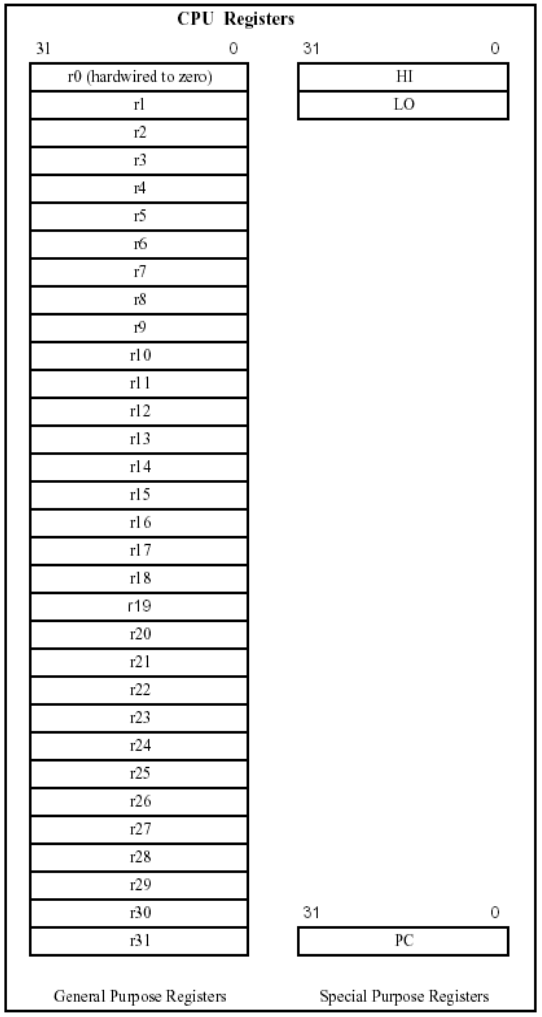
rs		Bits 23 .. 21							
Bits		0	1	2	3	4	5	6	7
25..24		000	001	010	011	100	101	110	111
0	00	MFC0	保留	可扩展	可扩展	MTC0	保留	可扩展	可扩展
1	01	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展
2	10	CO							
3	11								

- **COP0 中 rs为CO时funciton段编码**

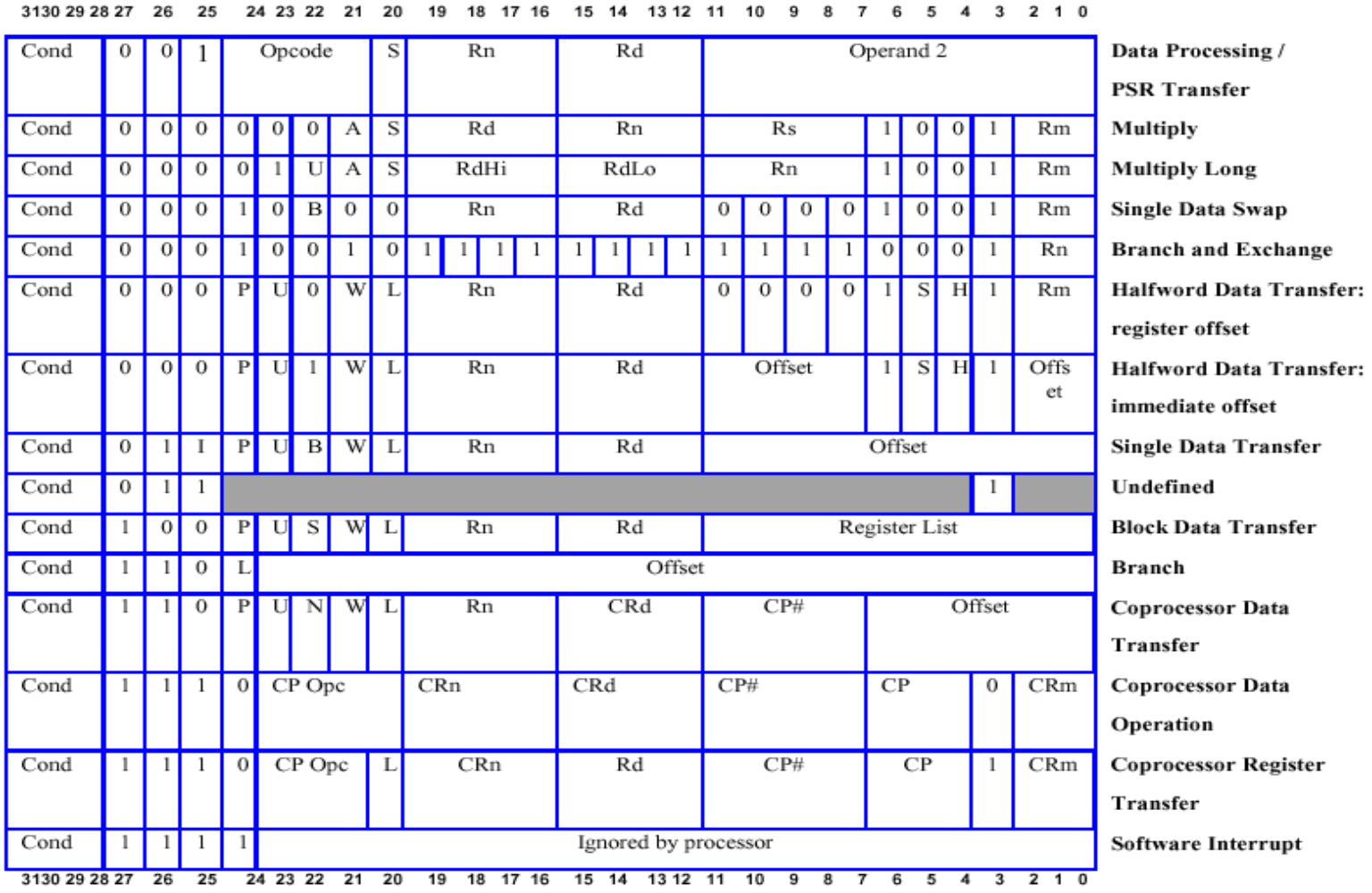
function		Bits 2 .. 0							
		0	1	2	3	4	5	6	7
Bits 5..3		000	001	010	011	100	101	110	111
0	000	可扩展	TLBR	TLBWI	可扩展	可扩展	可扩展	TLBWR	可扩展
1	001	TLBP	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展
2	010	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展
3	011	ERET	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	DERET
4	100	WAIT	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展
5	101	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展
6	110	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展
7	111	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展	可扩展

