

计算机组成原理

(Computer Organization Principles)

- 主讲教师 蒋永国 (jiangyg@ouc.edu.cn)
(信息学院南楼, B313室)
- 实验教师 徐惠敏 (yziping@ouc.edu.cn)
(信息学院南楼, B420室)

For Students of Computer 2017

Ch2: Data Representation

数据的机器级表示

第一讲 数值数据的表示

**第二讲 非数值数据表示及
数据的宽度、存储排列、纠/检错**

第一讲：数值数据的表示

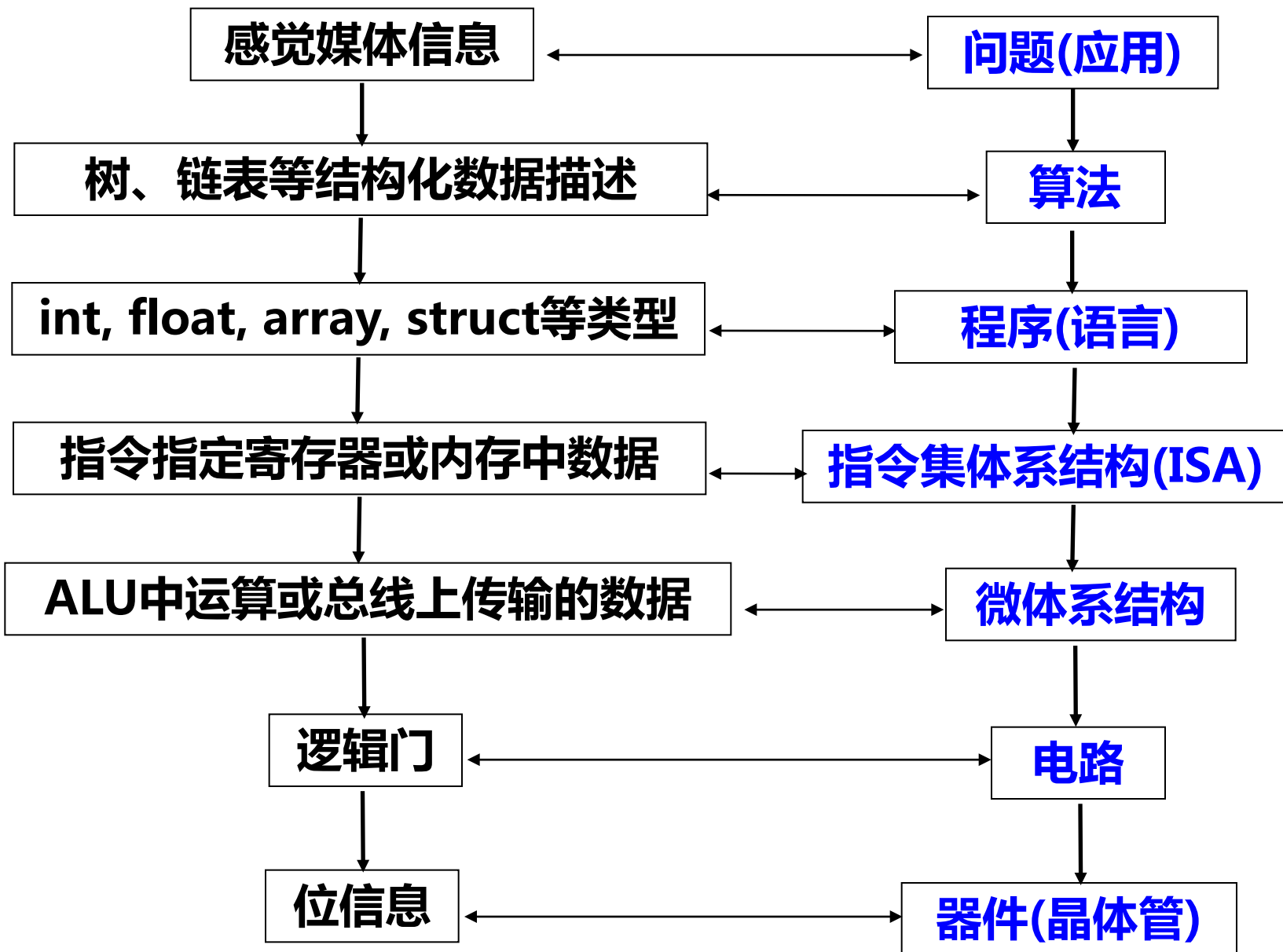
主 要 内 容

- ◆ 定点数的表示
 - 进位计数制
 - 定点数的二进制编码
 - 原码、补码、移码表示
 - 定点整数的表示
 - 无符号整数、带符号整数
- ◆ 浮点数格式和表示范围
- ◆ 浮点数的规格化
- ◆ IEEE754浮点数标准
 - 单精度浮点数、双精度浮点数
 - 特殊数的表示形式
- ◆ C语言程序中的整数类型、浮点数类型
- ◆ 十进制数表示

“转换”的概念在数据表示中的反映

抽象概括

具体实现



对连续信息采样，
以使信息离散化

对离散样本用0和1
进行编码

文字、图、表、声音、
视频等各种媒体信息

最终用户角度

输入设备

输出设备

二进制编码表示的各种数据

各类数据之间的
转换关系

数组、结构、字符串等结构化数据

高级语言程序员角度

指令系统能识别
的基本类型数据

低级语言程序员和
硬件系统设计者角度

数值型数据

非数值型数据

定点运算指令

二进制数

二进制编码的
十进制数

逻辑数据

编码字符
如：西文字符和汉字

整数（定点数）

实数（浮点数）

逻辑、位操作或字符处理指令

浮点运算指令

无符号整数

带符号整数

信息的二进制编码

◆ 计算机的外部信息与内部机器级数据

◆ 机器级数据分两大类：

- 数值数据：无符号整数、带符号整数、浮点数（实数）、十进制数
- 非数值数据：逻辑数（包括位串）、西文字符和汉字

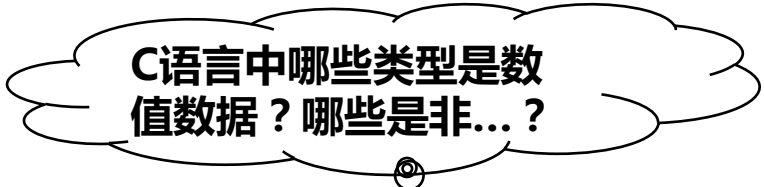
