



计算机系统结构

第二章 指令系统

主 讲：刘超

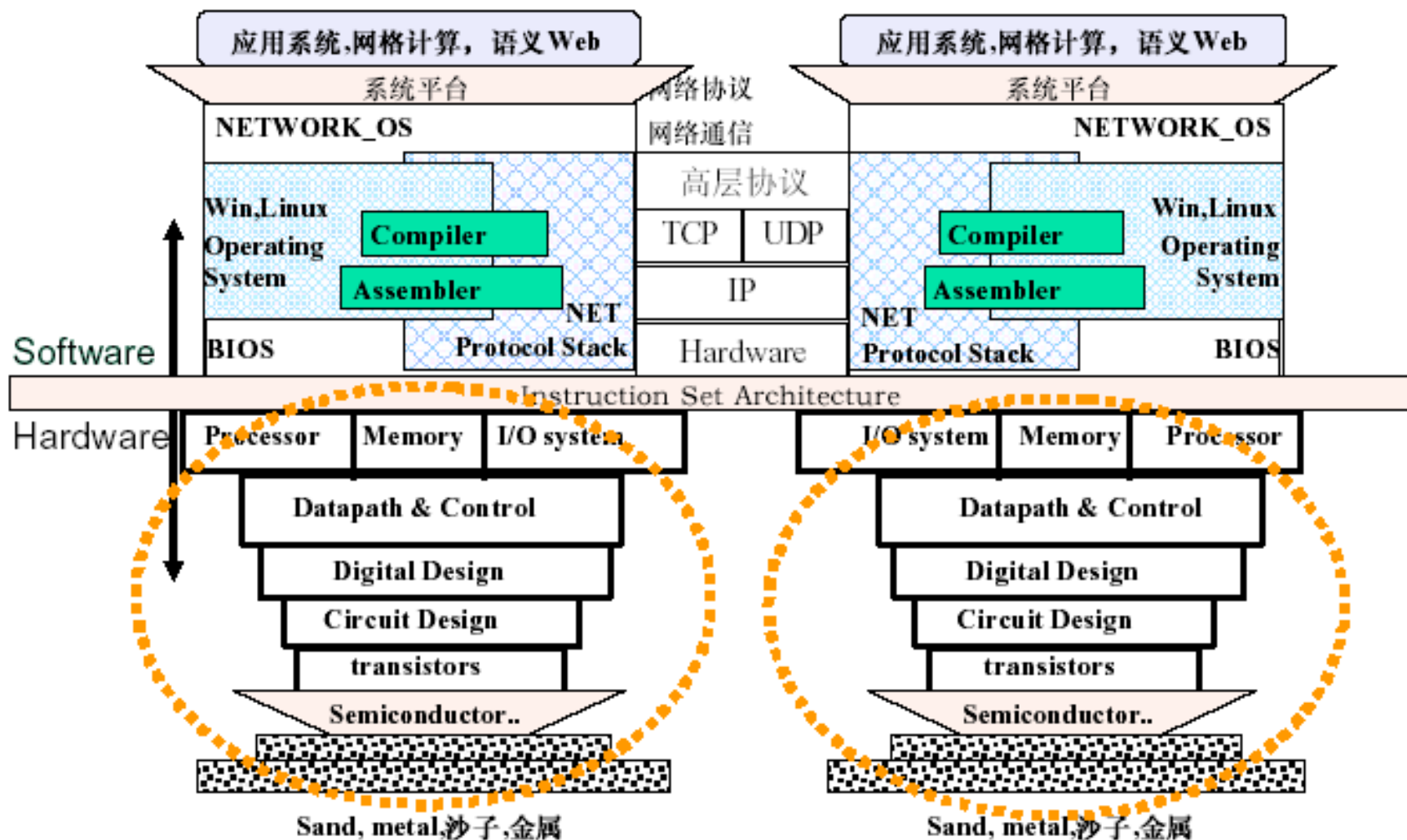
中国地质大学（武汉）计算机学院

第二章 指令系统

- 2.1 指令系统结构的分类
- 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 指令系统的发展方向和优化
- 2.6 小结
- 2.7 习题

指令系统的地位

- 指令系统是计算机系统中软件和硬件分界面的一个重要标志。



Instruction Set Architecture

- **Critical interface between hardware and software**

- Standardizes instructions, machine language bit patterns, etc.
- Advantage: **different implementations of the same architecture**
- Disadvantage: **sometimes prevents using new innovations**

■ Examples	(versions)	Introduced in
■ Intel	(8086, 80386, Pentium, ...)	1978
■ IBM Power	(Power 2, 3, 4, 5)	1985
■ HP PA-RISC	(v1.1, v2.0)	1986
■ MIPS	(MIPS I, II, III, IV, V)	1986
■ Sun Sparc	(v8, v9)	1987
■ Digital Alpha	(v1, v3)	1992
■ PowerPC	(601, 604, ...)	1993

什么是指令系统？

- 在计算机系统的设计中，由**硬件**设计人员采用各种技术**实现指令系统**，由**软件**设计人员**使用指令系统**来编制各种系统软件和应用软件。用这些软件来填补用硬件实现的指令系统与编程语言之间的语义差距。因此，**指令系统**：

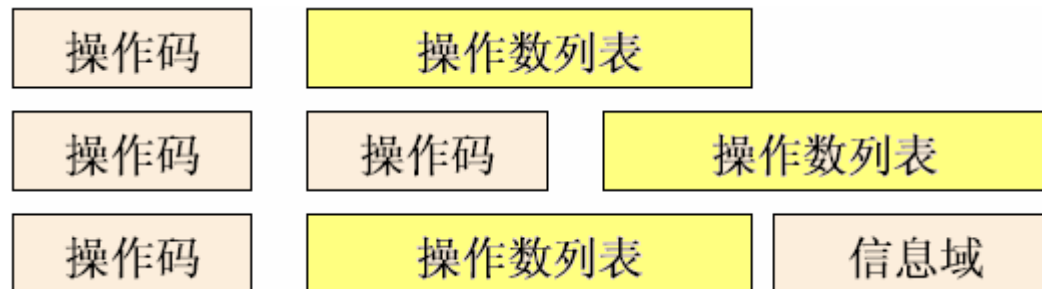
- 是机器语言程序设计者看到的机器的主要属性。
- 是软、硬件的主要界面。
- 很大程度上决定了计算机系统具有的基本功能。

什么是指令？

■ 指令的含义：

- 一般含义：指明要执行的**操作**以及**操作的对象**。
- 组成：由**操作码**和**操作数**构成。

■ 例：



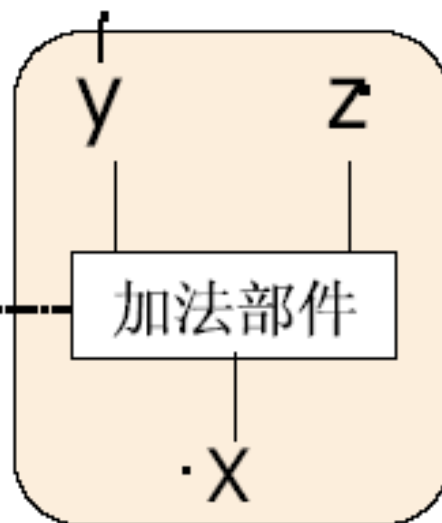
指令如何实现？

$$X = y + z$$

0.2345

0.3456×10^{34}

计算机如何实现？
---计算机指令



操作码

操作数1

操作数2

操作数3

操作码对应于功能部件
操作数对应于部件的输入
和输出。

通常，操作码（动词）放
在前面

设计指令系统应关心哪些问题？

$$X = y + z$$

0.2345

0.3456×10^{34}

操作码

操作数1

操作数2

操作数3

操作数1

操作数2

操作码

操作数3

数据表示

在计算机内
如何表示？

存在哪里？
如何找到？

地址分配
地址码个数
寻址方法
地址码优化

硬件支持
那些运算？

指令功能
操作码优化

数据类型、数据表示与数据结构

- **数据类型**是指计算机系统中可以使用和处理的各种数据的类型，例如，字符、布尔数、整数、实数、串、向量、栈、链表、队列、阵列、树、表和文件等。
- **数据表示**研究的是计算机硬件能够直接识别的、可以被指令系统直接调用的那些数据类型。
 - 常用的、简单的、容易实现
- **数据结构**研究的是面向系统软件、面向应用领域所需处理的各种数据类型，以及这些数据类型之间的逻辑物理关系，并给出相应的算法。
- **系统结构设计**确定数据结构和数据表示的界面，即决定在所有的数据类型中，哪些由硬件实现，哪些用软件实现，并研究它们的实现方法等。

Instruction Set Architecture (ISA)

- Is a subset of Computer Architecture
- Definition by Amdahl, Blaaw, and Brooks – 1964

“... the attributes of a [computing] system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.”

- An ISA encompasses ...
 - Instructions and Instruction Formats
 - Data Types, Encodings, and Representations
 - Programmable Storage: Registers and Memory
 - Addressing Modes: Accessing Instructions and Data
 - Handling Exceptional Conditions

2.1 指令系统结构的分类

- 区别不同指令系统结构的**主要因素**

CPU中用来存储操作数的存储单元的类型

- CPU中用来存储操作数的存储单元

- 堆栈

- 累加器

- 通用寄存器组

- 将指令系统的结构分为三种类型

2.1 指令系统结构的分类

- 区别不同指令系统结构的**主要因素**

CPU中用来存储操作数的存储单元的类型

- CPU中用来存储操作数的存储单元

- 堆栈

- 累加器

- 通用寄存器组

- 将指令系统的结构分为三种类型

2.1 指令系统结构的分类

- 堆栈结构（不能随机访问）
- 累加器结构（累加寄存器保存一个中间结果）
- 通用寄存器结构

根据操作数的来源不同，又可进一步分为：

- 寄存器-存储器结构（RM结构）
（操作数可以来自存储器）

- 寄存器-寄存器结构（RR结构）
（所有操作数都是来自通用寄存器组）

也称为load-store结构，这个名称强调：只有load指令和store指令能够访问存储器。

2.1 指令系统结构的分类

4. 对于不同类型的结构，操作数的位置、个数以及操作数的给出方式（显式或隐式）也会不同。

- 显式给出：用指令字中的操作数字段给出
- 隐式给出：使用事先约定好的单元

2.1 指令系统结构的分类

例：表达式 $Z=X+Y$ 在4种类型指令系统结构上的代码。 假设：X、Y、Z均保存在存储器单元中，并且不能破坏X和Y的值。

堆 栈	累加器	寄存器（RM型）	寄存器（RR型）
push X	load X	load R1, X	load R1, X
push Y	add Y	add R1, Y	load R2, Y
add	store Z	store R1, Z	add R3, R1, R2
pop Z			store R3, Z

2.1 指令系统结构的分类

5. 通用寄存器型结构

- 现代指令系统结构的主流
- 在灵活性和提高性能方面有明显的优势
 - 跟其它的CPU内部存储单元一样，寄存器的访问速度比存储器快。
 - 对编译器而言，能更加容易、有效地分配和使用寄存器。

2.1 指令系统结构的分类

- 寄存器可以用来存放变量。

- (1) 减少对存储器的访问，加快程序的执行速度；（因为寄存器比存储器快）

- (2) 用更少的地址位（相对于存储器地址来说）来对寄存器进行寻址，从而有效地减少程序的目标代码的大小。

2.1 指令系统结构的分类

6. 根据ALU指令的操作数的两个特征对通用寄存器型结构进一步细分

- ALU指令的操作数个数

- 3个操作数的指令

- 两个源操作数、一个目的操作数

- 2个操作数的指令

- 其中一个操作数既作为源操作数，又作为目的操作数。

- ALU指令中存储器操作数的个数

- 可以是0~3中的某一个，为0表示没有存储器操作数。

2.1 指令系统结构的分类

7. ALU指令中操作数个数和存储器操作数个数的典型组合

ALU指令中存储器操作数的个数	ALU指令中操作数的最多个数	结构类型	机器实例
0	3	RR	MIPS, SPARC, Alpha, PowerPC, ARM
1	2	RM	IBM 360/370, Intel 80x86, Motorola 68000
	3	RM	IBM 360/370
2	2	MM	VAX
3	3	MM	VAX

2.1 指令系统结构的分类

8. 通用寄存器型结构进一步细分为3种类型

- 寄存器—寄存器型（RR型）
- 寄存器—存储器型（RM型）
- 存储器—存储器型（MM型）

9. 3种通用寄存器型结构的优缺点

表中 (m , n) 表示指令的 n 个操作数中有 m 个存储器操作数。

指令系统结构类型	优 点	缺 点
寄存器—寄存器型 (0, 3)	指令字长固定，指令结构简洁，是一种简单的代码生成模型，各种指令的执行时钟周期数相近。	与指令中含存储器操作数的指令系统结构相比，指令条数多，目标代码不够紧凑，因而程序占用的空间比较大。
寄存器—存储器型 (1, 2)	可以在ALU指令中直接对存储器操作数进行引用，而不必先用load指令进行加载。容易对指令进行编码，目标代码比较紧凑。	指令中的两个操作数不对称。在一条指令中同时对寄存器操作数和存储器操作数进行编码，有可能限制指令所能够表示的寄存器个数。指令的执行时钟周期数因操作数的来源（寄存器或存储器）不同而差别比较大。
存储器—存储器型 (2, 2) 或 (3, 3)	目标代码最紧凑，不需要设置寄存器来保存变量。	指令字长变化很大，特别是3操作数指令。而且每条指令完成的工作也差别很大。对存储器的频繁访问会使存储器成为瓶颈。这种类型的指令系统结构现在已不用了。

2.2 寻址技术

- 1) 什么是寻址技术?
- 2) 编址方式
- 3) 编址单位
- 4) 寻址方式

1) 什么是寻址技术?