MIPS32 4K的寄存器结构





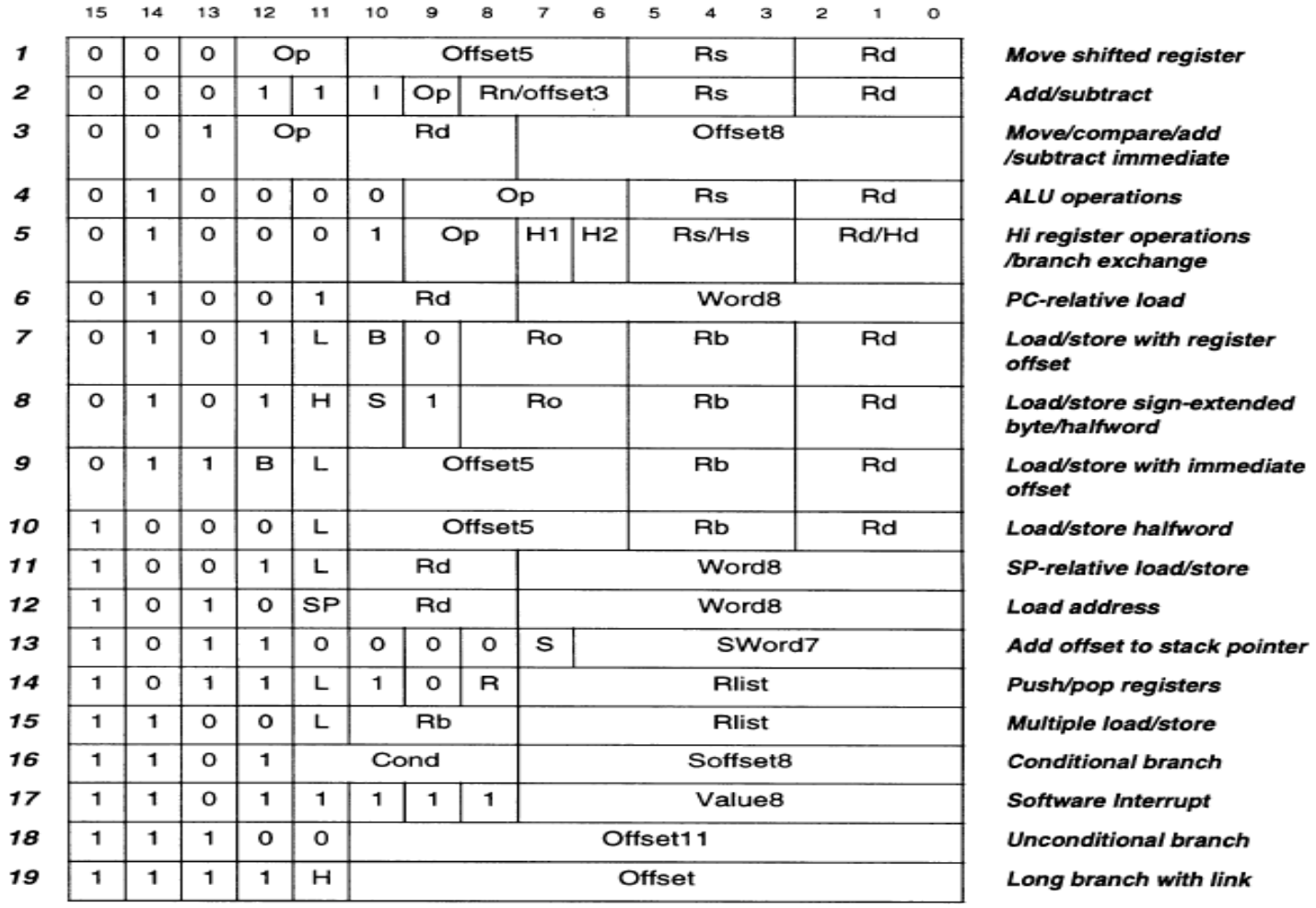
2. ARM处理机的指令格式（32位）



- **ARM处理机的指令格式（条件执行）**

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

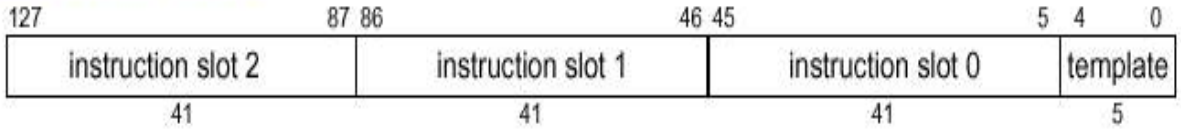
- **ARM处理机的指令格式（16位）**



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	Op		Offset5					Rs			Rd			Move shifted register
2	0	0	0	1	1	I	Op	Rn/offset3			Rs			Rd			Add/subtract
3	0	0	1	Op		Rd			Offset8								Move/compare/add /subtract immediate
4	0	1	0	0	0	0	Op				Rs			Rd			ALU operations
5	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			Rd/Hd			Hi register operations /branch exchange
6	0	1	0	0	1	Rd			Word8								PC-relative load
7	0	1	0	1	L	B	0	Ro			Rb			Rd			Load/store with register offset
8	0	1	0	1	H	S	1	Ro			Rb			Rd			Load/store sign-extended byte/halfword
9	0	1	1	B	L	Offset5					Rb			Rd			Load/store with immediate offset
10	1	0	0	0	L	Offset5					Rb			Rd			Load/store halfword
11	1	0	0	1	L	Rd			Word8								SP-relative load/store
12	1	0	1	0	SP	Rd			Word8								Load address
13	1	0	1	1	0	0	0	0	S	SWord7							Add offset to stack pointer
14	1	0	1	1	L	1	0	R	Rlist								Push/pop registers
15	1	1	0	0	L	Rb			Rlist								Multiple load/store
16	1	1	0	1	Cond				Soffset8								Conditional branch
17	1	1	0	1	1	1	1	1	Value8								Software Interrupt
18	1	1	1	0	0	Offset11											Unconditional branch
19	1	1	1	1	H	Offset											Long branch with link