◆ 计算机内部所有信息都用二进制（即：0和1）进行编码

◆ 用二进制编码的原因：

- 制造二个稳定态的物理器件容易
- 二进制编码、计数、运算规则简单
- 正好与逻辑命题对应，便于逻辑运算，并可方便地用逻辑电路实现算术运算

◆ 真值和机器数

- 机器数：用0和1编码的计算机内部的0/1序列
- 真值：机器数真正的值，即：现实中带正负号的数



C语言中哪些类型是数值数据？哪些是非...？

首先考虑数值数据的表示

数值数据的表示

◆ 数值数据表示的三要素

- 进位计数制
- 定、浮点表示
- 如何用二进制编码

即：要确定一个数值数据的值必须先确定这三个要素。

例如，机器数 01011001 的值是多少？ 答案是：不知道！

◆ 进位计数制

- 十进制、二进制、十六进制、八进制数及其相互转换

◆ 定/浮点表示（解决小数点问题）

- 定点整数、定点小数
- 浮点数（可用一个定点小数和一个定点整数来表示）

◆ 定点数的编码（解决正负号问题）

- 原码、补码、反码、移码（反码很少用）

Sign and Magnitude （原码的表示）

Decimal	Binary
---------	--------

0	0000
---	------

1	0001
---	------

2	0010
---	------

3	0011
---	------

4	0100
---	------

5	0101
---	------

6	0110
---	------

7	0111
---	------

Decimal	Binary
---------	--------

-0	1000
----	------

-1	1001
----	------

-2	1010
----	------

-3	1011
----	------

-4	1100
----	------

-5	1101
----	------

-6	1110
----	------

-7	1111
----	------

◆ 容易理解， 但是：

- ✓ 0 的表示不唯一，故不利于程序员编程
- ✓ 加、减运算方式不统一
- ✓ 需额外对符号位进行处理，故不利于硬件设计
- ✓ 特别当 $a < b$ 时，实现 $a - b$ 比较困难