■ 寻址技术

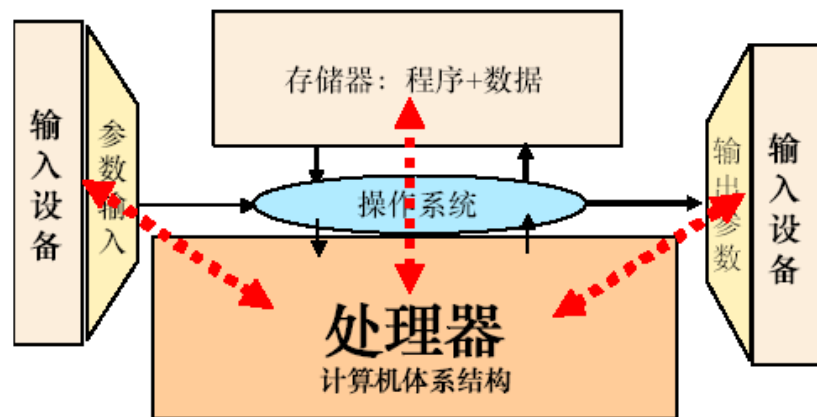
- 寻找数据以及其它信息的地址
- 指令系统是软硬件的主要分界面，而寻址技术是指令格式设计中要重点考虑的一个方面。
- 涉及编址方式，寻址方式，定位方式

■ 寻址对象

- 主存，寄存器，堆栈，外设
- 以主存储器为主

■ 计算机如何与这些设备通信

- 给这些设备编号，分配一个地址
- 按照分配的地址通信



2) 编址方式

■ ① 三个零地址空间

- 寄存器、主存、I/O设备分别编址；

■ ② 两个零地址空间

- 寄存器独立编址
- 主存、I/O设备统一编址；

■ ③ 一个零地址空间

- 所有存储设备统一编址；

■ ④ 隐含编址方式

- 无零地址空间。如在Cache中、堆栈计算机中。

3) 编址单位

- **字编址**(按存储字长编址)

- 字编址是指每个编址单位与一次可访问的数据存储单元相一致。

- **字节编址**

- 字节编址是指每个编址单位都是一个字节。

- **位编址**

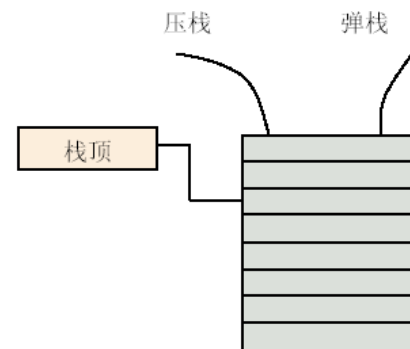
- 位编址是指每个编址单位都是一个二进制位

4) 指令和数据的寻址方式

- 什么是寻址方式？
- 指令的寻址方式
- 操作数寻址方式

什么是寻址方式？

- 寻找指令或操作数存储单元的方法称**寻址方式**，分类如下
 - **立即寻址**
 - 直接给出操作数
 - **寄存器寻址**
 - **主存寻址**
 - **直接寻址**：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（有效地址），此方法地址码较长，而且修改数据地址不便
 - **间接寻址**：指令中给出的是操作数地址的地址
 - **变址寻址**：指令中使用变址寄存器。指令操作数的有效地址是变址寄存器的值加上一个偏移量
 - **堆栈寻址**
 - 地址是隐含的，指令中不必给出操作数的地址，
 - 支持高级语言，有利于编译
 - 程序代码量短
 - 支持嵌套和递归
 - 其它，中断等



指令的寻址方式

顺序执行：PC寻址

非顺序执行：转移指令

如jump \$1000

bgt R1

——条件转移vs无条件转移

——相对转移vs绝对转移

状态寄存器SR：N (正负号), Z (全零), V (溢出), C (进位)等

操作数寻址方式

1. **隐含方式**。如ADD A中的累加器
2. **立即寻址**。如INT #3
3. **寄存器寻址**。如INC R1

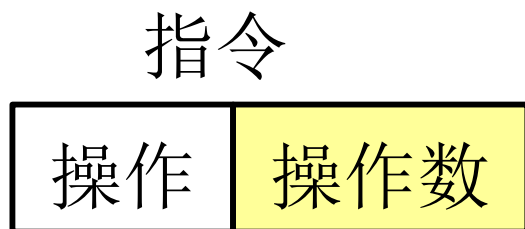


图 立即寻址

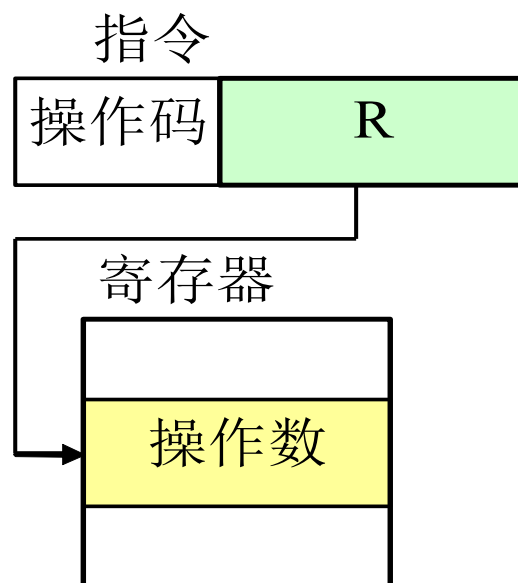


图 寄存器寻址

操作数寻址方式（续）

■ 4. 直接寻址。如INC 1000

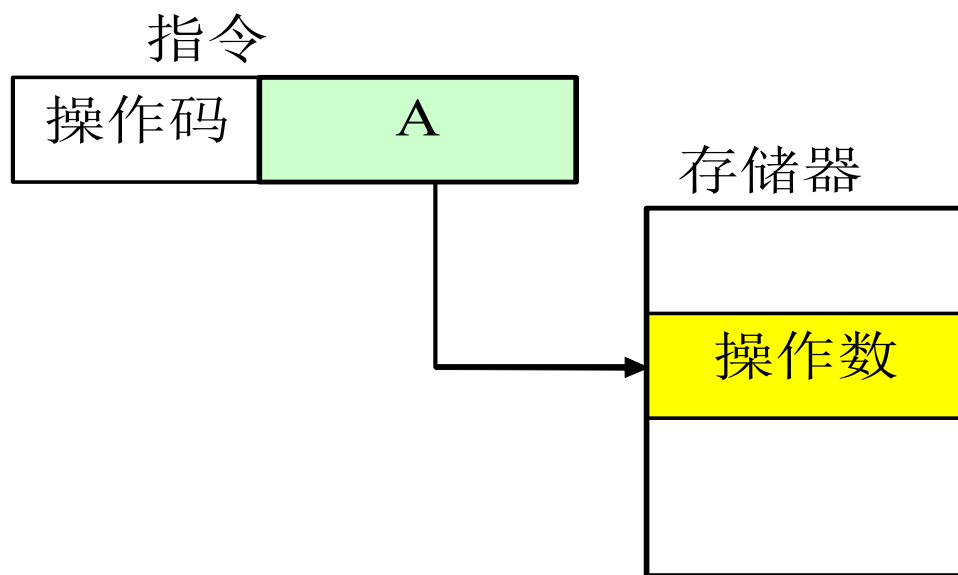


图 直接寻址

操作数寻址方式（续）

5. 间接寻址

寄存器间接寻址 如INC (R1),
存储器间接寻址 如INC (1000)

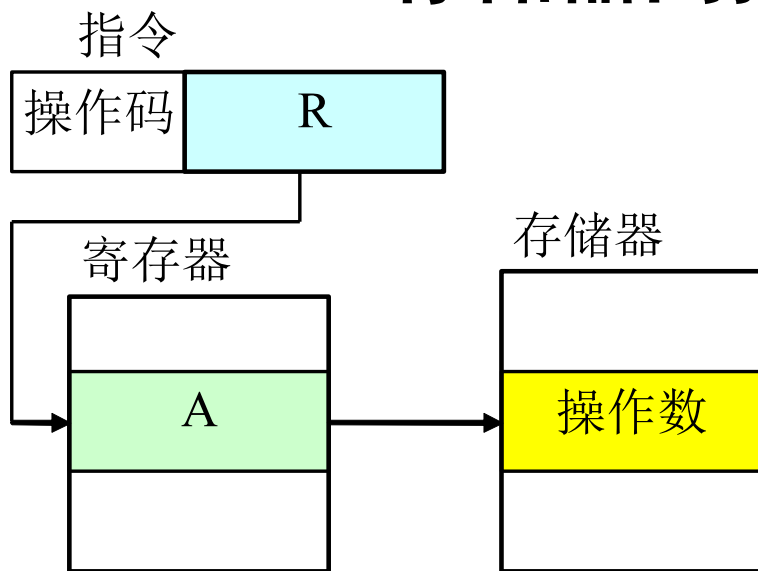


图 寄存器间接寻址

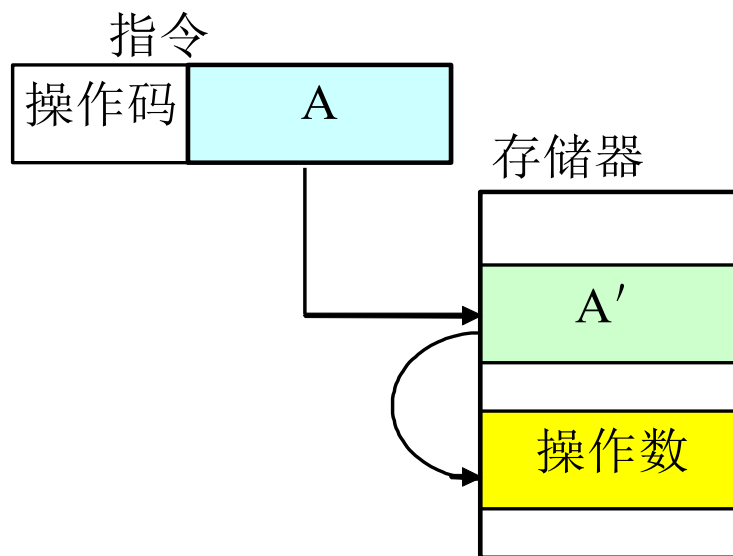


图 存储器间接寻址

操作数寻址方式（续）

6. 相对寻址

相对于程序计数器位移

如INC 8(PC)

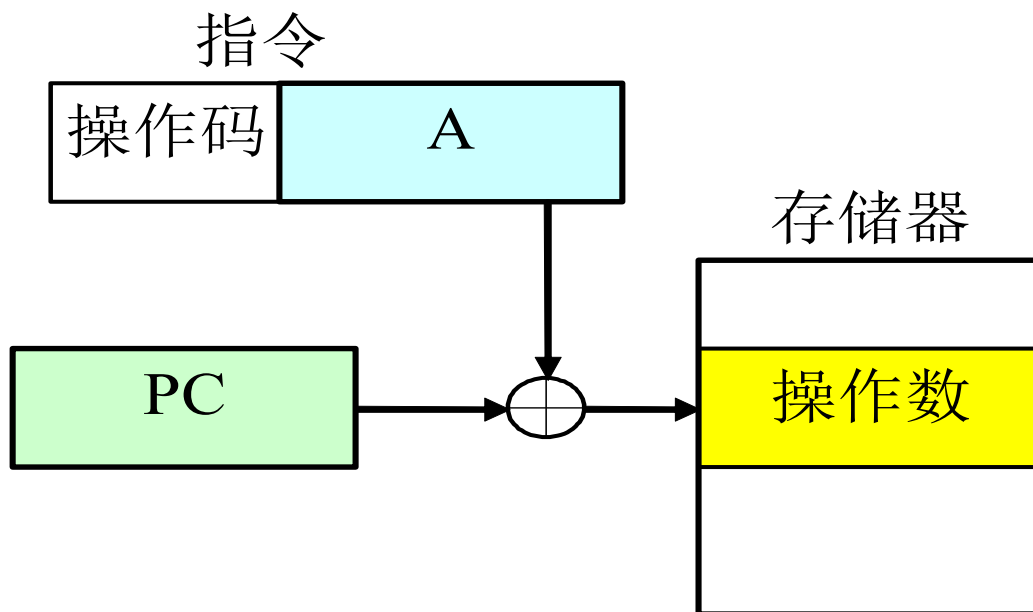


图 相对寻址

操作数寻址方式（续）

7. 变址和基址寻址

如INC (R1) (8)

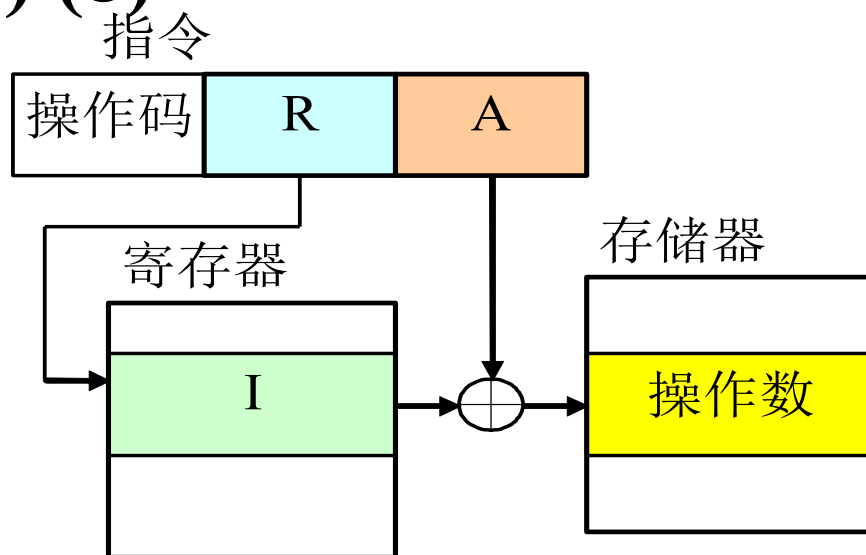


图 变址寻址

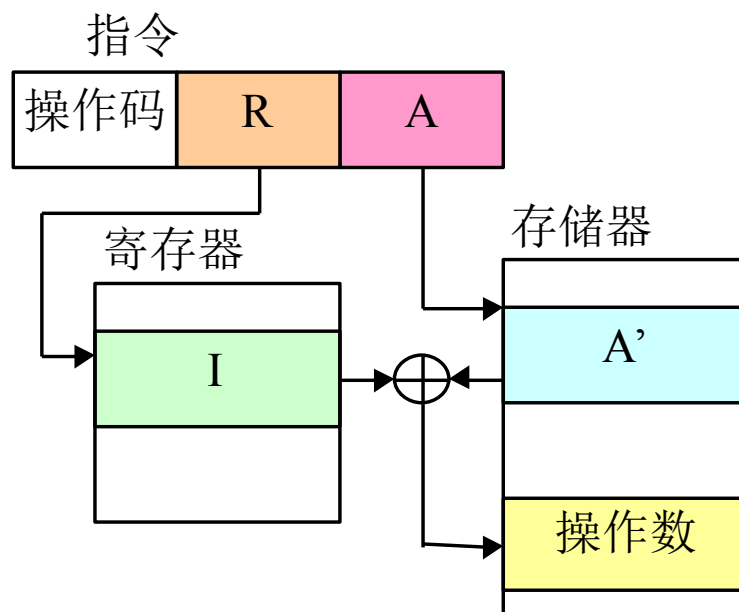
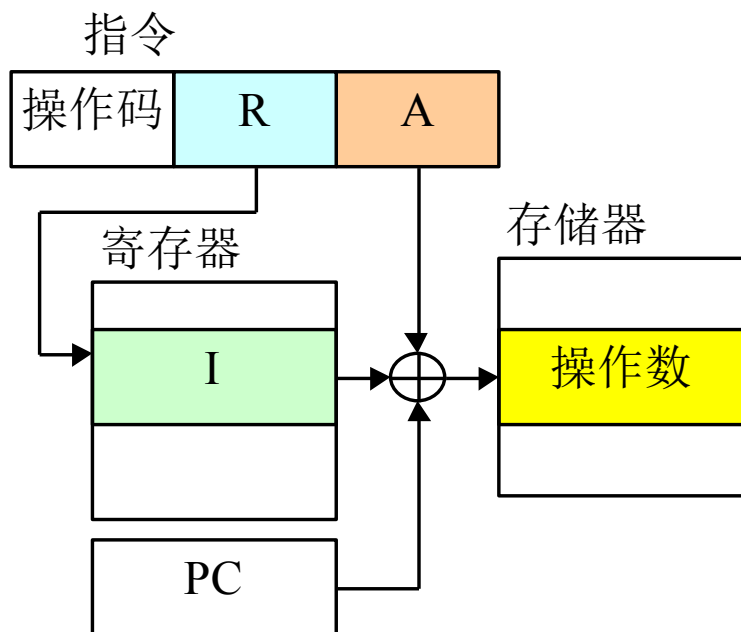
变址寻址：便于数组访问（变址寄存器可存放数组首地址）