3. 安腾(Intanium)处理机的指令格式

Figure 3-15. Bundle Format



指令格式：

OpCode	Reg 1	Reg 2	Reg 3	Predicate
14	7	7	7	6

1个bundle有三条指令和一个template，Template指示这三条指令的类型及stop的位置；Predicate指示本指令的启动时刻。

- 1个group可以有任意条指令，同一group内的指令可以同时执行。
- 一个时钟周期可发射2个bundle。
- 共9个功能部件：2个整数部件i、2个浮点部件f、 2个访存部件m、3个分支部件b。

```
{ .mii
    add r1 = r2, r3
    sub r4 = r4, r5 ;;
    shr r7 = r4, r12 ;;
}
{ .mmi
    ld8 r2 = [r1] ;;
    st8 [r1] = r23
    tbit p1,p2=r4,5
}
{ .mbb
    ld8 r45 = [r55]
(p3)br.call b1=func1
(p4)br.cond Label1
}
{ .mfi
    st4 [r45]=r6
    fmac f1=f2,f3
    add  r3=r3,8 ;;
}
```

Figure 3. Example instruction groups.

条件执行方式的简单例子

- 一般指令系统

```
        CMP  a, b
        BEQ   EQ
        X=3
        JMP   END
EQ:    X=4
END:
```

- 条件执行代码

```
cmp.eq  p1,p2=a,b
(p1) x=3
(p2) x=4
```

C语言CASE语句的例子

C Code Example

```
switch (type)
{  case 'a':
    type=type+10;    break;
   case 'b':
    type=type+20;    break;
   default:
    break;  }
```

汇编代码:

```
 mov1 r1=type
 ld4  r2=[r1];;
 cmp.eq  p1,p2=‘a’,r2
 cmp.eq  p3,p4=‘b’,r2;;
 (p1)add  r2=10,r2
 (p3)add  r2=20,r2;;
 st4[r1]=rs
 default::
```

2.4 指令系统的功能设计

2.4.1 基本指令系统：通用计算机必须有**5**类基本指令

2.4.2 指令系统的性能：完整性、规整性、高效率和兼容性

2.4.3 指令系统的优化设计：**CISC**、**RISC**和**VLIW**

2.4.1 基本指令系统

通用计算机必须有 5 类基本指令

- 数据传送类指令
- 运算类指令
- 程序控制指令
- 输入输出指令
- 处理机控制和调试指令

1. 数据传送类指令

- 由如下三个主要因素决定：

1） 数据存储设备的种类：通用寄存器、主存储器、堆栈

2） 数据单位：字、字节、位、数据块等

3） 采用的寻址方式

- 例如，考虑数据存储设备的种类：

寄存器→寄存器　　寄存器→主存储器

寄存器→堆栈　　主存储器→寄存器

主存储器→主存储器　　主存储器→堆栈

堆栈→寄存器　　堆栈→主存储器

2. 运算类指令 ：

考虑四个因数的组合：

- 操作种类：加、减、乘、除、与、或、非、异或、比较、移位、检索、转换、匹配、清除、置位等
- 数据表示：定点、浮点、逻辑、十进制、字符串、向量等
- 数据长度：字、双字、半字、字节、位、数据块等
- 数据存储设备：寄存器、主存储器、堆栈等

以寄存器加法指令为例，一般设置如下几种：

- 寄存器-寄存器型的定点单字长加法指令
- 寄存器-寄存器型的定点双字长加法指令
- 寄存器-寄存器型的定点半字加法指令
- 寄存器-寄存器型的字节加法指令
- 寄存器-寄存器型的浮点单字长加法指令
- 寄存器-寄存器型的浮点双字长加法指令
- 寄存器-寄存器型的单字长逻辑加法指令
- 寄存器-寄存器型的定点向量加法指令
- 寄存器-寄存器型的浮点向量加法指令

以移位指令为例：需要组合以下三个因素：

1） 移位方向：左移(L)、右移(R)

2） 移位种类：算术移位(A)、逻辑移位(L)、循环移位(R)

3） 移位长度：单字长(S)、双字长(D)

组合起来共有：3×2×2＝12种，其中，逻辑左移与算术左移相同，一般机器中要设置10条移位指令

一般机器中要设置10条移位指令

SLAS	单字长算术左移
SRAS	单字长算术右移
SLLS(SRLS)	单字长逻辑左移,单字长算术左移
SLRS	单字长循环左移
SRRS	单字长循环右移
SLAD	双字长算术左移
SRAD	双字长算术右移
SLLD(SRLD)	双字长逻辑左移,双字长算术左移
SLRD	双字长循环左移
SRRD	双字长循环右移

3. 程序控制指令

- 有多种转移指令：
  - 无条件转移，条件转移指令
  - 程序调用与返回指令
  - 循环控制指令
  - ……

- 无条件转移指令
  - 局部无条件转移：采用相对寻址方式，转移范围一般在+127到-128之间
  - 全局无条件转移：在整个寻址空间内转移
- 转移条件
  - 零(Z)、正负(N)、进位(C)、溢出(V)及其组合
  - 由条件转移指令之前的指令产生条件码
  - 由条件转移指令本身产生转移条件
  - 多组条件码

一般条件转移指令

- 条件码由转移指令之前的指令产生，对指令流水线的影响小。
- 例如：

```
BEQ    ADR   ;等于零转移
BLS    ADR   ;小于转移
BLEQ   ADR   ;小于等于转移
BLSU   ADR   ;不带符号小于转移
BLEQU  ADR   ;不带符号小于等于转移
BCC    ADR   ;没有进位转移
BVC    ADR   ;没有溢出转移
```

复合条件转移指令

- 代替2条指令，首先进行运算，并根据运算的结果决定是否转移
- 不需要条件码，与高级语言一致。例如：

```
DNB   R   ADR    ;R←(R)-1,如果R≠0转移
INB   R   ADR    ;R←(R)+1,如果R≠0转移
JEQ   R1，R2，ADR  ;如果(R1)=(R2)转移
JAD   EQ，Rd，Rs，ADR  ;Rd←(Rd)+(Rs),
                        ;如果(Rd)=0转移
```