从 50年代开始，整数都采用补码来表示
但浮点数的尾数用原码定点小数表示

补码特性 - 模运算 (modular运算)

重要概念：在一个模运算系统中，一个数与它除以“模”后的余数等价。

时钟是一种模12系统

假定钟表时针指向10点，要将它拨向6点，则有两种拨法：

① 倒拨4格： $10 - 4 = 6$

② 顺拨8格： $10 + 8 = 18 \equiv 6 \pmod{12}$

模12系统中： $10 - 4 \equiv 10 + 8 \pmod{12}$

$-4 \equiv 8 \pmod{12}$

则，称8是-4对模12的补码（即：-4的模12补码等于8）。

同样有 $-3 \equiv 9 \pmod{12}$

$-5 \equiv 7 \pmod{12}$ 等

结论1：一个负数的补码等于模减该负数的绝对值。

结论2：对于某一确定的模，某数减去小于模的另一数，总可以用该数加上另一数负数的补码来代替。

补码 (modular运算)：+ 和- 的统一

补码的表示

现实世界的模运算系统举例

例1：“钟表”模运算系统

假定时针只能顺拨，从10点倒拨4格后是几点？

$$10 - 4 = 10 + (12 - 4) = 10 + 8 = 6 \pmod{12}$$

例2：“4位十进制数”模运算系统

假定算盘只有四档，且只能做加法，则在算盘上计算

9828-1928等于多少？

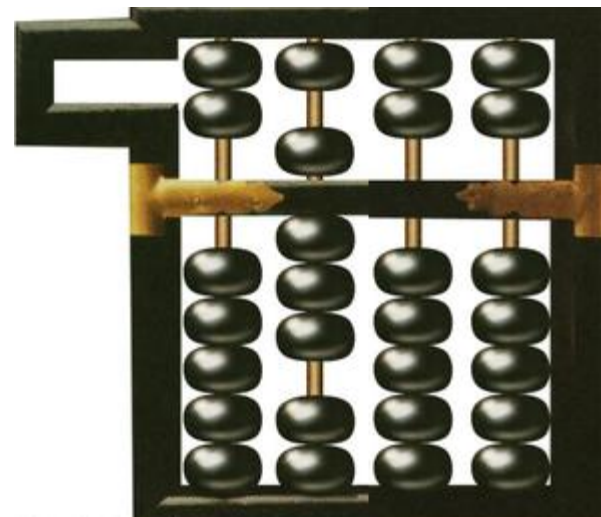
$$9828 - 1928 = 9828 + (10^4 - 1928)$$

$$= 9828 + 8072$$

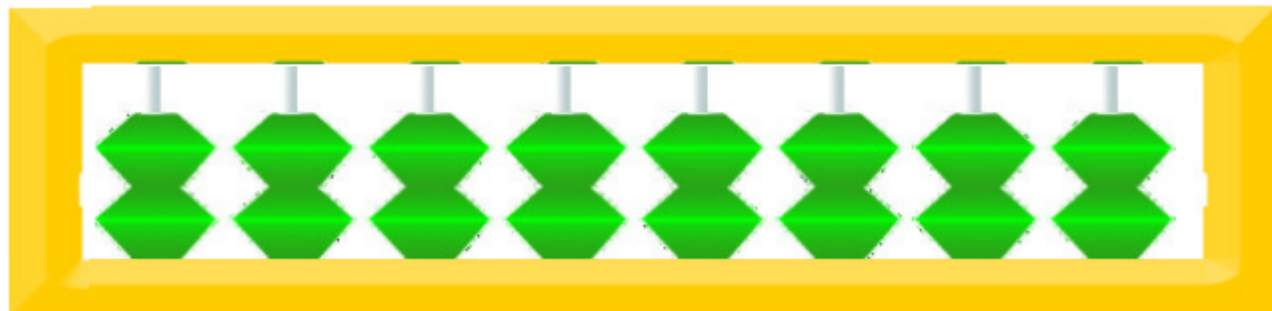
$$= \boxed{1}7900$$

$$= 7900 \pmod{10^4}$$

取模即只留余数，高位“1”被丢弃！
相当于只有低4位留在算盘上。



计算机中的运算器是模运算系统



8位二进制加法器模运算系统

计算 $0111\ 1111 - 0100\ 0000 = ?$

$$0111\ 1111 - 0100\ 0000 = 0111\ 1111 + (2^8 - 0100\ 0000)$$

$$= 0111\ 1111 + 1100\ 0000 = \boxed{1}0011\ 1111 \pmod{2^8}$$

$= 0011\ 1111$

只留余数，“1”被丢弃

结论1： 一个负数的补码等于对应正数补码的“各位取反、末位加1”