基址寻址：可扩大寻址范围，可实现程序浮动（两次变换）

操作数寻址方式（续）

8. 复合寻址

如 $\text{INC } 8(\text{PC}+\text{R1})$ 、 $\text{INC } (\text{R1})(1000)$



操作数寻址方式格式

←: 赋值操作

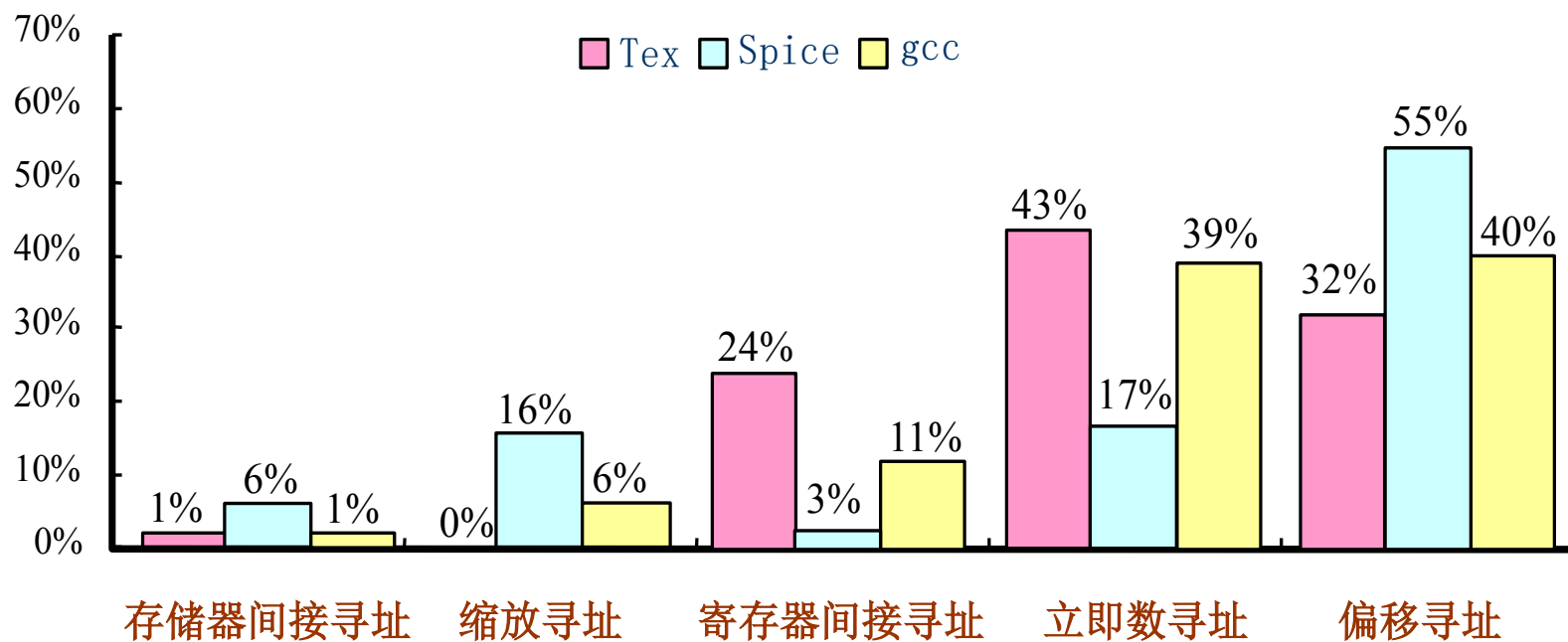
- Mem: 存储器
- Regs: 寄存器组
- 方括号: 表示内容
 - Mem[]: 存储器的内容
 - Regs[]: 寄存器的内容
 - Mem[Regs[R1]]: 以寄存器R1中的内容作为地址的存储器单元中的内容

寻址方式	指令实例	含 义
寄存器寻址	ADD R1 , R2	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R1}] + \text{Regs}[\text{R2}]$
立即值寻址	ADD R3 , #6	$\text{Regs}[\text{R3}] \leftarrow \text{Regs}[\text{R3}] + 6$
偏移寻址	ADD R3 , 120(R2)	$\text{Regs}[\text{R3}] \leftarrow \text{Regs}[\text{R3}] + \text{Mem}[120 + \text{Regs}[\text{R2}]]$
寄存器间接寻址	ADD R4 , (R2)	$\text{Regs}[\text{R4}] \leftarrow \text{Regs}[\text{R4}] + \text{Mem}[\text{Regs}[\text{R2}]]$
索引寻址	ADD R4 , (R2 + R3)	$\text{Regs}[\text{R4}] \leftarrow \text{Regs}[\text{R4}] + \text{Mem}[\text{Regs}[\text{R2}] + \text{Regs}[\text{R3}]]$
直接寻址或绝对寻址	ADD R4 , (1010)	$\text{Regs}[\text{R4}] \leftarrow \text{Regs}[\text{R4}] + \text{Mem}[1010]$
存储器间接寻址	ADD R2 , @(R4)	$\text{Regs}[\text{R2}] \leftarrow \text{Regs}[\text{R2}] + \text{Mem}[\text{Mem}[\text{Regs}[\text{R4}]]]$
自增寻址	ADD R1 , (R2)+	$\begin{aligned} \text{Regs}[\text{R1}] &\leftarrow \text{Regs}[\text{R1}] + \text{Mem}[\text{Regs}[\text{R2}]] \\ \text{Regs}[\text{R2}] &\leftarrow \text{Regs}[\text{R2}] + d \end{aligned}$
自减寻址	ADD R1, -(R2)	$\begin{aligned} \text{Regs}[\text{R2}] &\leftarrow \text{Regs}[\text{R2}] - d \\ \text{Regs}[\text{R1}] &\leftarrow \text{Regs}[\text{R1}] + \text{Mem}[\text{Regs}[\text{R2}]] \end{aligned}$
缩放寻址	ADD R1 , 80(R2)[R3]	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R1}] + \text{Mem}[80 + \text{Regs}[\text{R2}] + \text{Regs}[\text{R3}] * d]$

操作数寻址方式

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

操作数寻址方式使用情况统计



立即数寻址方式和偏移寻址方式的使用频度最高。

操作数寻址方式

两种表示寻址方式的方法

- 将寻址方式编码于操作码中，由操作码描述相应操作的寻址方式。

适合：处理机采用load-store结构，寻址方式只有很少几种。

- 在指令字中设置专门的寻址字段，用以直接指出寻址方式。
 - 灵活，操作码短，但需要设置专门的寻址方式字段，而且操作码和寻址方式字段合起来所需要的总位数可能会比隐含方法的总位数多。

适合：处理机具有多种寻址方式，且指令有多个操作数。

2.3 指令格式的优化设计

- 指令格式优化主要目标和研究内容
- 指令的组成
- 操作码的优化
- 指令字格式的优化
- 指令格式设计举例

1) 指令格式优化的主要目标和研究内容

- **指令格式的优化设计**：是指如何用最短的二进制位数来表示指令的操作信息和地址信息，使指令的平均字长最短。
- **主要目标**
 - 节省程序的存储空间
 - 指令格式尽量规整，减少硬件译码的难度
- **研究内容**
 - 操作码的优化表示
 - 地址码的优化表示

2) 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**
- **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
 - **操作数的数据类型**：
 - 定点数、浮点数、复数、字符、字符串、逻辑数、向量等
 - 若采用自定义数据表示法，则操作码不必指出操作数的数据类型
- **地址码**通常包括三部分内容：
 - **操作数的地址**：直接地址、间接地址、立即数、寄存器编号、变址寄存器编号
 - **地址的附加信息**：如偏移量、块长度、跳距等
 - **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

3) 操作码的优化表示

操作码的优化编码方法通常有三种：

- ① 定长操作码；
- ② Huffman编码；
- ③ 扩展编码。

① 定长操作码

- 所有指令的操作码长度都是相等的。
- 规整简单，但浪费空间。

如果操作码有 n 个

那么操作码的位数至少应该有

$$l = \lceil \log_2 n \rceil \text{位}$$

- 例：已知 操作码的个数 $n = 15$ ，求定长编码的最小平均码长。
- 解：

$$l = \lceil \log_2 15 \rceil = 4$$

② Huffman编码

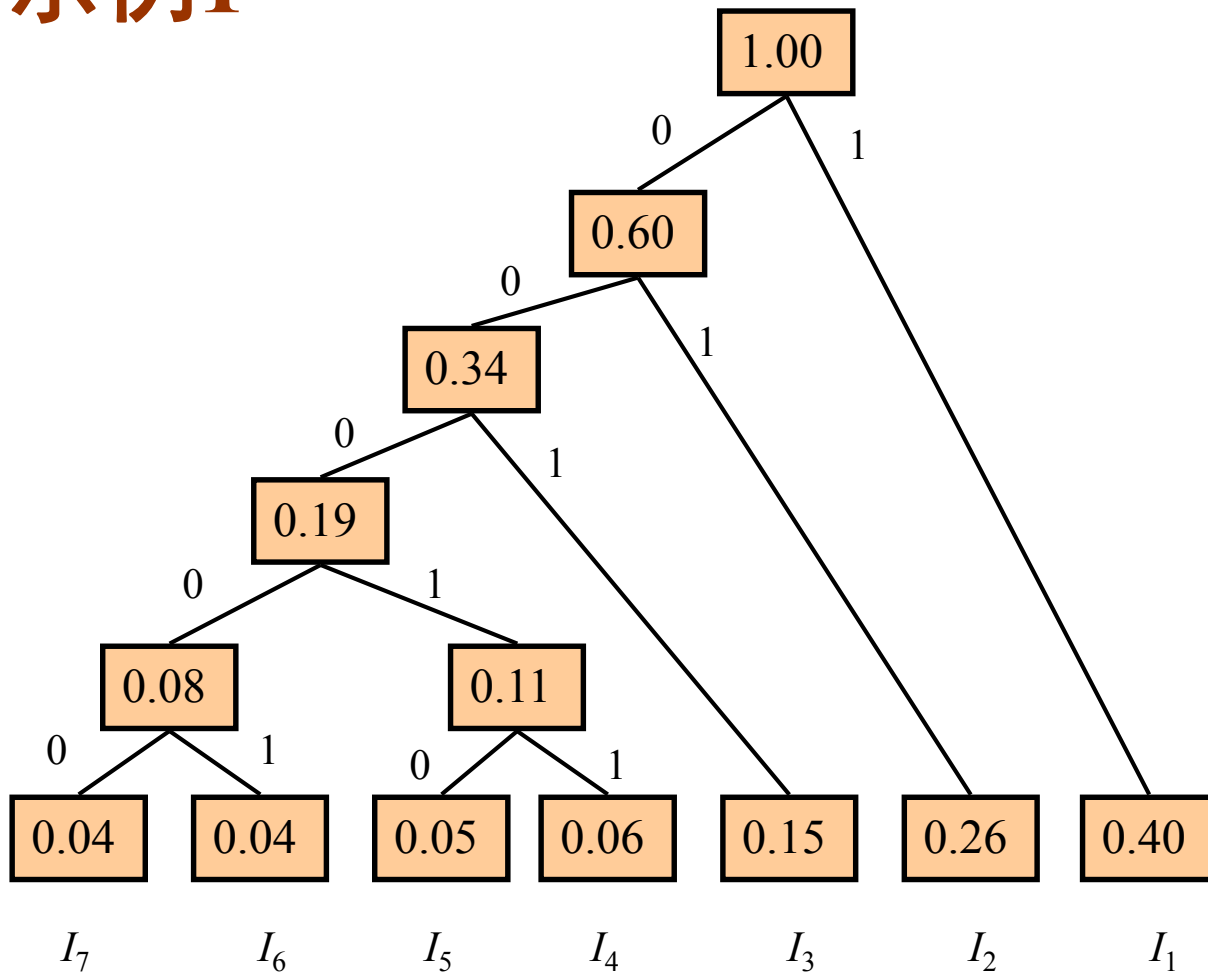
- 对使用频度最高的对象，用最短位数的编码表示，对使用最低的对象，用最长位数的编码表示。
- 构造方法：使用哈夫曼树来构造

利用Huffman树进行操作码编码

- 利用Huffman树进行操作码编码的方法，又称为最小概率合并法。
 - 将各事件按其使用频度从小到大依次排列；
 - 每次从中选择两个频度值最小的结点，将其合并成一个新的结点，并把新结点画在所选结点的上面，
 - 然后用两条边把新结点分别与那两个结点相连。
 - 新结点的频度值是所选两个结点的频度值的和。
 - 把新结点与其他剩余未结合的结点一起，再以上面的步骤进行处理，反复进行，直到全部结点都结合完毕、形成根结点为止。

Huffman编码示例1

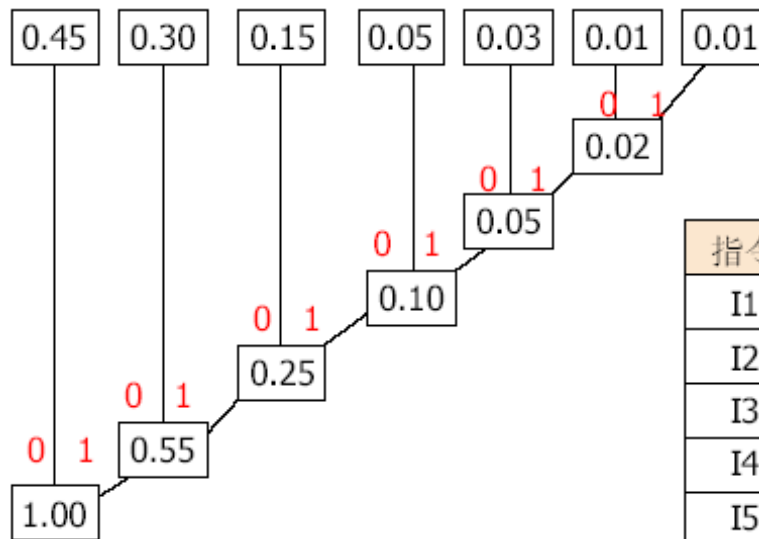
指令	频率	Huffman 编码	长度
I_1	0.40	1	1
I_2	0.26	01	2
I_3	0.15	001	3
I_4	0.06	00011	5
I_5	0.05	00010	5
I_6	0.04	00001	5
I_7	0.04	00000	5



Huffman编码示例2

- 计算机具有7种操作码。固定长编码需要3位二进制。如何采用 Huffman编码。

指令	概率
I1	0.45
I2	0.30
I3	0.15
I4	0.05
I5	0.03
I6	0.01
I7	0.01



指令	概率	Huffman编码	编码长度
I1	0.45	0	1
I2	0.30	10	2
I3	0.15	110	3
I4	0.05	1110	4
I5	0.03	11110	5
I6	0.01	111110	6
I7	0.01	111111	6



注：

■ 哈夫曼树构造过程可以总结为：

从小到大排序，
最小两个合并，
重复上述过程，
只剩一个结束。

■ 哈夫曼树不是唯一的(因为相同的频率可以任取一个在前，且编码时又可任取左1或左0)，但所得的平均码长应是一样的。

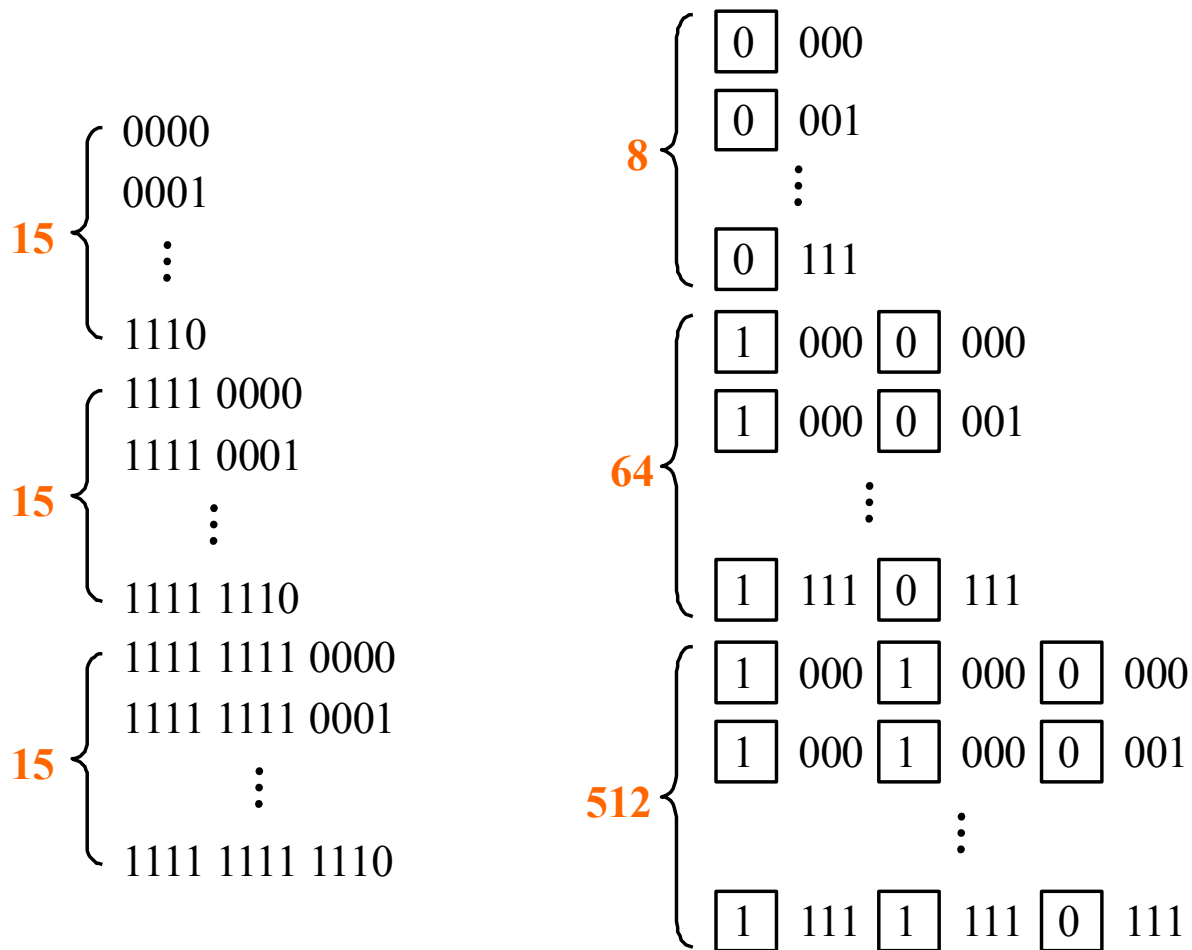
③ 扩展编码

- Huffman编码有优点，但缺点也比较明显：
 - 优点：平均信息长度短，冗余量小。
 - 缺点：操作码不规整，码长种类较多，不利于硬件的译码，也不利于软件的编译，另外，也很难与地址码配合形成长度规整的指令编码。
- 改进：扩展编码法
 - 由定长操作码与Huffman编码法相结合形成的折中方案
 - 主要思想：限定几种码长，使用频度高的用短码，使用频度低的用长码，短码不能是长码的前缀。