循环控制指令

- 用1条指令完成循环的控制
- 代替3条指令的功能：运算、比较和转移。
- 例如： JL Rs，Rz，Ri，ADR
  - Rs中存放循环变量，Rz中存放循环终值，Ri中存放循环的步长。
  - 地址个数太多时，可以把其中的1个或几个地址隐含起来
  - 例如，在IBM370下列机中，Ri隐含，循环步长放在与Rz紧相邻的下一寄存器中。

隐含转移指令

- 应用场合：用于特殊的IF..THEN..结构中，THEN部分只有一条指令
- 实现方法：把IF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

  COMP >=,Ra,Rb ;若(Ra)>=(Rb)则取消

  INC　Rb
- 达到的效果：不需要专门的转移指令，程序执行效率高。

程序调用和返回指令

- 本身可以不带条件，也可以带有条件

CALL		转入子程序
RETURN		从子程序返回
CALL	CC	当条件CC满足时转入子程序
RETURN	CC	当条件CC满足时从子程序返回

中断控制指令：

开中断、关中断

改变屏蔽

中断返回

自陷等

4. 输入输出指令

- 启动、停止、测试设备，数据输入、输出等
- 采用单一的直接寻址方式，
- 在多用户或多任务环境下，输入输出指令属于特权指令

5. 处理机控制和调试指令

- 处理机状态切换指令，处理机有多个状态
- 硬件和软件的调试指令
- 硬件调试指令：开关状态读取等
- 软件调试指令：断点设置、跟踪，自陷指令等

2.4.2 指令系统的性能

➢完整性、规整性、高效率和兼容性

1. 完整性是指应该具备的基本指令种类

   • 如：通用计算机必须有 5 类基本指令

2. 规整性包括对称性和均匀性

➢对称性：所有的寄存器同等对待，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

3. 高效率：

- 指令的执行速度要快
- 指令的使用频度要高
- 各类指令要有一定的比例，如：运算类指令占40%以上，数据传送类指令占30%等。

4. 兼容性：

- 在同一系列机内，指令系统，包括寻址方式和数据表示等保持基本不变；
- 可以适当增加指令、增加寻址方式，增加数据表示等；但不能减少任何已有的……。

2.4.3 指令系统的优化设计

- 优化指令系统设计的3个阶段：
  - 60年代至70年代中期：CISC，复杂指令系统
  - 70年代后期至现在：RISC精简指令系统
  - 80年代初期至现在：VLIW：
- 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。

1. 复杂指令系统计算机

- **CISC(Complex Instruction Set Computer）**
- 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
- 优化的途径：
  - 面向目标代码
  - 面向高级语言
  - 面向操作系统

2. 精简指令系统计算机

- **RISC(Reduced Instruction Set Computer）**
- 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

3. 超长指令字

- **VLIW (Very Long Instruction Word)**
- 一种显式指令级并行指令系统
- 二维程序结构
- 指令级并行度高

2.5 RISC指令系统

三类指令系统：**CISC、RISC、VLIW**

2.5.1 从CISC到RISC

2.5.2 RISC的定义与特点

2.5.3 RISC思想的精华

2.5.4 RISC的关键技术

2.5.1 从CISC到RISC

70年代，指令系统已经非常复杂

机　　型 (生产年代)	IBM370/168 (1973)	VAX-11 (1978)	iAPX 432 (1982)	Dorado (1978)
指令种类	208	303	222	270
微程序容量	420K	480K	64K	136K
指令长度	16-48	16-456	6-321	8-24
采用的工艺	ECL MSI	TTL MSI	NMOS VLSI	ECL MSI
指令操作类型	存储器-存储器 存储器-寄存器 寄存器-寄存器	存储器-存储器 存储器-寄存器 寄存器-寄存器	面向堆栈 存储器-存储器	面向堆栈
Cache 容量	64KB	64KB	0	64KB

- 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. 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

2. VLSI技术的发展引起的问题

- VLSI工艺要求规整性，RISC正好适应了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
- 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。

3. 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
  - CISC处理机的指令平均执行周期都在4以上
- 在CISC中，增强指令系统功能，简化了软件，硬件复杂了
- 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

2.5.2 RISC的定义与特点

卡内基梅隆(Carnegie Mellon)大学论述RISC的特点如下：

(1)大多数指令在单周期内完成

(2)LOAD/STORE结构

(3)硬布线控制逻辑

(4)减少指令和寻址方式的种类

(5)固定的指令格式

(6)注重编译的优化

90年代初, **IEEE的Michael Slater对RISC描述：**

1) RISC为提高流水线效率, 应具有下述特征：

- **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**
- **只有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**

2) 为使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**

2.5.3 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： $P = I \cdot CPI \cdot T$
  - 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

硬件方面：

采用硬布线控制逻辑

减少指令和寻址方式的种类

使用固定的指令格式

采用LOAD/STORE结构

指令执行过程中设置多级流水线等

软件方面：

十分强调优化编译技术的作用。

RISC设计思想也可以用于CISC中

- x86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，超标量处理机、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。

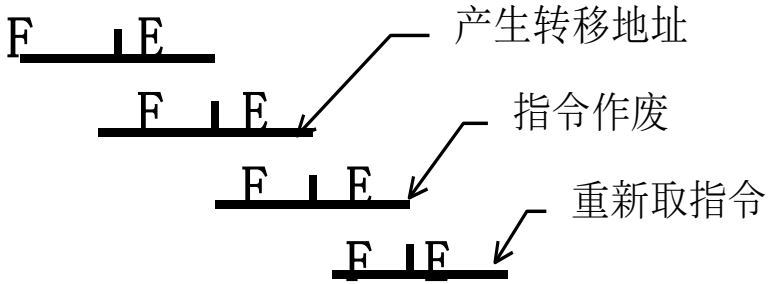
2.5.4 RISC的关键技术

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。

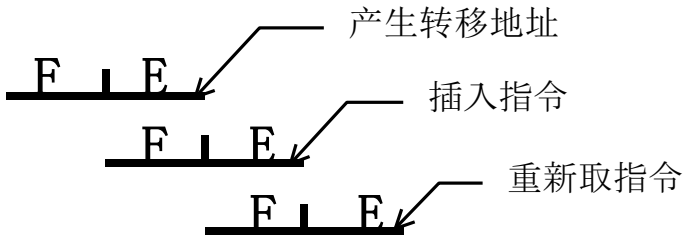
- 无条件转移指令的延迟执行

```
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  EXIT        ;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 EXIT         ;如果(R3)=(R4)则转移
3：MOVE R1,R2       ;被插入的指令
    ……
NEXT: MOVE  R4, A
```