运算器是一个模运算系统，适合用补码表示和运算

计算机中运算器只有有限位。假定为 n 位，则运算结果只能保留低 n 位，故可看成是个只有 n 档的二进制算盘。所以，其模为 2^n 。

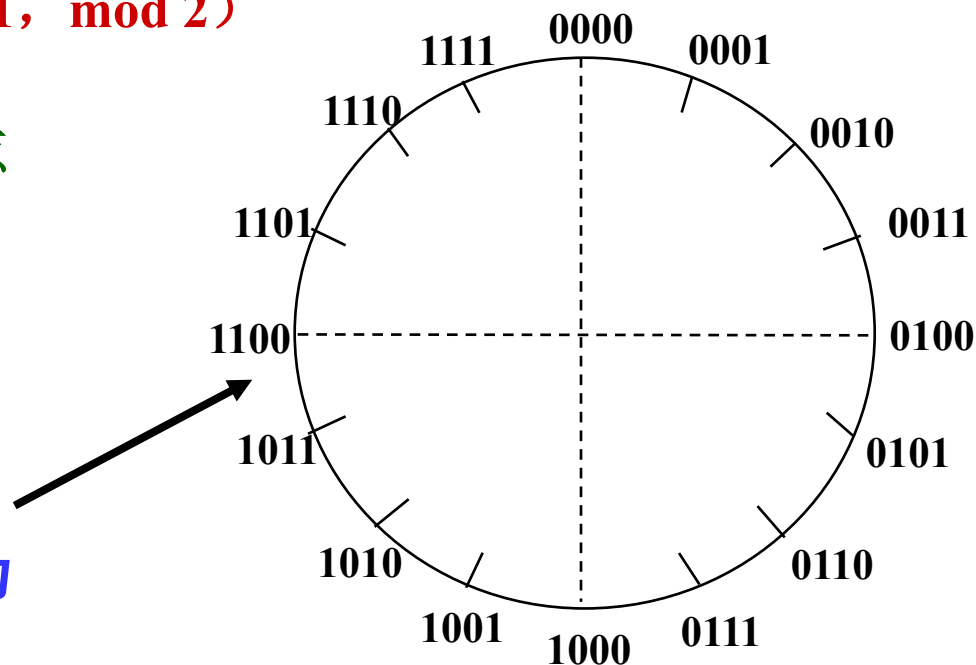
补码的定义 假定补码有 n 位，则：

定点整数： $[X]_{\text{补}} = 2^n + X \quad (-2^n \leq X < 2^n, \text{ mod } 2^n)$

定点小数： $[X]_{\text{补}} = 2 + X \quad (-1 \leq X < 1, \text{ mod } 2)$

注：实际上在计算机中并不使用定点小数表示！不需要掌握这个知识点。

当 $n=4$ 时，共有16个机器数：
 $0000 \sim 1111$ ，可看成是模为
 2^4 的钟表系统。真值的范围为
 $-8 \sim +7$



求特殊数的补码

假定机器数有n位

$$\textcircled{1} [-2^{n-1}]_{\text{补}} = 2^n - 2^{n-1} = 10\dots0 \text{ (n-1个0)} \quad (\text{mod } 2^n)$$

$$\textcircled{2} [-1]_{\text{补}} = 2^n - 0\dots01 = 11\dots1 \text{ (n个1)} \quad (\text{mod } 2^n)$$

$$\textcircled{3} [-1.0]_{\text{补}} = 2 - 1.0 = 1.00\dots0 \text{ (n-1个0)} \quad (\text{mod } 2)$$

$$\textcircled{4} [+0]_{\text{补}} = [-0]_{\text{补}} = 00\dots0 \text{ (n个0)}$$

注：计算机中并不会出现-1.0的补码，这里只是想说明同一个真值在机器中可能有不同的机器数！

补码与真值之间的简便转换


例: 设机器数有8位, 求123和-123的补码表示。

如何快速得到123的二进制表示?