扩展编码的表示方法

- **用码长表示：**用横线隔开不同码长，例如4-8-12法。
- **用码点数表示：**例如15/15/15法，8/64/512法
 - 15/15/15法，每一种码长都有4位可编码位（前头可以有相同的扩展标识前缀），可产生16个码点（即编码组合），但是至多只能使用其中15个来表示事件，留下1个或多个码点组合作为更长代码的扩展标识前缀。已经用来表示事件的码点组合不能再作为其它更长代码的前导部分，否则接收者会混淆。这就是“**非前缀原则**”。
 - 8/64/512法，每一种码长按4位分段，每一段中至少要留下1位或多位作为扩展标识。各段剩下的可编码位一起编码，所产生的码点用来对应被编码事件。每一段中的标识位指出后面还有没有后续段。

扩展编码的表示方法



15/15/15 编码法

8/64/512 编码法

扩展编码的分类

- **等长编码法**是对出现频率较低的操作码用较长的编码表示时，每次扩展的编码位数相等，如2-4-6扩展法是指每次扩展时加长2位。
- **不等长编码法**是指每次扩展的编码位数不相等，如1-2-3-5扩展法的前两次扩展都加长1位，第3次扩展加长2位。

示例：扩展编码

I_i	P_i	1 - 2 - 3 - 5 编 码	2-4编码 (3/4)	2-4编码 (2/8)
I1	0.45	0	00	00
I2	0.30	10	01	01
I3	0.15	110	10	1000
I4	0.05	11100	1100	1001
I5	0.03	11101	1101	1010
I6	0.01	11110	1110	1011
I7	0.01	11111	1111	1100

编码方法性能指标

■ 平均码长 l

$$l = \sum_{i=1}^n (P_i \cdot l_i)$$

P_i 是操作码 i 的使用概率

l_i 是操作码 i 的长度

n 为操作码个数

■ 信息熵 H (Entropy) : 表示用二进制编码表示 n 个码点时, 理论上的最短平均码长。任何实际编码得到的平均码长 l 都大于 H 。

$$H = - \sum_{i=1}^n [P_i \cdot \log_2(P_i)]$$

■ 信息冗余量: 表明消息编码中“无用成分”所占的百分比, 用来衡量代码优化的程度。

$$R = \frac{l - H}{l} = \left(1 - \frac{H}{l}\right)$$

例：定长编码的信息冗余量

- 定长编码的平均码长为

$$l = \lceil \log_2 n \rceil$$

- 定长编码的信息冗余量为

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

例：编码方法性能比较

- 某计算机有7种不同的操作码，其使用概率如图所示，试比较定长编码、Huffman编码、1-2-3-5不等长扩展编码、2-4(3/4)等长扩展编码、2-4(2/8)等长扩展编码的熵以及平均码长、信息冗余量。

指令	概率
I1	0.45
I2	0.30
I3	0.15
I4	0.05
I5	0.03
I6	0.01
I7	0.01

解：

■ 熵H=

—

$$(2 \times 0.01 \times \log_2 0.01 + 0.03 \times \log_2 0.03 + 0.05 \times \log_2 0.05 + 0.15 \times \log_2 0.15 + 0.3 \times \log_2 0.3 + 0.45 \times \log_2 0.45) \\ \approx 1.95$$

解（续）：

- 定长编码的平均码长

$$l = \lceil \log_2 7 \rceil = 3$$

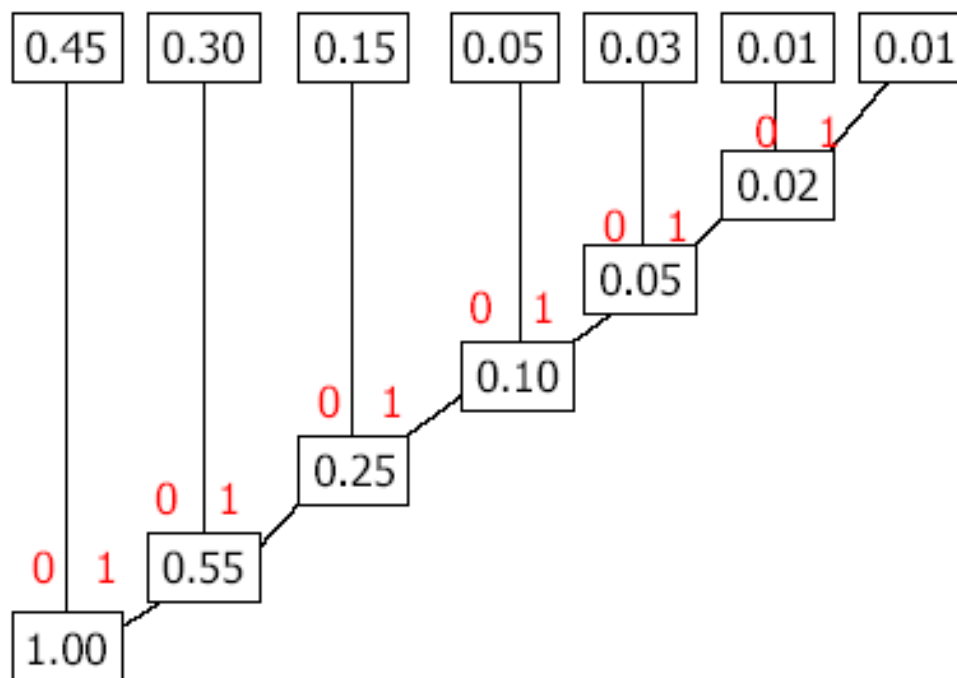
- 定长编码的信息冗余量

$$R = \frac{l - H}{l} = 1 - \frac{H}{l} = 1 - \frac{1.95}{3} \approx 35\%$$

解（续）：

■ Huffman编码

指令	概率
I1	0.45
I2	0.30
I3	0.15
I4	0.05
I5	0.03
I6	0.01
I7	0.01



解（续）：

■ Huffman编码的平均码长

$$l = \sum_{i=1}^n (P_i \cdot l_i) =$$

$$0.45 \times 1 + 0.30 \times 2 + 0.15 \times 3 + 0.05 \times 4 + 0.03 \times 5 + 0.01 \times 6 + 0.01 \times 6 \\ = 1.97$$

■ Huffman编码的信息冗余量

$$R = \frac{l - H}{l} = \left(1 - \frac{H}{l}\right) = \left(1 - \frac{1.95}{1.97}\right) = 1.0\%$$

■ 可见，Huffman编码明显优于定长编码

指令	概率	Huffman编码	编码长度
I1	0.45	0	1
I2	0.30	10	2
I3	0.15	110	3
I4	0.05	1110	4
I5	0.03	11110	5
I6	0.01	111110	6
I7	0.01	111111	6

解（续）：

■ 扩展编码

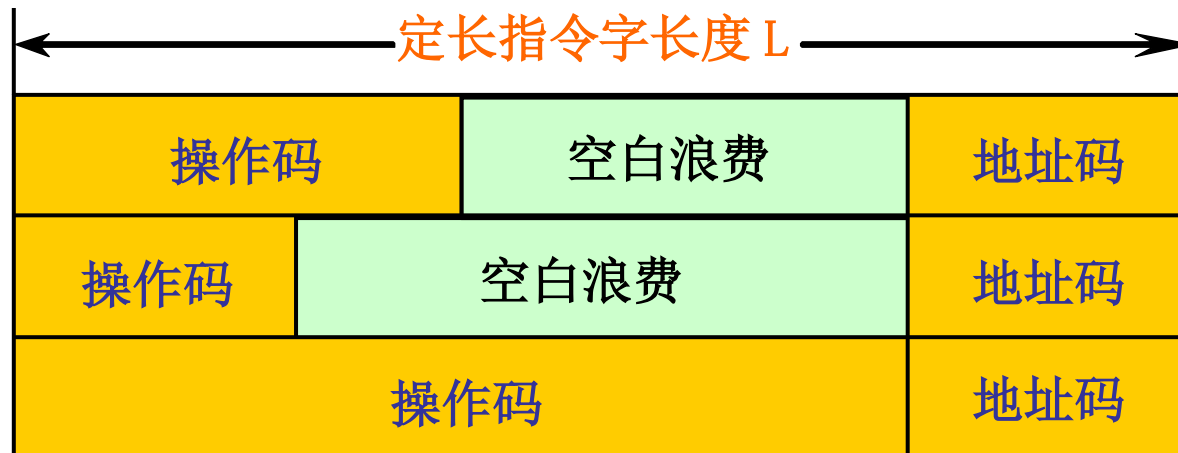
I_i	P_i	1-2-3-5编码	2-4编码（3/4）	2-4编码（2/8）
I1	0.45	0	00	00
I2	0.30	10	01	01
I3	0.15	110	10	1000
I4	0.05	11100	1100	1001
I5	0.03	11101	1101	1010
I6	0.01	11110	1110	1011
I7	0.01	11111	1111	1100

解（续）：

- 采用1-2-3-5不等长扩展编码的操作码平均长度为：
 - $l=(0.45 \times 1+0.30 \times 2+0.15 \times 3+(0.05+0.03+0.01+0.01) \times 5=2.00$ 位
 - 信息冗余量为： $R=1-1.95/2.00=2.5\%$
- 采用2-4(3/4)等长扩展编码的操作码平均长度为：
 - $l=(0.45+0.30+0.15) \times 2+(0.05+0.03+0.01+0.01) \times 4=2.20$ 位
 - 信息冗余量为： $R=1-1.95/2.20=11.4\%$
- 采用2-4(2/8)等长扩展编码的操作码平均长度为：
 - $l=(0.45+0.30) \times 2+(0.15+0.05+0.03+0.01+0.01) \times 4=2.50$ 位
 - 信息冗余量为： $R=1-1.95/2.50=22\%$
- 由上可知，1-2-3-5不等长扩展编码优于后两种等长扩展编码。

4) 指令字格式的优化

- 前面我们学习了操作码的优化，但是一条指令码还包括地址码。两者合理安排才能使指令格式得到优化。
- 如果指令字的宽度固定，地址码的长度和个数固定，则操作码的缩短并不能带来好处，只是使指令字中出现空白浪费。



4) 指令字格式的优化

■ 采用地址个数可变和/或地址码长度可变的方案

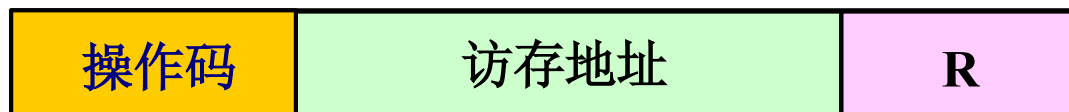
- 利用操作码缩短所带来的好处
- 最常用的操作码最短，其地址字段个数最多。

能够使指令的功能增强，从总体上减少所需指令条数。

寄存器—寄存器型



寄存器—存储器型



带立即操作数



4) 指令字格式的优化

■ 考虑因素

- 计算机中寄存器的个数和寻址方式的数目对机器的指令字长有很大的影响；
- 指令字的平均长度增加了，程序的平均长度也就增加了；
- 在指令系统的设计中，要在指令字长与寄存器的个数以及寻址方式的个数之间进行折中。

■ 指令系统的3种编码格式

可变长度编码格式、固定长度编码格式、混合型编码格式

4) 指令字格式的优化

■ 可变长度编码格式

- 当指令系统的寻址方式和操作种类很多时，这种编码格式是最好的。
- 用最少的二进制位来表示目标代码。
- 可能会使各条指令的字长和执行时间相差很大。

操作码	地址描述符 1	地址码 1	...	地址描述符 n	地址码 n
-----	---------	-------	-----	---------	-------

4) 指令字格式的优化

■ 固定长度编码格式

- 将操作类型和寻址方式一起编码到操作码中。
- 当寻址方式和操作类型非常少时，这种编码格式非常好。
- 可以有效降低译码的复杂度，提高译码的速度。
- 大部分RISC的指令系统均采用这种编码格式。