- 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. 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）

- 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。

例子：

```
        R  R  R
        ……          } “IF”部分的程序代码
        S  S  S

        COMP    R1, R2, THRU
        T  T  T
        ……          } “THEN”部分的程序代码
        U  U  U
THRU:   V  V  V
```

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

3. 重叠寄存器窗口技术

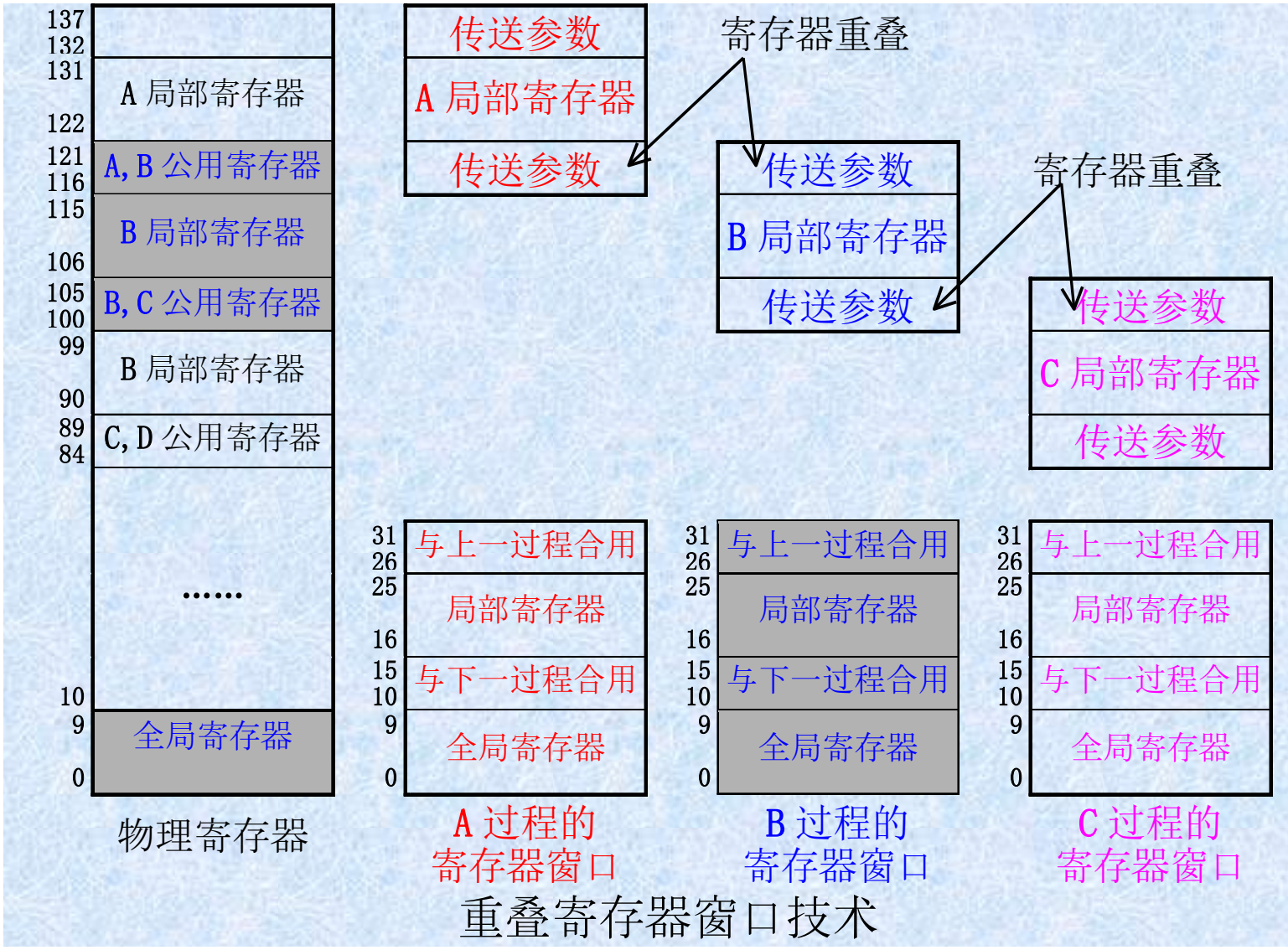
- **Overlapping Register Window**
- 原因：在RISC中，子程序比CISC中多，因为传送参数而访问存储器的信息量很大
- 美国加洲大学伯克利分校的F Baskett提出
- 实现方法：设置一个数量比较大的寄存器堆，并把它划分成很多个窗口。在每个过程使用的几个窗口中：
  - 有一个窗口是与前一个过程共用
  - 有个窗口是与下一个过程共用

例子：（在RISC II中采用的方法）

目前，SUN公司的SPARC、SuperSPARC、UtraSPARC等处理机，把最后一个过程与第一个过程的公用寄存器重叠起来，形成一个循环圈。

效果：可以减少大量的访存操作。

另外，要在主存中开辟一个堆栈，当调用层数超过规定层数（寄存器溢出）时，把益出部分的寄存器中内容压入堆栈。


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

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	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
调整前的指令序列	调整后的指令序列