解: $123 = 127 - 4 = 01111111\text{B} - 100\text{B} = 01111011\text{B}$

$-123 = -01111011\text{B}$

$[01111011]_{\text{补}} = 2^8 + 01111011 = 100000000 + 01111011$
 $= 01111011 \pmod{2^8}$, 即 7BH。

$[-01111011]_{\text{补}} = 2^8 - 01111011 = 10000\ 0000 - 01111011$
 $= 1111\ 1111 - 0111\ 1011 + 1$
 $= 1000\ 0100 + 1$  各位取反, 末位加1
 $= 1000\ 0101$, 即 85H。

Two's Complement (补码的表示)

- ◆ 正数：符号位 (sign bit) 为0，数值部分不变
- ◆ 负数：符号位为1，数值部分“各位取反，末位加1”

变形 (模4) 补码：双符号，用于存放可溢出的中间结果。

Decimal	补码	变形补码	Decimal	Bitwise Inverse	补码	变形补码
0	0000	00000	-0	1111	0000	00000
1	0001	00001	-1	1110	1111	11111
2	0010	00010	-2	1101	1110	11110
3	0011	00011	-3	1100	1101	11101
4	0100	00100	-4	1011	1100	11100
5	0101	00101	-5	1010	1011	11011
6	0110	00110	-6	1001	1010	11010
7	0111	00111	-7	1000	1001	11001
8	1000	01000	-8	0111	1000	11000

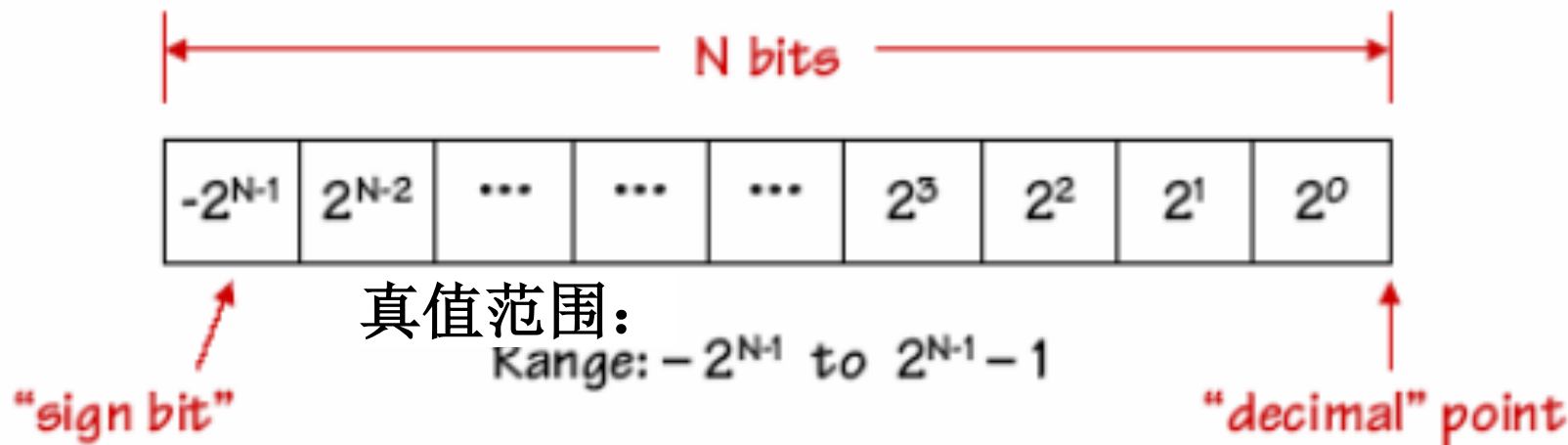
+0和-0
表示唯一

值太大，用4位补码无法表示，故“溢出”！但用变形补码可保留符号位和最高数值位。

如何求补码的真值

令: $[A]_{\text{补}} = a_{n-1}a_{n-2}\cdots a_1a_0$

则: $A = -a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0$



8-bit 2's complement example:

$$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$$

符号为0, 则为正数, 数值部分同

符号为1, 则为负数, 数值各位取反, 末位加1

例如: 补码 "11010110" 的真值为: $-0101010 = -(32+8+2) = -42$

Excess (biased) notion- 移码表示

- 什么是“excess (biased) notation-移码表示”？

将每一个数值加上一个偏置常数 (Excess / bias)

- 一般来说，当编码位数为 n 时，bias取 2^{n-1}

Ex. $n=4$: $E_{\text{biased}} = E + 2^3$ (bias= $2^3 = 1000\text{B}$)

-8 (+8) ~ 0000B

-7 (+8) ~ 0001B

...