操作码	地址码 1	地址码 2	地址码 3
-----	-------	-------	-------

4) 指令字格式的优化

■ 混合型编码格式

- 提供若干种固定的指令字长。
- 以期达到既能够减少目标代码长度又能降低译码复杂度的目标。

操作码	地址描述符	地址码
-----	-------	-----

操作码	地址描述符 1	地址描述符 2	地址码
-----	---------	---------	-----

操作码	地址描述符	地址码 1	地址码 2
-----	-------	-------	-------

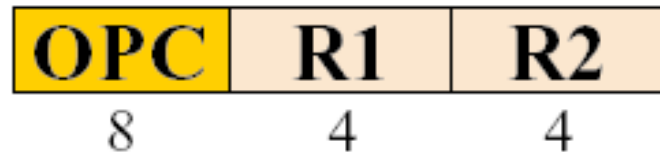
5) 指令格式设计举例

- IBM370指令设计
- PDP-11指令设计
- MIPS指令设计

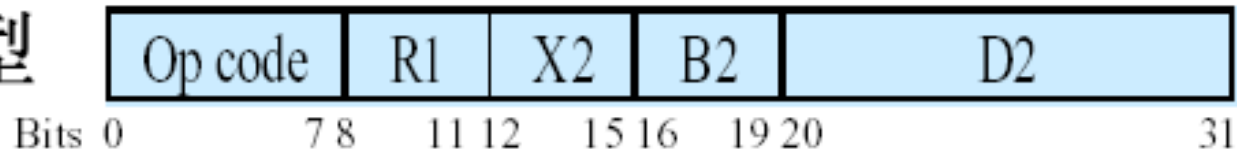
IBM370指令设计

指令不定长
操作码定长
操作数采用两地址
R: 寄存器
S: 存储器
X: 变址
I: 立即数

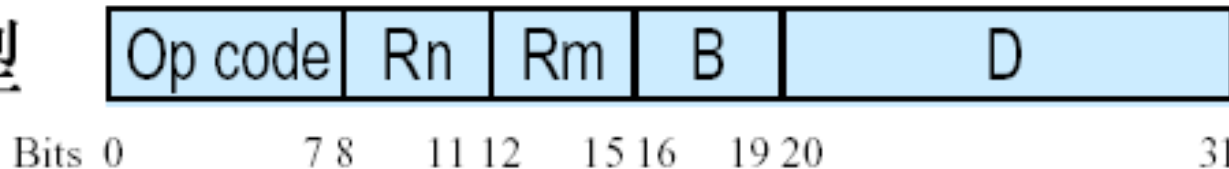
● RR型



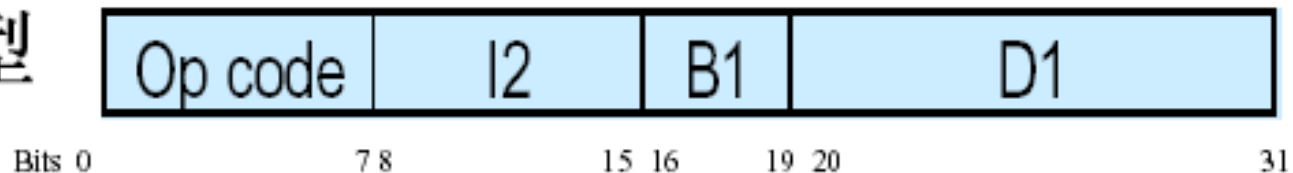
● RX型



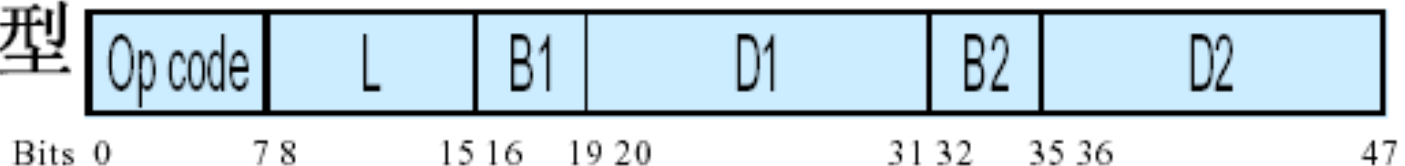
● RS型



● SI型



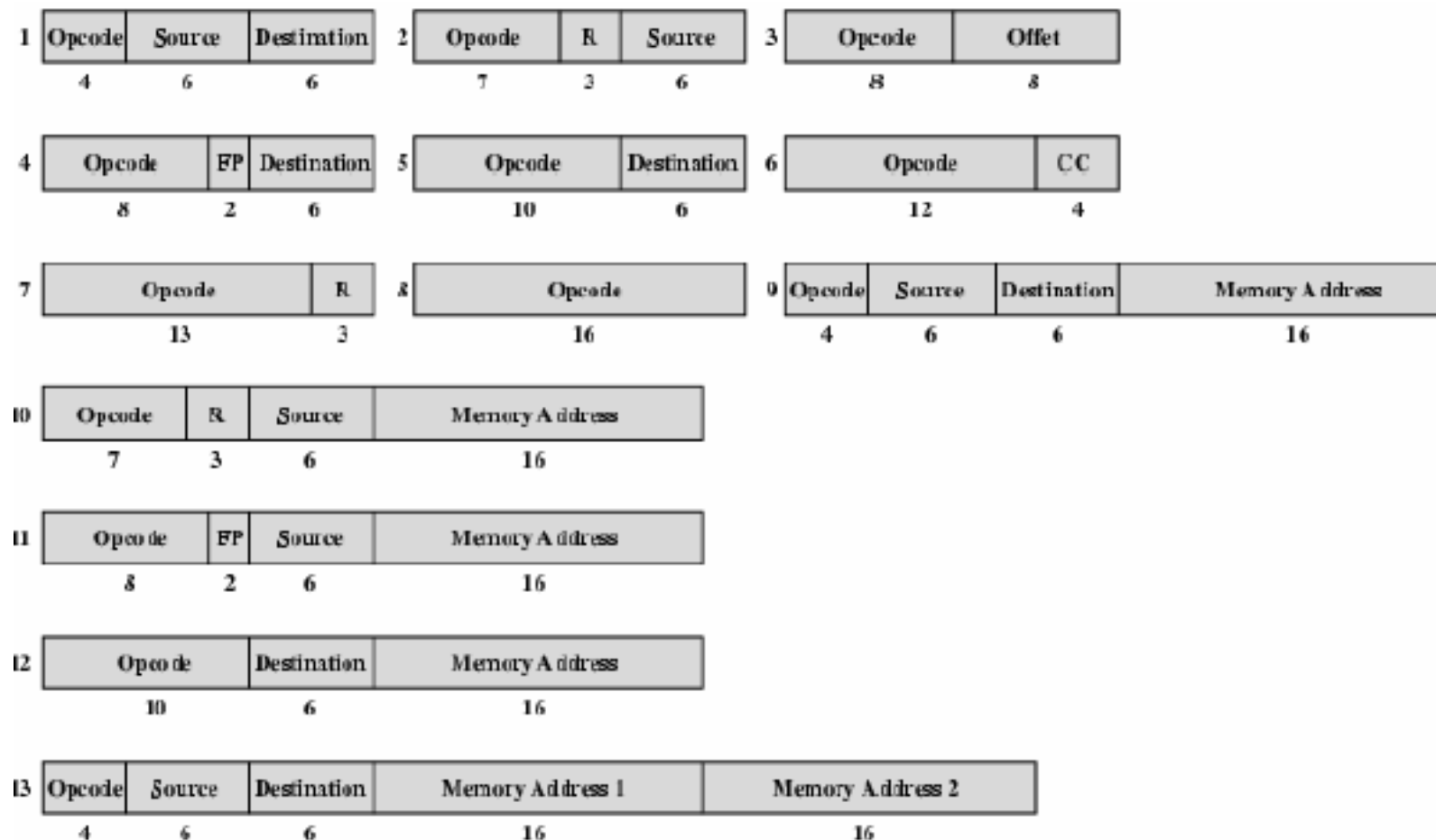
● SS型



IBM370指令组成

- 指令长度：不等长，有16、32、48位
- 操作码部分长度：等长，均为8位
- 地址码部分长度：8、24、40位
- 地址个数：均为二地址；寻址方式：多种
 - RR 寄存器—寄存器寻址
 - RX 寄存器—变址寻址
 - RS 寄存器—主存直接寻址，可将 R_n 到 R_m 的数据与主存单元中的数据相互传输
 - SI 主存直接—立即数寻址
 - SS 主存直接—主存直接寻址

PDP-11指令设计



Numbers below fields indicate bit length

Source and Destination each contain a 3-bit addressing mode field and a 3-bit register number

FP indicates one of four floating-point registers

R indicates one of the general-purpose registers

CC is the condition code field

PDP-11指令组成

- 指令长度：不等长（单字、双字、三字）
- 操作码部分长度：4位开始，分别扩展到4,7,8,10,12,13,16位
- 地址个数：零地址、一地址、二地址。
- 寻址方式：8种
- 不同长度操作码配以不同个数、不同寻址方式、不同长度的地址码。

MIPS指令系统结构

MIPS的寄存器:

- 32个64位通用寄存器（GPRs）
 - R0, R1, ..., R31
 - 也称为整数寄存器
 - R0的值永远是0
- 32个64位浮点数寄存器（FPRs）
 - F0, F1, ..., F31

MIPS指令系统结构

- 用来存放32个单精度浮点数（32位），也可以用来存放32个双精度浮点数（64位）。
- 存储单精度浮点数（32位）时，只用到FPR的一半，其另一半没用。

■ 一些特殊寄存器

- 它们可以与通用寄存器交换数据。
- 例如浮点状态寄存器：用来保存有关浮点操作结果的信息。

MIPS指令系统结构

■ MIPS的数据表示

■ 整数

字节（8位） 半字（16位）

字（32位） 双字（64位）

■ 浮点数

单精度浮点数（32位） 双精度浮点数（64位）

- 字节、半字或者字在装入64位寄存器时，用零扩展或者用符号位扩展来填充该寄存器的剩余部分。装入以后，对它们将按照64位整数的方式进行运算。

MIPS指令系统结构——数据寻址方式

- 立即数寻址与偏移量寻址；

立即数字段和偏移量字段都是16位的。

- 寄存器间接寻址是通过把0作为偏移量来实现的；

- 16位绝对寻址是通过把R0（其值永远为0）作为基址

寄存器来完成的；

- 4. MIPS的存储器是按字节寻址的，地址为64位；

- 5. 所有存储器访问都必须是边界对齐的。

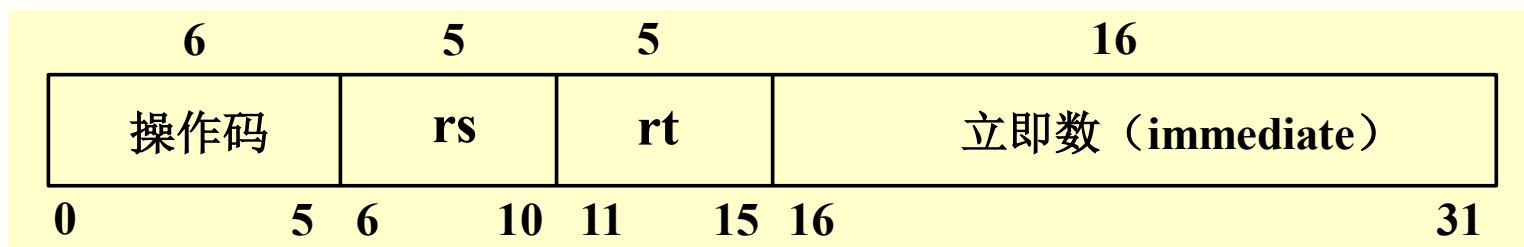
MIPS指令系统结构—指令格式

- 寻址方式编码到操作码中
- 所有的指令都是32位的
- 操作码占6位
- 3种指令格式
 - 3种格式中，同名字段的位置固定不变。

MIPS指令系统结构

■ I类指令

- 包括所有的load和store指令，立即数指令，分支指令，寄存器跳转指令，寄存器链接跳转指令。
- 立即数字段为16位，用于提供立即数或偏移量。



■ load指令

访存有效地址: $\text{Regs}[\text{rs}] + \text{immediate}$

从存储器取来的数据放入寄存器rt

■ store指令

访存有效地址: $\text{Regs}[\text{rs}] + \text{immediate}$

要存入存储器的数据放在寄存器rt中

■ 立即数指令

$\text{Regs}[\text{rt}] \leftarrow \text{Regs}[\text{rs}] \text{ op } \text{immediate}$

■ 分支指令

转移目标地址: $\text{Regs}[\text{rs}] + \text{immediate}$, rt无用

■ 寄存器跳转、寄存器跳转并链接

转移目标地址为 $\text{Regs}[\text{rs}]$

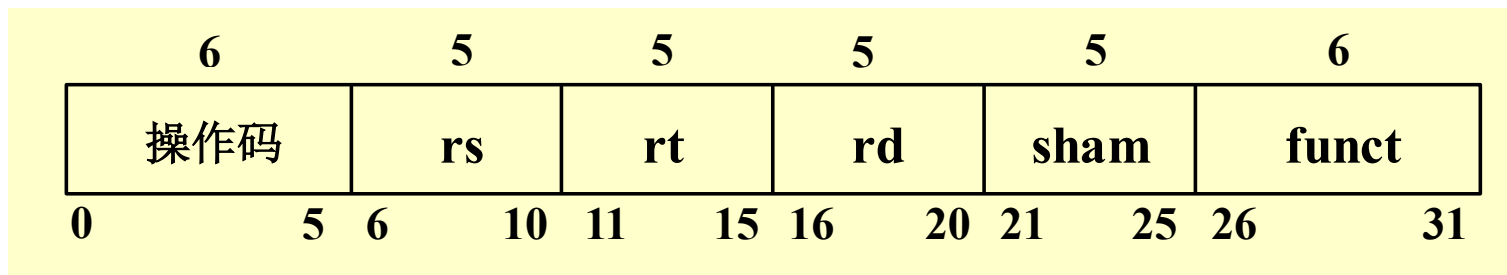
■ R类指令

■ 包括ALU指令，专用寄存器读/写指令，move指令等。

■ ALU指令

$\text{Regs}[\text{rd}] \leftarrow \text{Regs}[\text{rs}] \text{ funct } \text{Regs}[\text{rt}]$

funct为具体的运算操作编码



■ J类指令

- 包括跳转指令，跳转并链接指令，自陷指令，异常返回指令。
- 在这类指令中，指令字的低26位是偏移量，它与PC值相加形成跳转的地址。



MIPS指令系统结构——操作

■ MIPS指令可以分为四大类

- load和store
- ALU操作
- 分支与跳转
- 浮点操作

■ 符号的意义

- $x \leftarrow_n y$: 从y传送n位到x
- $x, y \leftarrow z$: 把z传送到x和y

■ 下标：表示字段中具体的位；

- 对于指令和数据，按从最高位到最低位（即从左到右）的顺序依次进行编号，最高位为第0位，次高位为第1位，依此类推。
- 下标可以是一个数字，也可以是一个范围。

例如：Regs[R4]₀：寄存器R4的符号位

Regs[R4]₅₆₋₆₃：R4的最低字节

■ Mem：表示主存；

- 按字节寻址，可以传输任意个字节。

■ 上标：用于表示对字段进行复制的次数。

例如：0³²：一个32位长的全0字段

MIPS指令系统结构

- **符号##**：用于两个字段的拼接，并且可以出现在数据传送的任何一边。

举例： R8、R10： 64位的寄存器， 则

$$\text{Regs}[\text{R8}]_{32-63} \leftarrow_{32} (\text{Mem} [\text{Regs}[\text{R6}]]_0)^{24} \quad \text{## Mem}[\text{Regs}[\text{R6}]]$$

表示的意义是：

以R6的内容作为地址访问内存，得到的字节按符号位扩展为32位后存入R8的低32位，R8的高32位（即 $\text{Regs}[\text{R8}]_{0-31}$ ）不变。

load和store指令

指令举例	指令名称	含义
LD R2, 20(R3)	装入双字	$\text{Regs}[R2] \leftarrow_{64} \text{Mem}[20+\text{Regs}[R3]]$
LW R2, 40(R3)	装入字	$\text{Regs}[R2] \leftarrow_{64} (\text{Mem}[40+\text{Regs}[R3]])_0^{32} \text{ ## } \text{Mem}[40+\text{Regs}[R3]]$
LB R2, 30(R3)	装入字节	$\text{Regs}[R2] \leftarrow_{64} (\text{Mem}[30+\text{Regs}[R3]])_0^{56} \text{ ## } \text{Mem}[30+\text{Regs}[R3]]$
LBU R2, 40(R3)	装入无符号字节	$\text{Regs}[R2] \leftarrow_{64} 0^{56} \text{ ## } \text{Mem}[40+\text{Regs}[R3]]$
LH R2, 30(R3)	装入半字	$\text{Regs}[R2] \leftarrow_{64} (\text{Mem}[30+\text{Regs}[R3]])_0^{48} \text{ ## } \text{Mem}[30+\text{Regs}[R3]] \text{ ## } \text{Mem}[31+\text{Regs}[R3]]$
L.S F2, 60(R4)	装入半字	$\text{Regs}[F2] \leftarrow_{64} \text{Mem}[60+\text{Regs}[R4]] \text{ ## } 0^{32}$
L.D F2, 40(R3)	装入双精度浮点数	$\text{Regs}[F2] \leftarrow_{64} \text{Mem}[40+\text{Regs}[R3]]$
SD R4, 300(R5)	保存双字	$\text{Mem}[300+\text{Regs}[R5]] \leftarrow_{64} \text{Regs}[R4]$
SW R4, 300(R5)	保存字	$\text{Mem}[300+\text{Regs}[R5]] \leftarrow_{32} \text{Regs}[R4]$
S.S F2, 40(R2)	保存单精度浮点数	$\text{Mem}[40+\text{Regs}[R2]] \leftarrow_{32} \text{Regs}[F2]_{0..31}$
SH R5, 502(R4)	保存半字	$\text{Mem}[502+\text{Regs}[R4]] \leftarrow_{16} \text{Regs}[R5]_{48..63}$

ALU指令

- 寄存器—寄存器型（RR型）指令或立即数型
- 算术和逻辑操作：加、减、与、或、异或和移位等

指令举例	指令名称	含义
DADDU R1, R2, R3	无符号加	$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + \text{Regs}[R3]$
DADDIU R4, R5, #6	加无符号立即数	$\text{Regs}[R4] \leftarrow \text{Regs}[R5] + 6$
LUI R1, #4	把立即数装入到一个字的高16位	$\text{Regs}[R1] \leftarrow 0^{32} \text{ ## } 4 \text{ ## } 0^{16}$
DSLL R1, R2, #5	逻辑左移	$\text{Regs}[R1] \leftarrow \text{Regs}[R2] \ll 5$
DSLT R1, R2, R3	置小于	$\text{If}(\text{Regs}[R2] < \text{Regs}[R3])$ $\text{Regs}[R1] \leftarrow 1 \text{ else } \text{Regs}[R1] \leftarrow 0$