5. 以硬件为主固件为辅

- 固件的主要缺点是：执行速度低
- 固件的主要优点是：
  - 便于实现复杂指令，便于修改指令系统
- 以硬联逻辑为主来实现指令系统

2.5.5 RISC优化编译技术

RISC对编译器带来的方便主要有：

1） 指令系统比较简单、对称、均匀，指令选择工作简单

2） 选择寻址方式的工作简单

3） 因为采用LOAD/STORE方式，省去了是否生成访问存储器指令的选择工作。

4） 由于大多数指令在一个周期内执行完成，为编译器调整指令序列提供了极大的方便。

RISC对编译器造成的困难主要有：

1） 必须精心安排每一个寄存器的用法，以便充分发挥每一个通用寄存器的效率，尽量减少访问主存储器的次数。

2） 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等。

3） 要设计复杂的子程序库，RISC的子程序库通常要比CISC的子程序库大得多。

2.6 VLIW指令系统

2.6.1 什么是**WLIW**

2.6.2 指令级并行技术

2.6.3 **VLIW**的主要特点

2.6.4 **VLIW**处理机

2.6.5 目标代码兼容问题

2.6.1 什么是WLIW

1. VLIW (Very Long Instruction Word) 的背景

- 由美国J. A. Fisher教授于1981年首先提出
- 最初来源于水平微程序
- 由J. A. Fisher创建的Mutiflow公司研制了的世界上第一台VLIW处理机TRACE28/300。
- 一条指令中包含有多个能够同时执行的操作
- TRACE28/300处理机的一条超长指令中最多有28条可以同时执行的指令。
- 算法和编译技术是关键
- 在下一代处理机中将普遍采用

2. 什么是VLIW指令系统

- 一种显式指令级并行指令系统
- 在一条VLIW指令中包含有多个相同或不同的操作字段（每个操作字段的功能相当于一般处理机中的一条指令）
- 每个操作字段能够分别独立控制各自的功能部件同时工作
- 二维程序结构
- 指令级并行度高

2.6.2 指令级并行

- 提出VLIW指令系统的主要目的是要开发程序中的指令级并行性(Instruction Level Parallelism)
- 超标量(Superscalar)处理机 依靠设置多条指令流水线，并通过同时发射多条指令来提高处理机的运算速度
- 超流水线(Superpipelining)处理机 通过分时使用同一条指令流水线的不同部分来提高处理机的运算速度
- VLIW处理机

2.6.3 VLIW的主要特点

1.采用显式并行指令计算(EPIC：Explicitly Parallel Instruction Computing)方式。

- 在VLIW处理机上运行的程序是一个二维指令矩阵，每一行上的所有操作组成一条超长指令，他们之间没有数据相关、控制相关和功能部件冲突，这些指令可以在VLIW处理机上同时执行
- 超标量处理机和超流水线处理机通常采用隐式并行指令方式。程序是一维线性的指令序列，每条指令中一般只包含一个操作。

2. 指令级并行度高

- 超标量处理机和超流水线处理机的指令级并行度一般为2左右，通常不超过4
- 目前多数VLIW处理机的指令级并行度在4至8之间，有的已经达到几十
- 由于在VLIW中通过并行编译器来开发程序中的指令级并行性，可以在一个循环、一个函数、甚至整个程序中寻找指令级并行性，而且，可以采用软件流水、循环展开等指令级并行度很高的方法充分开发程序中的多种并行性

3. 硬件结构规整、简单

- VLIW处理机主要由很规则的寄存器、存储器、运算部件和数据通路等组成，不规则的控制器很简单，而且，不需要复杂的指令并行调度窗口及多发射机制等。

4. 编译器的实现难度大

- VLIW并行编译器主要依靠指令级并行算法、数据相关性分析算法、寄存器分配算法及并行编译技术等来显式开发程序中的指令级并行性，从而提高处理机的运行速度。要研制指令级并行度高的编译器难度很大。

2.6.4 VLIW处理机

1. 安腾(Intanium)处理机

- Intel公司与HP公司联合研制
- 在Intel公司称为IA-64处理机
- 安腾(Intanium)处理机有自己独立的系统软件和应用软件

2. DAISY (Dynamically Architected Instruction Set from Yorktown) 处理机

- 由IBM公司研制
- 采用动态二进制转换技术实现与X86处理机兼容

3. Crusoe处理机

- 由Transmeta公司研制
- 已经大量应用于笔记本计算机中，一个重要特点是功耗很低。
- 采用动态二进制转换技术把X86通用处理机的程序直接映射到Crusoe处理机的VLIW结构中执行。