MIPS指令系统结构——控制指令

- 由一组跳转和一组分支指令来实现控制流的改变
- 典型的MIPS控制指令

指令举例	指令名称	含义
J name	跳转	$PC_{36..63} \leftarrow name \ll 2$
JAL name	跳转并链接	$Regs[R31] \leftarrow PC+4$; $PC_{36..63} \leftarrow name \ll 2$; $((PC+4) - 2^{27}) \leq name < ((PC+4) + 2^{27})$
JALR R3	寄存器跳转并链接	$Regs[R31] \leftarrow PC+4$; $PC \leftarrow Regs[R3]$
JR R5	寄存器跳转	$PC \leftarrow Regs[R5]$
BEQZ R4, name	等于零时分支	if ($Regs[R4] == 0$) $PC \leftarrow name$; $((PC+4) - 2^{17}) \leq name < ((PC+4) + 2^{17})$
BNE R3, R4, name	不相等时分支	if ($Regs[R3] \neq Regs[R4]$) $PC \leftarrow name$ $((PC+4) - 2^{17}) \leq name < ((PC+4) + 2^{17})$
MOVZ R1, R2, R3	等于零时移动	if ($Regs[R3] == 0$) $Regs[R1] \leftarrow Regs[R2]$

2.4 指令系统的功能设计

- 指令系统性能
- 指令的种类

指令系统性能

设计指令系统时，在功能方面的最基本要求是：

- **完整性(Completeness)** 是指应该具备的基本指令种类，能处理机器所具有的各种数据表示。
- **规整性(Consistency)** 指各寄存器、内存单元在指令系统中处于对等地位，便于译码、执行。主要表现在：
 - **对称性**：所有存储设备的使用，操作码的设置等都要对称，如对所有寄存器要同等对待。有 $(R1)op(R2) \rightarrow R1$ ，也应该有 $(R2)op(R1) \rightarrow R1$
 - **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，这一般很难实现，通常规定运算需在寄存器中完成。
- **可扩充性(Extendibility)**：要保留一定余量的操作码空间，以备将来扩展。
- **兼容性**：在同一系列机内指令系统不变（可以适当增加）
- **高效率**：指令的执行速度要快；指令的使用频度要高；各类指令之间要有一定的比例

指令系统的完整性

- 指令的种类：
 - 数据传送
 - 运算
 - 程序控制
 - 输入输出
 - 处理机控制和调试

■ 1、数据传送类指令

由三个主要因素决定：

- 数据存储设备的种类
- 数据单位：字、字节、位、数据块等
- 采用的寻址方式

■ 2、算术和逻辑运算类指令

考虑四个因素的组合：

- (1) 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
- (2) 数据表示：定点、浮点、逻辑、十进制、字符串、定点向量等
- (3) 数据长度：字、双字、半字、字节、位、数据块等
- (4) 数据存储设备：通用寄存器、主存储器、堆栈等

■ 3、程序控制指令

主要包括三类：

■ 转移指令

- 无条件转移：局部无条件转移，采用相对寻址
- 有条件转移：转移条件有：全零(Z)、正负号(N)、进位(C)、溢出(V)及它们的组合等。如BEQ（等于零转移），BCS（有进位转移），BVS（有溢出转移）。

■ 程序调用和返回指令

- CALL 转入子程序
- RETURN 从子程序返回，本身可以带有条件
- 中断控制指令和自陷指令也属此类，如开中断、关中断、改变屏蔽状态、中断返回、自陷等等指令。

■ 循环控制指令

■ 4、输入输出指令

- 主要有：启动设备、停止设备、测试设备、控制设备，数据输入、数据输出等指令
- 多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，必须用系统调用进入操作系统，由操作系统对设备统一进行管理
- 也可以不设置输入输出指令，输入输出设备与主存储器共用同一个零地址空间，访存的指令也能访问输入输出设备

■ 5、处理机控制和调试指令

- **处理机状态切换指令**：在一般的计算机系统中，处理机至少有两个或两个以上状态，即管态和用户态，或称主态和从态。两个状态间需要进行切换。系统的指令也相应分为两种类型：供系统管理员使用的特权指令，以及供普通用户使用的一般指令，管态下可执行特权指令，用户态下只能执行一般指令。

■ 调试指令

- **硬件调试指令**：如钥匙位置读取、开关状态的读取，寄存器和主存单元的显示等；
- **软件调试指令**：如断点的设置、跟踪，自陷井指令等。

2.5 指令系统的发展方向和优化

- CISC和RISC
- 复杂指令系统计算机---CISC
- 精简指令系统计算机---RISC

1) CISC和RISC

■ 计算机指令系统设计存在着两种截然不同的思想：

- 复杂指令系统计算机CISC (Complex Instruction Set Computer)：通过强化指令集功能，并实现软件功能向硬件功能转移来提高计算机的性能（设置复杂的功能更强的指令来代替原先由软件子程序实现的功能）。
- 精简指令系统计算机RISC (Reduced Instruction Set Computer) 只保留功能简单的指令，而功能较复杂的指令则用子程序来实现，通过尽可能降低指令集结构的复杂性，来达到简化指令集的实现和提高性能的目的。

2) 复杂指令系统计算机---CISC

- 按CISC方向发展改进指令系统的基本思想是：**如何进一步增强原有指令的功能，以及设置更为复杂的新指令来取代原先由软件子程序完成的功能，实现软件功能的硬化。**
- 按照CISC方向，主要可以从三个方面来改造指令系统：
 - 面向目标程序实现优化
 - 面向高级程序语言实现优化
 - 面向操作系统实现优化

面向目标程序实现优化

- **目标：普遍缩短包括系统软件和应用软件在内的各种机器语言目标程序的长度，减少程序执行过程中处理机和主存间信息交换的次数，减少目标程序的执行时间。具体途径有：**

- 一、通过统计机器语言目标程序中的各种指令与指令串的**静态使用频度**和程序在执行过程中指令的**动态使用频度**来改造指令，对使用频度高的指令可以增强其功能，加快其执行速度，缩短其指令字长；对使用频度高的指令串可以增设功能更强的新指令或复合指令来替代；对使用频度很低的指令可以将其功能合并到某些频度高的指令中去，或在开发新型号机器时将其取消。
- 二、将常用的宏指令或子程序实现的功能改为用强功能复合指令来实现。
- 三、尽量减少指令集中不执行数据变换的非功能型指令的比例，使真正执行数据变换等运算功能型指令所占的比例增大。

面向高级程序语言实现优化

- **目标：缩短高级语言和机器语言之间的语义差距，支持编译程序，减少编译所需的时间，提高目标程序形成效率。**
具体途径有：

- 一、统计高级语言源程序中各种语句的使用频度，增设与高频度语句的语义差别小的新指令；
- 二、面向编译，优化代码生成，从增强结构的规整性和对称性方面来改进指令系统；
- 三、改进后的指令系统应尽可能与各种高级语言的同一功能语句的保持同等程度的语义差距；
- 四、设计面对各种高级语言优化实现的多种指令系统，并且能够动态地切换，自适应各种高级语言程序的运行。
- 五、发展高级语言计算机。高级语言机器是指不需要编译即可运行高级语言程序的计算机，它通过硬件或固件来直接解释执行高级语言的语句。

面向操作系统实现优化

- **目标：缩短操作系统与计算机系统结构之间的语义差距，减少运行操作系统的时间和占用的存储空间。具体途径有：**
 - 一、对操作系统中常用指令和指令串的使用频度进行统计分析来改进；
 - 二、增设专用于支持操作系统功能的新指令，尤其是一些固定的管理类功能指令；
 - 三、把操作系统中频繁使用且对速度影响大的某些软件子程序进行硬化或固化，或直接改用硬件或微程序解释实现；
 - 四、设置专门的处理机来运行操作系统，发展功能分布的多处理机系统。

3) 精简指令系统计算机---RISC

■ CISC存在的问题：

- 指令系统日趋庞大和复杂，不宜于用VLSI技术实现，使机器的设计周期延长，成本升高，设计错误增多，系统可靠性降低；
- 指令系统中，约有80%的指令的使用频度很低，利用率低，因此，降低了指令系统的性能价格比。
- 功能复杂的指令使指令的译码和操作的复杂，执行速度慢。并使得指令系统的指令平均周期数较大，增大了程序的执行时间，降低了程序的执行速度；
- 复杂的指令系统使得高级语言源程序的优化编译变得困难，时空开销增大，编译效率难以提高，且编译程序复杂。

■ 解决办法---RISC

- RISC的设计思想：只保留功能简单的指令，功能较复杂的指令用子程序（软件）来实现。

设计RISC机器的一般原则

- 1、**只选择使用频度很高的指令**，在此基础上再增加少量能效支持操作系统、高级语言实现及其它功能的指令，让指令条数大大减少；
- 2、**减少指令、寻址方式的种类**，一般不超过两种寻址方式；简化指令的格式，使之也限制在两种之内，并争取让全部指令都具有相同的长度；
- 3、**大多数指令（90%以上）都在一个机器周期内完成**；
- 4、**采用寄存器间运算结构**：扩大通用寄存器的数量，一般不少于32个寄存器，以**尽可能减少访存操作**。所有指令中，只有取（LOAD）/存（STORE）指令才可以访存，其他指令的操作一律都在寄存器间进行。
- 5、**硬联控制为主，固件实现为辅**。所有简单指令直接通过硬件译码和实现，从而提高指令执行速度，只有少数指令采用微程序解释实现；
- 6、**优化设计编译程序**，以简单有效的方式来支持高级语言的实现。

设计RISC机器的一般原则（简单版）

- 1、精简指令条数，保留使用频度高的指令；
- 2、简化指令格式，采用简单寻址方式，绝大多数指令可以在单周期内执行完成；
- 3、采用寄存器间运算结构，减少访存次数；
- 4、指令以硬联组合电路实现为主，少量指令可以用微程序解释方式执行；
- 5、优化编译程序的设计。

减少CPI是RISC思想的精华

- 任何一个程序在计算机上的执行时间可用如下公式表示：

$$T_{CPU} = I_N \cdot CPI \cdot T_C$$

- 其中： T_{CPU} 是执行某个程序所用的CPU时间；
- I_N 是该程序包含的指令条数；
- CPI是每条指令执行的平均时钟周期数；
- T_C 是一个时钟周期的时间长度。

减少CPI是RISC思想的精华（续）

- RISC的速度要比CISC快3~10倍左右，因为：
 - 1. 程序所执行的总指令条数 I_N : RISC的 I_N 长度可能比CISC的长30%至40%。
 - 2. 对于指令平均执行周期数CPI: RISC的CPI多数为1，而CISC的CPI多数为4~6。
 - 3. 对于一个周期的时间长度 T_C : RISC一般采用硬布线逻辑实现，因此，RISC的 T_C 比CISC的 T_C 小。

类型	指令条数 I_N	指令平均周期数CPI	周期时间 T_C
CISC	1	2~15	33ns~5ns
RISC	1.3~1.4	1.1~1.4	10ns~2ns



例如：Intel公司的80x86处理机的CPI在不断缩小

8088的CPI大于20

80286的CPI大约是5.5

80386的CPI进一步减小到4左右

80486的CPI已经接近2

Pentium处理机的CPI已经与RISC十分接近

目前，超标量、超流水线处理机的CPI已经达到
0.5，实际上用IPC (Instruction Per Cycle)更确切。

设计RISC机器的关键技术

- 1、在CPU中设置大量的寄存器组，并采用**重叠寄存器窗口**的技术；
- 2、指令采用重叠和**流水**的方式解释执行，并采用**优化延迟转移技术**；
- 3、在逻辑上采用**硬联实现为主，适当辅以微程序解释**的技术；
- 4、采用**优化编译技术**。

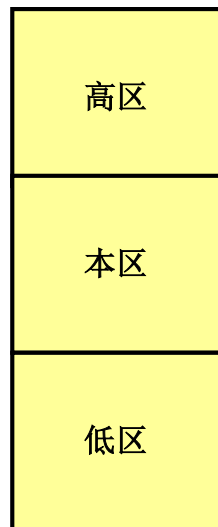
RISC的关键技术（简单版）

- 1、重叠寄存器窗口技术
- 2、流水和优化延迟转移技术
- 3、以硬件为主固件为辅技术
- 4、优化编译技术

■ 重叠寄存器窗口（Overlapping Register Windows）技术：

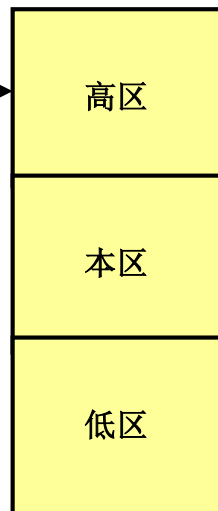
- 在RISC结构中，为减少过程调用中保存现场和建立新现场，及返回时恢复现场等辅助操作，通常设置一个数量很大的寄存器堆，并将所有寄存器分成若干个组，每组有若干的寄存器，称为寄存器窗口。
- 每个寄存器窗口又分为大小固定的三个区：高区，本区，低区。每个过程被调用时都会被分配一个寄存器窗口，高区用来存放调用者传来的参数，并存放返回给调用者的参数；本区存放局部变量，低区用于在本过程调用其他过程时传递参数给被调用过程，被调用过程执行完后的结果也存放在此区中。
- 每一对调用和被调用过程之间的寄存器窗口的低区（调用进程的）和高区（被调用进程的）相互重叠。一旦发生过程调用和返回，由一个窗口切换到另一个窗口时，这些参数可通过两个窗口重叠的寄存器自动被传送，而不需额外的传送时间。

过程A



重叠

过程B



重叠

过程C



重叠寄存器窗口技术

■ 流水和优化延迟转移技术：

- 流水是让本条指令的执行与下条指令的预取在时间上重叠起来。
- 延迟转移是在转移指令后增加空指令，减少转移方向错误所带来的损失，但是这样做又会浪费一个机器周期。
- 优化延迟转移则是将转移指令与其前面的一条指令对换位置，从而在准备将控制转向目标指令的同时，执行紧随转移指令后的那条指令，这样，预取的指令不会作废，不影响流水执行，也不会浪费机器周期。
- 优化延迟转移是由编译程序自动完成的，对应用程序员透明，对系统程序员不透明。

在流水线上执行转移指令时出现的问题

■ 100 LOAD X,R1

■ 101 ADD 1, R1

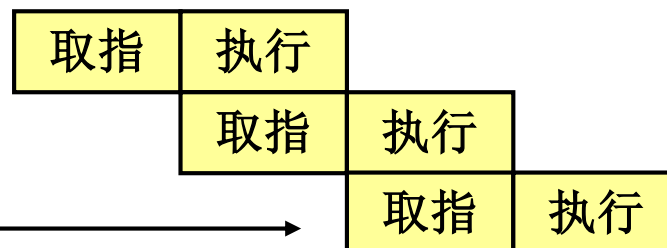
■ 102 **BRANCH 105**

■ 103 **ADD R1,R2** →

■ 104 SUB R3,R1

■ 105 **STORE R1,Y**

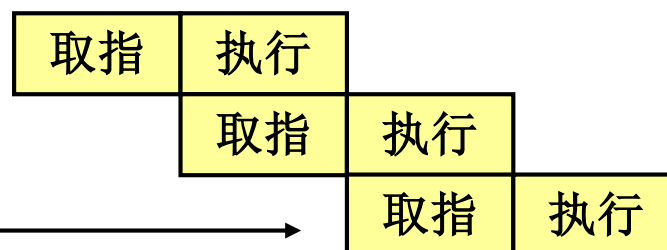
■ 106 ...