4. 嵌入式、DSP、JAVA虚拟机

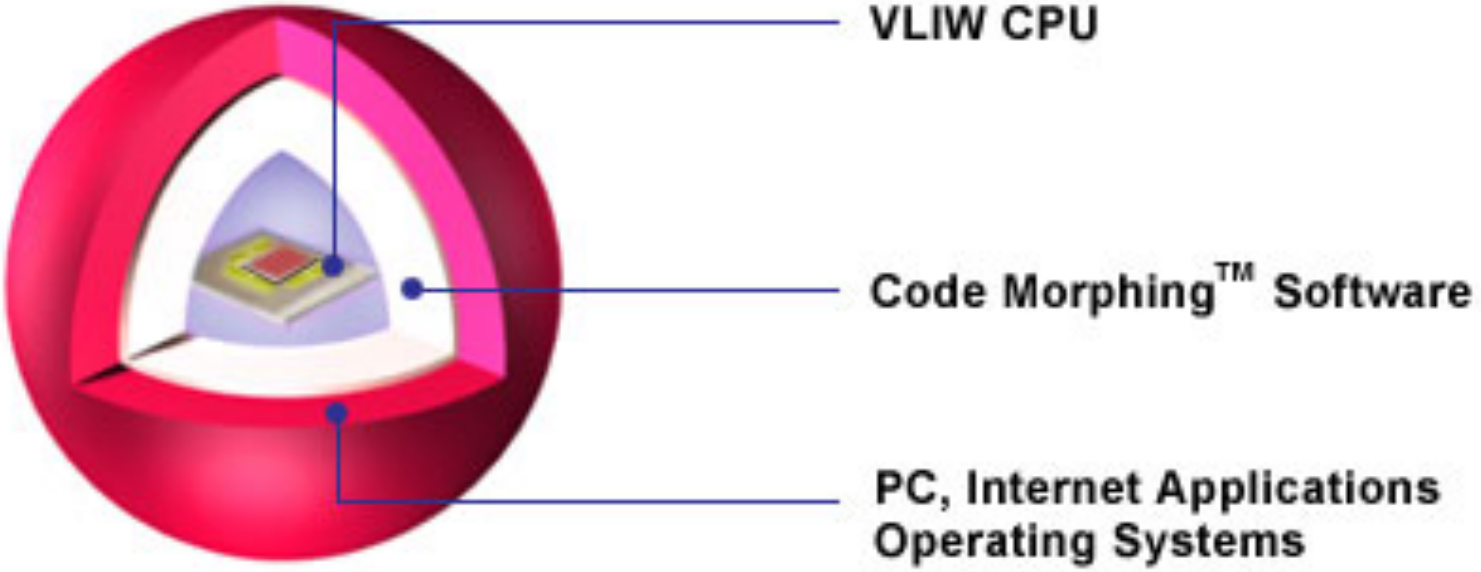
- 很多专用处理机采用VLIW体系结构

2.6.5 目标代码兼容问题

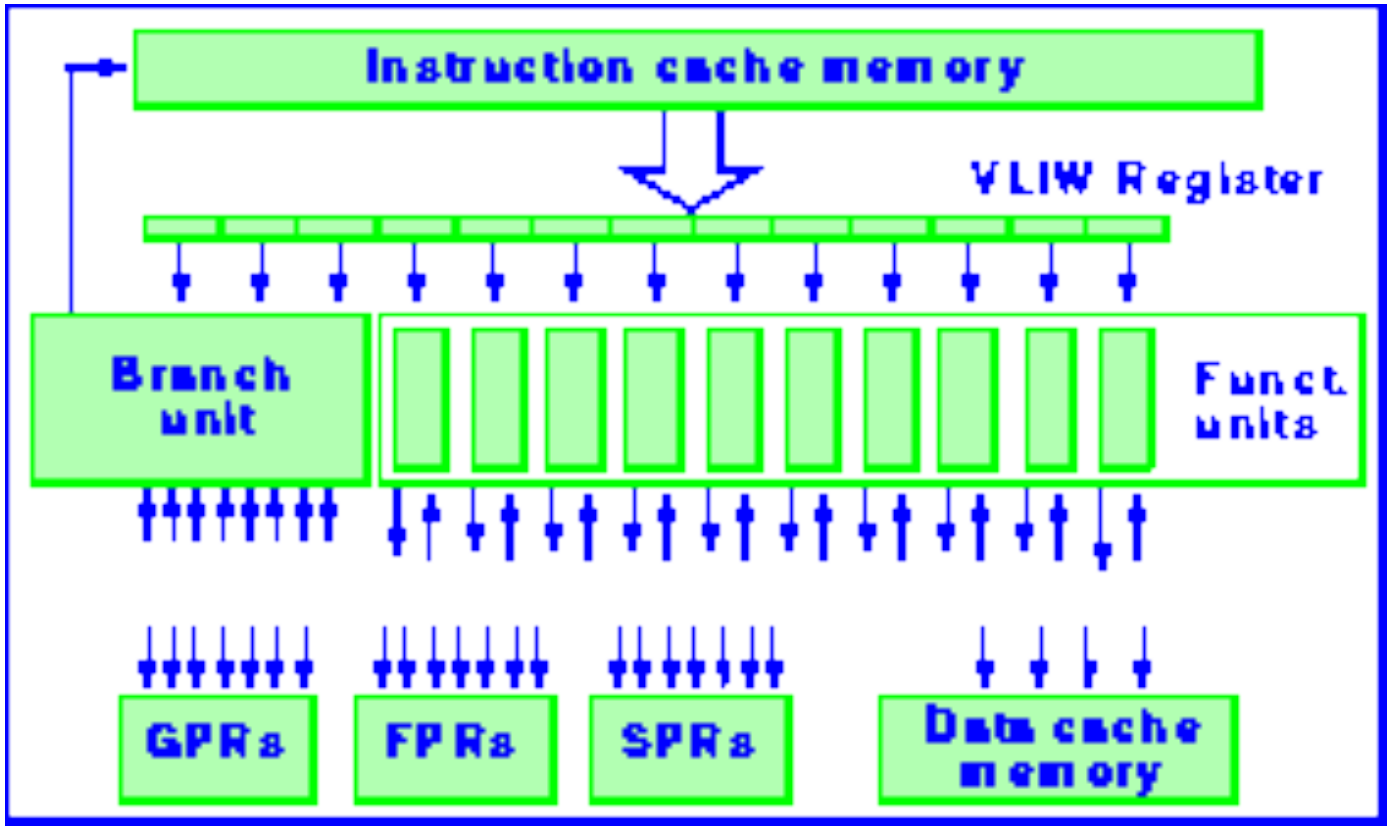
1. 动态代码转换技术：两个成功的先例

- IBM公司推出了**开放源代码DAISY**，它不仅可以实现IBM的VLIW处理器与X86处理机之间的二进制兼容，还可以实现PowerPC、S/390、IBM的Java虚拟机与VLIW处理器之间的二进制兼容
- Transmeta公司推出了“**代码映射软件**”（Code Morphing Software），这种软件可以保证Transmeta公司的VLIW处理机Crusoe能够与X86处理机之间实现二进制兼容

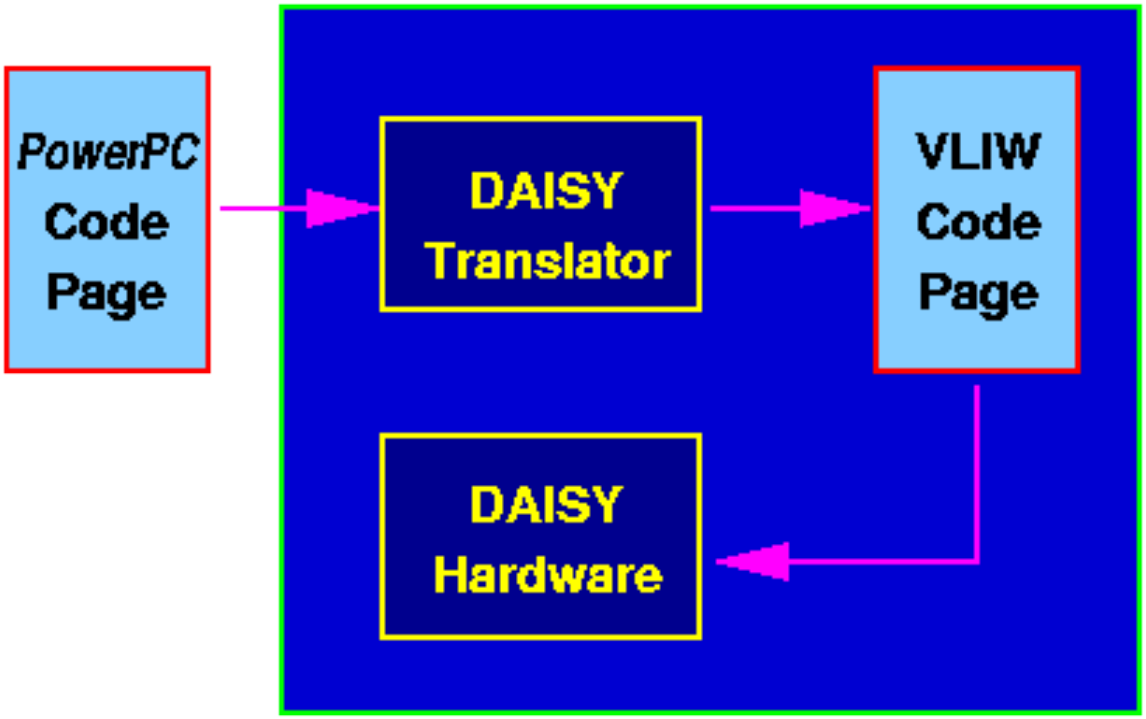
Crusoe处理机动态转换技术



2. IBM公司的VLIW处理机

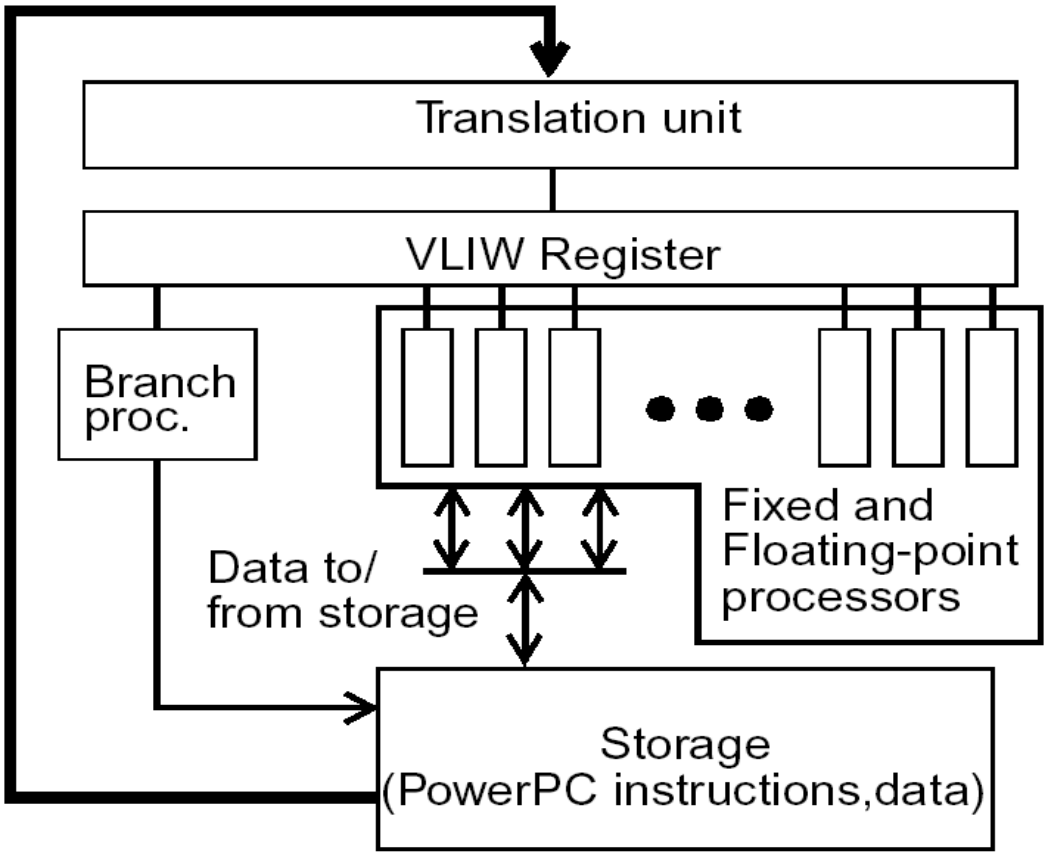


IBM公司推出的开放源代码DAISY



DAISY Architecture

IBM公司的PowePC到VLIW的转换



3. 可执行代码的并行编译技术

有动态重编译和静态重编译两种方法：

- 动态重编译：**DAISY和Code Morphing**
- 静态重编译：还没有商业化的先例；难度很大，正在研究中。
- 目标：一种系列机的目标代码重新编译到本系列机或另一种系列机的并行目标代码。
- 方法：串行目标代码→中间表示形式→数据相关性分析→并行调度→并行目标代码。

意义：

1） 编译器不再受限于程序采用的编程语言、开发工具及开发环境，不同的编程语言都呈现为统一的二进制视图。

2） 将大量已有的二进制目标代码直接编译成能够在最新的超标量、超流水线或VLIW处理机上执行的并行可执行代码，充分利用最新处理机的硬件资源，大幅度缩短程序的执行时间。

3） 不同机器之间的软件能够很方便地移植。

4） 系统结构与软件可以同时并行发展，克服目前制约系统结构发展的限制因素。

关键技术

1）可执行代码的指令流重构和控制流重构，如非规则循环结构的整理和重构等。

2）中间代码的表示形式和表示方法。

3）如何把各种可执行代码转换成中间代码。

4）可执行代码的数据相关性分析，进行有效数组相关性分析的前提是正确重构数组下标表达式。

5）系统调用和中断指令在并行优化中的正确性及效率。

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. 浮点数的舍入方法和警戒位位数的设置
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