如果转移成功，则转移指令102后面应该执行105，但是转移指令未执行完时，103已经被调入流水线，被错误执行，程序运算结果将会出错。

延迟转移技术（插入空指令）

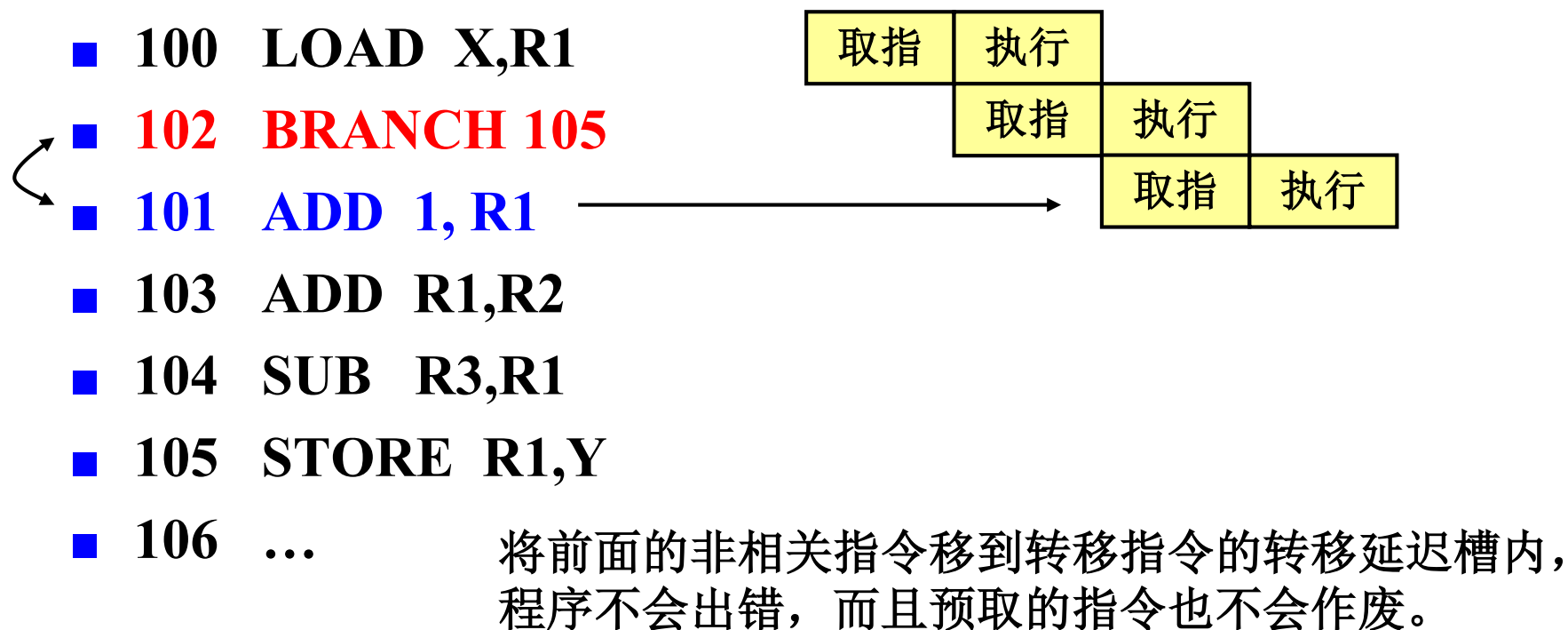
- 100 LOAD X,R1
- 101 ADD 1, R1
- 102 **BRANCH 105**
- **NOP**
- 103 ADD R1,R2
- 104 SUB R3,R1
- 105 STORE R1,Y
- 106 ...



在转移指令的转移延迟槽内添加空指令，程序不会出错，但是浪费了计算周期。

优化延迟转移技术

(将非相关指令移到转移指令的延迟槽内)



指令取消技术

- 采用优化延迟技术时，有时调整指令序列非常困难，找不到可以用来调整的有效指令。有些RISC处理机会采用**指令取消技术**。即根据转移指令的结果决定下面待执行指令是否应该取消。如果指令被取消，其效果相当于执行了一条空指令，不影响程序的运行。虽然如此，为了避免取消指令带来的流水线效率降低，还是应该尽量避免取消指令。可采用如下规则：
 - **如果是向后转移**（转移的目标地址小于当前程序计数器PC的值），则在转移不成功时取消下条指令，否则，执行下条指令。
 - **如果是向前转移**，则正好相反，在转移不成功时执行下条指令，否则，取消下条指令。

一个向后转移的例子

LOOP: XXX
YYY
... ..
ZZZ
COMP R1,R2,LOOP
WWW

调整前的程序

XXX
LOOP: YYY
... ..
ZZZ
COMP R1,R2,LOOP
XXX
WWW

调整后的程序

循环体的第一条指令“XXX”经调整后被复制在两个位置，一个在循环体前，进入循环体时要执行一次；一个在循环体后，如果COMP转移成功，则执行下面的XXX指令，然后返回到LOOP；如果转移不成功，则取消下面的XXX指令，接着执行WWW指令。

由于向后转移绝大多数都是成功的，只有循环结束后最后一次转移不成功。因此，采用这种指令取消技术能够使流水线在绝大多数情况下不断流，保持很高的流水效率。

一个向前转移的例子

- 例如：如下程序

RRR

.....

SSS

COMP R1, R2, THRU

TTT

.....

UUU

THRU: VVV

如果COMP指令的转移条件不成立，则下条指令TTT不取消，即转移指令后面的程序代码照常执行；如果转移条件成立，则下条指令TTT的执行被取消，程序转到THRU位置的VVV执行。

由于在向前转移时成功和不成功的概率相等，各为50%，因此，这里采用正常的指令取消技术就可以了。

- **硬件为主固件为辅的实现技术**：RISC主要采用硬连逻辑实现指令系统，对于那些必须的复杂指令，也可以用固件（微程序技术）实现。
- **优化编译技术**：主要目的是对指令进行重新排序和调度，优化代码顺序，减少目标代码长度，提高程序运行速度。主要突出两个方面的优化调度：一是如何最佳地分配RISC中大量寄存器的使用，从而减少访问存储器的次数；二是设法对程序中的指令序列进行调整，在保持原来语义正确的基础上尽量减少计算机的空等时间（例如前面所讲的指令延迟和指令取消技术）。

RISC优化编译技术

■ RISC对优化编译技术带来的方便有：

- RISC的指令系统简单，而且均匀、对称，优化编译程序不需要做复杂的指令选择工作。
- RISC的寻址方式简单,只有LOAD和STORE指令能够访问存储器，其他指令均在通用寄存器之间进行操作。因此简化了编译器选择寻址方式过程中的工作。
- 大多数指令都能在一个周期内执行完成，为优化编译器调整指令序列提供了极大的方便。

■ RISC对优化编译器造成的困难主要有：

- 优化编译器须精心安排每一个寄存器的用法；
- 优化编译器要做数据和控制相关性分析；
- 要设计复杂的子程序库。

RISC的优点

- 1、简化了指令系统的设计，适合于用VLSI实现；
- 2、提高了机器的运行速度和效率；
- 3、降低了设计成本，提高了系统的可靠性；

RISC的缺点

- 1、要设计复杂的子程序库，由于指令少，使原在CISC上由单一指令完成的某些复杂功能现在需要多条RISC指令才能完成。
- 2、对编译程序的设计质量要求较高，难度较大。相对来说，RISC机器上的编译程序要比CISC机器上的编译程序难写。如编译器需安排多个通用寄存器的使用，还要做数据和控制相关性分析，并设法调整指令序列，减少流水线断流的发生。
- 3、对浮点运算和虚拟存储器等的支持仍显不足；

发展趋势

- 今后，计算机发展改进的总趋势是：RISC和CISC互相结合，取长补短。

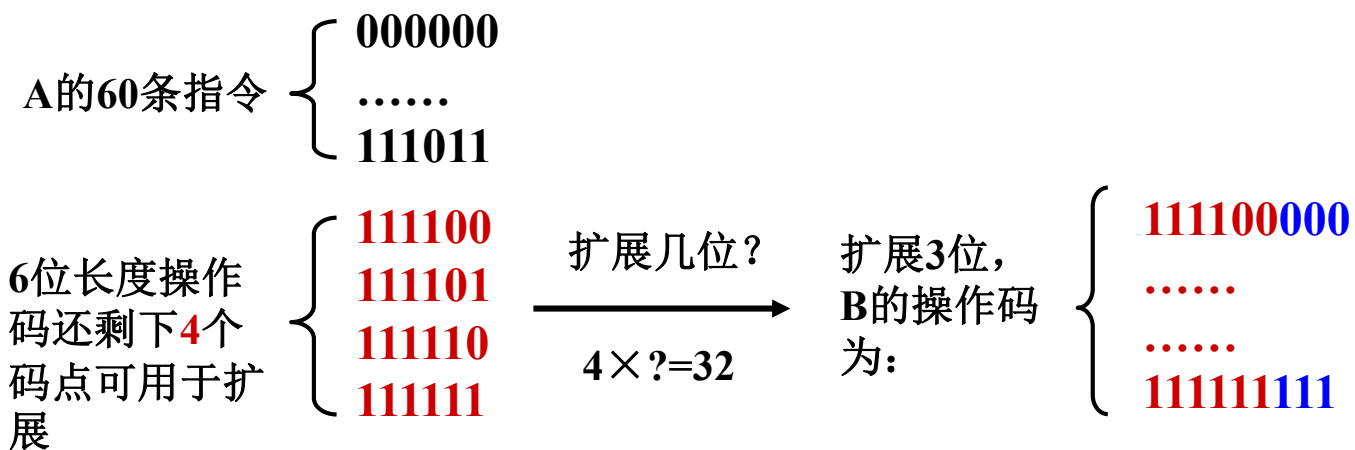
2.6 小结

- 本章主要内容有指令优化、指令系统发展方向（CISC, RISC）两个部分。具体细节如下：
- 指令优化
 - 操作码优化
 - 定长编码、Huffman编码、扩展编码方法
 - 编码方法性能指标（熵 H ，平均码长 L ，信息冗余量 R ）
 - 操作数优化
 - 地址表示形式、寻址方式、地址制；
 - 如何使操作数地址部分和操作码配合，使指令最短、最有效
- 指令系统发展方向
 - CISC、RISC定义；CISC的缺点及RISC的由来；
 - 设计RISC的一般原则
 - 设计RISC机器的关键技术


2.7 习题

- 有关指令优化的习题
- 有关RISC的习题

设计计算机A有60条指令，指令操作码为6位固定长度编码，从000000到111011。其后继产品B需要增加32条指令，并与A保持兼容，试采用操作码扩展技术为计算机B设计指令操作码。

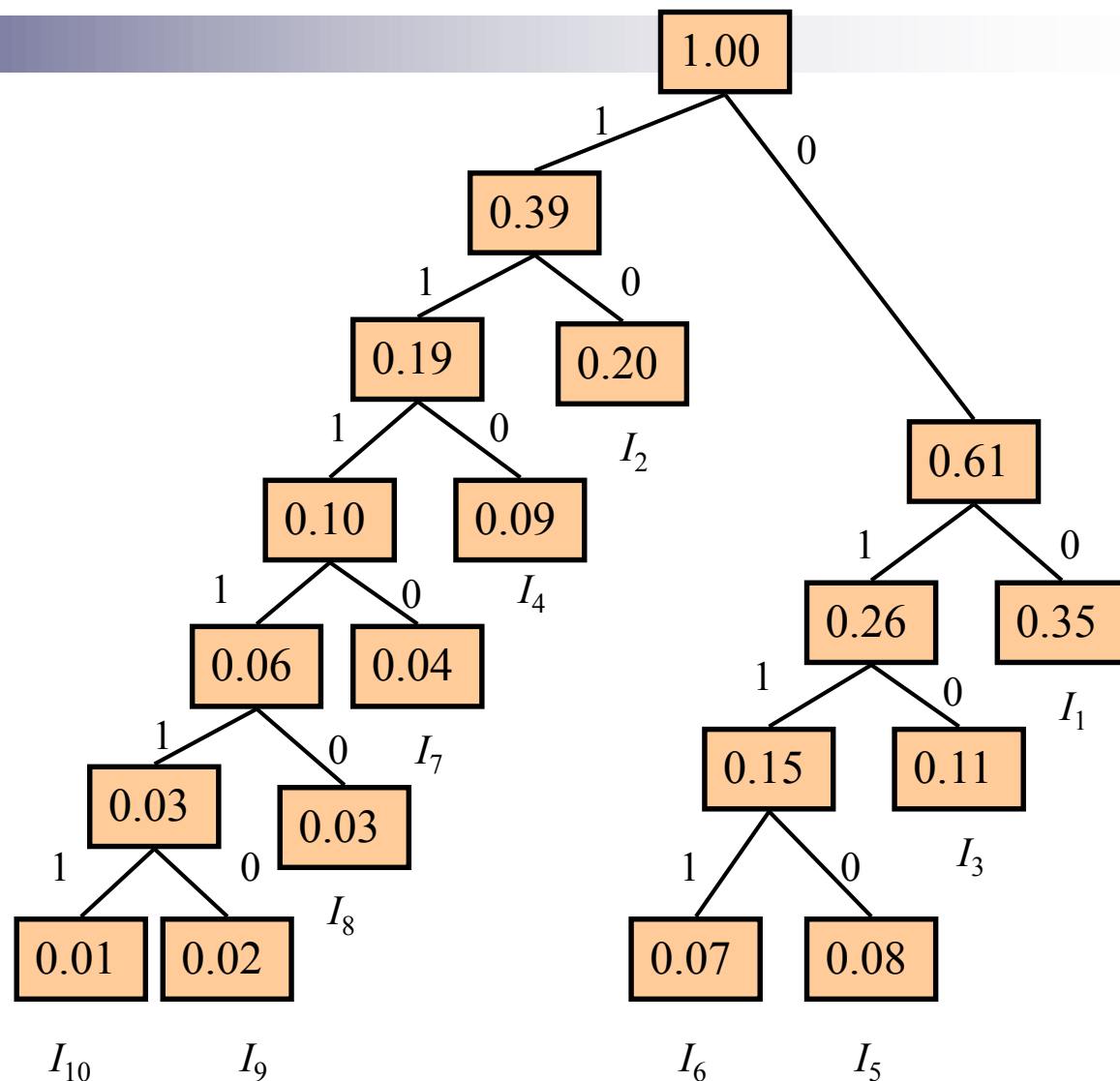


答: 6位操作码中保留了111100到111111四个码字，如果不需要再为将来进一步扩展保留码字，可增加3位扩展码，这样增加的32条指令的操作码为111100,000~111111,111。



某计算机有10条指令，它们的使用频率分别为0.35, 0.20, 0.11, 0.09, 0.08, 0.07, 0.04, 0.03, 0.02, 0.01，试用哈夫曼编码对它们的操作码进行编码，并计算平均编码长度。

指令	频率	Huffman编码	长度
I_1	0.35	00	2
I_2	0.20	10	2
I_3	0.11	010	3
I_4	0.09	110	3
I_5	0.08	0110	4
I_6	0.07	0111	4
I_7	0.04	1110	4
I_8	0.03	11110	5
I_9	0.02	111110	6
I_{10}	0.01	111111	6



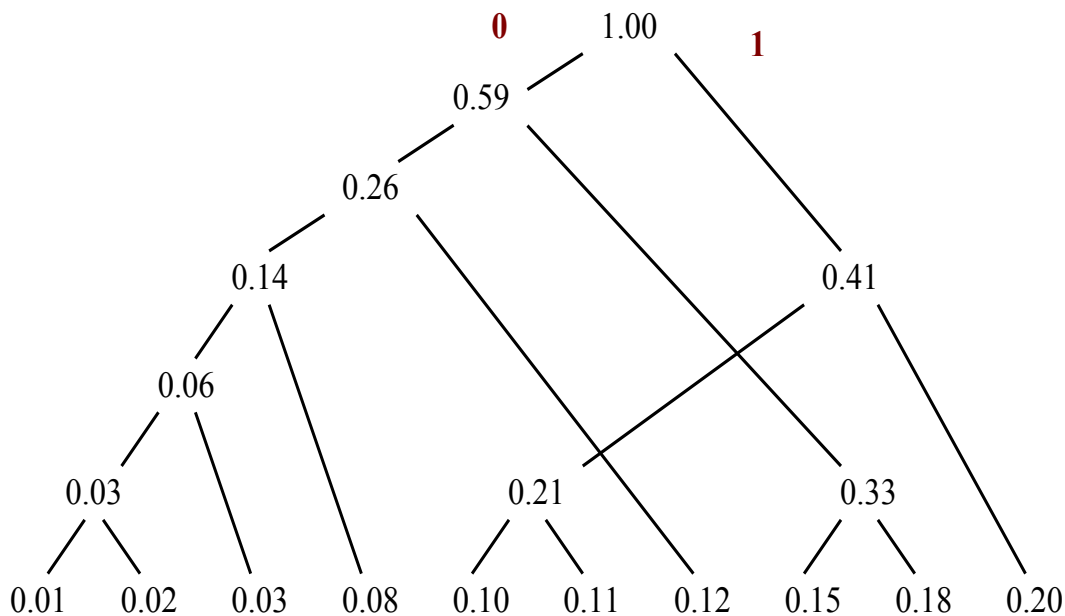
平均码长 $l = (0.35+0.20) \times 2 + (0.11+0.09) \times 3$
 $+ (0.08+0.07+0.04) \times 4 + 0.03 \times 5 + (0.02+0.01) \times 6 = 2.79$
 可以有其他类似编码结果，但是码长一定相同

设有一台简单计算机的指令系统共有10条指令，各指令的使用频率如下：

I_1 20%, I_2 12%, I_3 11%, I_4 15%, I_5 8%
 I_6 3%, I_7 2%, I_8 18%, I_9 10%, I_{10} 1%

(1) 用霍哈夫曼编码设计这10条指令的操作码，并计算操作码的平均编码长度；

答：(1) 哈夫曼树及编码为：



$$\text{平均码长 } l = 2 \times 20\% + 3 \times (18\% + 15\% + 12\% + 11\% + 10\%) + 4 \times 8\% + 5 \times 3\% + 6 \times (2\% + 1\%) = 3.03$$

指令	频率	Huffman编码	长度
I_1	0.20	11	2
I_8	0.18	011	3
I_4	0.15	010	3
I_2	0.12	001	3
I_3	0.11	101	3
I_9	0.10	100	3
I_5	0.08	0001	4
I_6	0.03	00001	5
I_7	0.02	000001	6
I_{10}	0.01	000000	6

(2) 分别用3-4扩展编码和7/3扩展编码，设计这10条指令的操作码，并计算操作码的平均编码长度；

答：(2) 3-4扩展编码和7/3扩展编码如表所示：

3-4扩展编码平均码长

$$\begin{aligned} l &= 3 \times (0.20 + 0.18 + 0.15 + 0.12 + 0.11 + 0.10) \\ &\quad + 4 \times (0.08 + 0.03 + 0.02 + 0.01) \\ &= 3 \times 0.86 + 4 \times 0.14 \\ &= 3.14 \end{aligned}$$

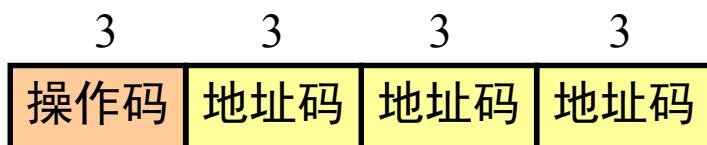
7/3扩展编码平均码长

$$\begin{aligned} l &= 3 \times (0.20 + 0.18 + 0.15 + 0.12 + 0.11 + 0.10 + 0.08) \\ &\quad + 5 \times (0.03 + 0.02 + 0.01) \\ &= 3 \times 0.94 + 5 \times 0.06 \\ &= 3.12 \end{aligned}$$

指令	频率	3-4扩展 编码	长度	7/3扩展 编码	长度
I_1	0.20	000	3	000	3
I_8	0.18	001	3	001	3
I_4	0.15	010	3	010	3
I_2	0.12	011	3	011	3
I_3	0.11	100	3	100	3
I_9	0.10	101	3	101	3
I_5	0.08	1100	4	110	3
I_6	0.03	1101	4	11100	5
I_7	0.02	1110	4	11101	5
I_{10}	0.01	1111	4	11110	5

- 若某计算机要求有如下形式的指令：三地址指令4条，单地址指令254条，零地址指令16条(不要求有二地址指令)。设指令字长为12位，每个地址码长为3位，试用扩展操作码为其编码。并画出3种指令格式，标出各段的长度。

三地址指令 4条



指令字长为12位，
每个地址码长为3
位

000

001

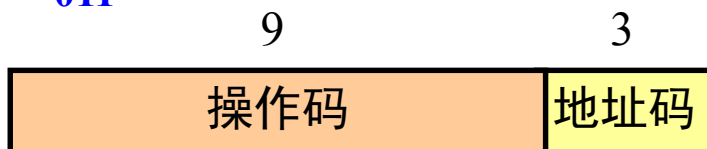
010

011

共有 $2^3=8$ 个码点，用掉4个

留四个码点100、101、110、111做为扩展

单地址指令 254条



100,000000

101.....

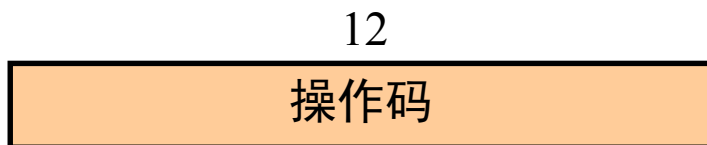
110.....

111,111101

共有 $4 \times 2^6=4 \times 64=256$ 个码点，用掉254个

留两个码点111,111110、111,111111做为扩展

零地址指令 16条



111,111110,000

.....

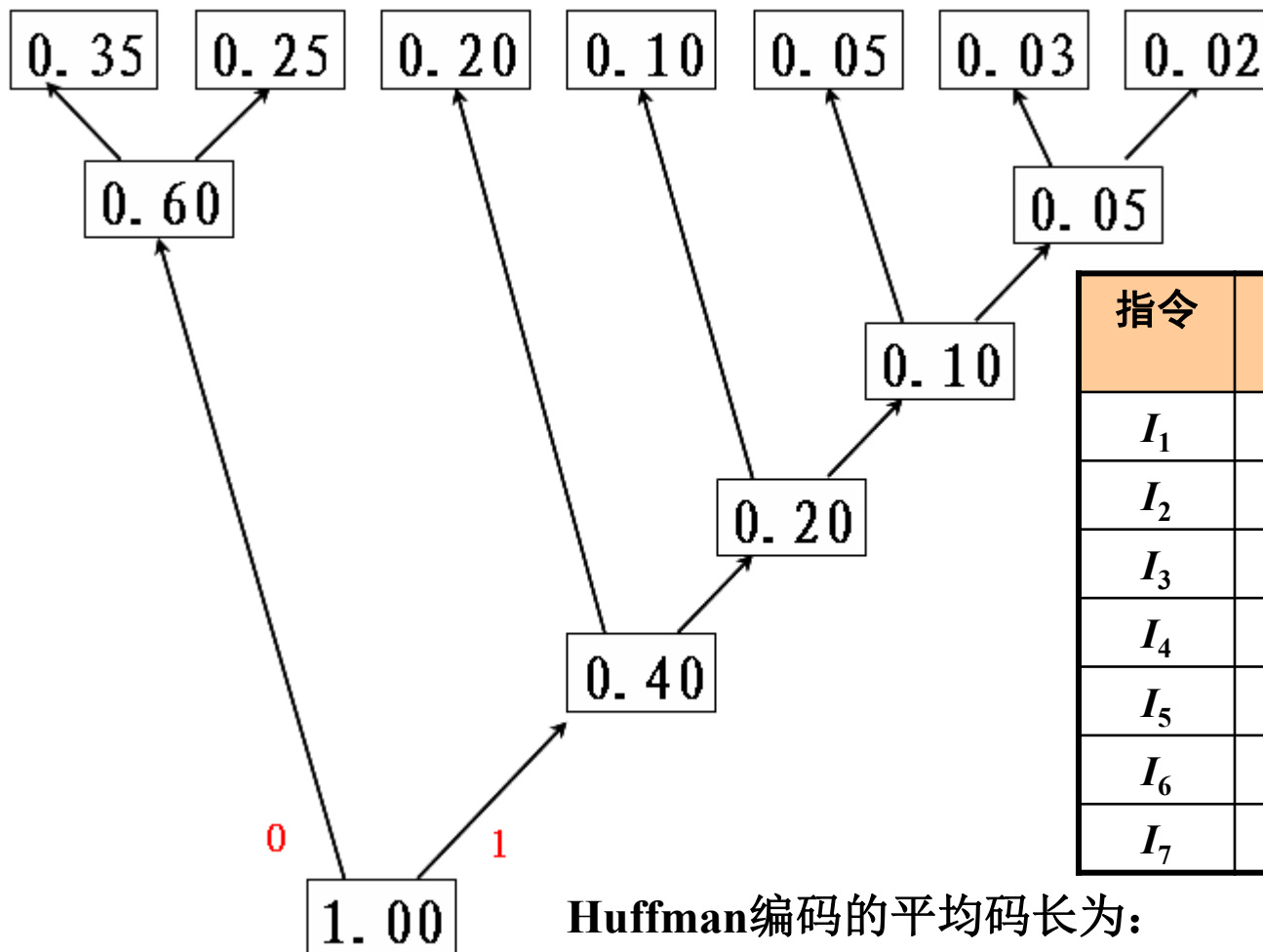
.....

111,111111,111

共有 $2 \times 2^3=2 \times 8=16$ 个码点，全部用掉，
刚好满足要求。

- 一台模型机共有7条指令，各指令的使用频率分别为35%，25%，20%，10%，5%，3%和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 ± 127 。请设计指令格式，并给出各字段的长度和操作码的编码。

[答] (1) 要使得到的操作码长度最短, 应采用Huffman编码, 构造Huffman树、编码及平均码长如下:



指令	频率	Huffman 编码	长度
I_1	0.35	00	2
I_2	0.25	01	2
I_3	0.20	10	2
I_4	0.10	110	3
I_5	0.05	1110	4
I_6	0.03	11110	5
I_7	0.02	11111	5

Huffman编码的平均码长为:

$$l = 2 \times (0.35 + 0.25 + 0.20) + 3 \times 0.10 + 4 \times 0.05 + 5 \times (0.03 + 0.02) = 1.6 + 0.3 + 0.2 + 0.25 = 2.35$$

(2) 设计8位字长的寄存器-寄存器型变址寻址方式指令如下，因为只有8个通用寄存器，所以每个寄存器地址需3位，操作码有 $8-3-3=2$ 位，设计格式如下：




三条指令的操作码分别为00，01，10，留下码点11作为扩展。

设计16位字长的寄存器-存储器型变址寻址方式指令如下：



四条指令的操作码分别为1100， 1101， 1110， 1111



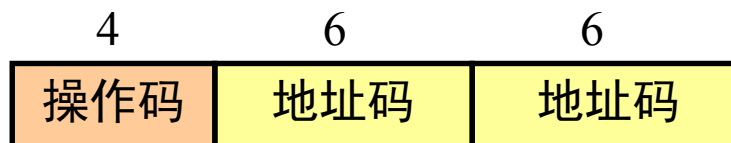
某处理机的指令字长为16位，有双地址指令、单地址指令和零地址指令三类，并假设每个地址字段的长度均为6位。

(1) 如果双地址指令有15条，单地址指令和零地址指令的条数基本相同，问单地址指令和零地址指令各有多少条？并且为这三类指令分配操作码。

(2) 如果要求三类指令的比例大致为1 : 9 : 9，问双地址指令、单地址指令和零地址指令各有多少条？并且为这三类指令分配操作码。

指令字长为16位，
每个地址码长为6位

双地址指令 15条



0000

0001 共有 $2^4=16$ 个码点，用掉15个

..... 留一个码点1111做为扩展

1110

10

6

单地址指令 ?条



1111,000000

1111.....

1111.....

1111,111110

共有 $1 \times 2^6 = 1 \times 64 = 64$ 个码点，用掉多少个，
留几个码点做为扩展？

1 : 1

16

零地址指令 ?条



111,111111,000000

.....

.....

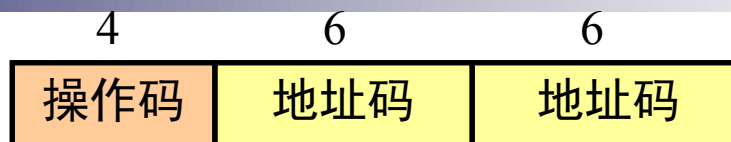
111,111111,111111

若前面留了1个扩展码点，则共有
 $1 \times 2^6 = 1 \times 64 = 64$ 个码点，单地址指令数:零
地址指令数=63:64 \approx 1:1，刚好满足要求。

若前面留了2个扩展码点，则共有
 $2 \times 2^6 = 2 \times 64 = 128$ 个码点，单地址指令数/
零地址指令数=62:128 \approx 1:2，不合要求。

指令字长为16位，
每个地址码长为6位

双地址指令 **14条**



0000

0001

.....

1101

共有 $2^4=16$ 个码点，**用掉多少个，留几个码点做为扩展？**

10

6

单地址指令 **126条**



1110,000000

1110.....

1111.....

1111,111101

1 : 9 : 9

若前面**留了1个码点**，共有 $1 \times 2^6 = 1 \times 64 = 64$ 个码点，双地址指令数/单地址指令数=15:64 \approx 1:4，**不合要求**。若前面留了2个码点，共有 $2 \times 2^6 = 2 \times 64 = 128$ 个码点，双地址指令数/单地址指令数=14:128 \approx 1:9，符合要求。

16

零地址指令 **128条**



111,111110,000000

.....

.....

111,111111,111111

若前面**留了1个码点**，共有 $1 \times 2^6 = 1 \times 64 = 64$ 个码点，单地址指令数/零地址指令数=127:64 \approx 2:1，**不合要求**。若前面留了2个码点，共有 $2 \times 2^6 = 2 \times 64 = 128$ 个码点，单地址指令数/零地址指令数=126:128 \approx 1:1，符合要求。

■ 名词解释

- **复杂指令系统计算机CISC** (Complex Instruction Set Computer): 通过强化指令集功能, 并实现软件功能向硬件功能转移来提高计算机的性能。
- **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子程序来实现, 通过尽可能降低指令集结构的复杂性, 来达到简化指令集的实现和提高性能的目的。

■ 简述CISC存在的主要问题

- 指令系统中，约有80%的指令的使用频度很低，利用率低，因此，降低了指令系统的性能价格比。
- 指令系统日趋庞大和复杂，不宜于用VLSI技术实现，使机器的设计周期延长，成本升高，设计错误增多，系统可靠性降低；
- 功能复杂的指令使指令的译码和操作变得复杂，并使得指令系统的指令平均周期数较大，增大了程序的执行时间，降低了程序的执行速度；
- 高级语言源程序的优化编译变得困难，编译的时空开销增大。

■ 简要说明RISC机器的设计原则。

- 1、精简指令条数，保留使用频度高的指令；
- 2、简化指令格式，采用简单寻址方式，绝大多数指令可以在单周期内执行完成；
- 3、扩大通用寄存器数量，只允许Load和Store指令可以访存；
- 4、指令以组合电路实现为主，少量指令可以用微程序解释方式执行；
- 5、优化编译程序的设计。

■ RISC采用了哪些关键技术？

- 1、重叠寄存器窗口技术
- 2、流水加延时转移技术
- 3、以硬件为主固件为辅技术
- 4、优化编译技术

■ 简述RISC的优缺点

优点：

- 1、简化了指令系统的设计，适合于用VLSI 实现；
- 2、提高了机器的运行速度和效率；
- 3、降低了设计成本，提高了系统的可靠性；

缺点：

- 1、要设计复杂的子程序库；
- 2、对编译程序的设计质量要求较高，难度较大；
- 3、对浮点运算和虚拟存储器等的支持仍显不足。

作业

- 6, 7, 11, 12, 13