0 (+8) ~ 1000B

...

+7 (+8) ~ 1111B

0的移码表示唯一

移码和补码仅第一位不同

移码主要用来表示

- 为什么要用移码来表示指数（阶码）？

浮点数阶码！

便于浮点数加减运算时的对阶操作（比较大小）

例: $1.01 \times 2^{-1} + 1.11 \times 2^3$

补码: $111 < 011$?
(-1) (3)

简化比较

$1.01 \times 2^{-1+4} + 1.11 \times 2^{3+4}$

移码: $011 < 111$
(3) (7)

Unsigned integer(无符号整数)

- ◆ 机器中字的位排列顺序有两种方式：（例：32位字： $0\dots01011_2$ ）
 - 高到低位从左到右：0000 0000 0000 0000 0000 0000 0000 1011 ← **LSB**
 - 高到低位从右到左：1101 0000 0000 0000 0000 0000 0000 0000 ← **MSB**
 - Leftmost和rightmost这两个词有歧义，故用**LSB(Least Significant Bit)**来表示最低有效位，用**MSB**来表示最高有效位
 - 高位到低位多采用从左往右排列
- ◆ 一般在全部是正数运算且不出现负值结果的情况下，可使用无符号数表示。例如，地址运算，编号表示，等等
- ◆ 无符号数的编码中**没有符号位**
- ◆ 能表示的最大值大于位数相同的带符号整数的最大值（Why?）
 - 例如，8位无符号整数最大是255（1111 1111）
而8位带符号整数最大为127（0111 1111）
- ◆ 总是整数，所以很多时候就**简称为“无符号数”**

Signed integer（带符号整数，定点整数）

- ◆ 计算机必须能处理正数(positive) 和负数(negative), MSB表示数符
- ◆ 有三种定点编码方式
 - Signed magnitude（原码）
现用来表示浮点（实）数的尾数
 - One's complement（反码）
现已不用于表示数值数据
 - Two's complement（补码）
50年代以来，所有计算机都用补码来表示定点整数
- ◆ 为什么用补码表示带符号整数？
 - 补码运算系统是模运算系统，加、减运算统一
 - 数0的表示唯一，方便使用
 - 比原码和反码多表示一个最小负数
 - 与移码相比，其符号位和真值的符号对应关系清楚

带符号整数和无符号数的比较

◆ 扩充操作有差别

- 例如，MIPS提供了两种加载指令（**load byte unsigned / load byte**）
 - 无符号数：`lbu $t0, 0($s0)`；\$t0高24位补0（称为0扩展）
 - 带符号整数：`lb $t0, 0($s0)`；\$t0高24位补符（称为符号扩展）

◆ 数的比较有差异

- 无符号数：MSB为1的数比MSB为0的数大
- 带符号整数：MSB为1的数比MSB为0的数小
- 例如，MIPS中提供了不同的比较指令，如：
 - 无符号数：`sltu $t0, $s0, $s1`（**set less than unsigned**）
 - 带符号整数：`slt $t1, $s0, $s1`（**set less than**）

假定：\$s0=1111 1111 1111 1111 1111 1111 1111 1111

\$s1=0000 0000 0000 0000 0000 0000 0000 0001

则：\$t0和\$t1分别为多少？ 答案：\$t0和\$t1分别为0和1。

◆ 溢出判断有差异（无符号数根据最高位是否有进位判断溢出，通常不判）

- MIPS规定：无符号数运算溢出时，不产生“溢出异常”

SKIP

扩展操作举例

例1（扩展操作）：在32位机器上输出si, usi, i, ui的十进制（真值）和十六进制值（机器数）是什么？

short si = -32768;

unsigned short usi = si;

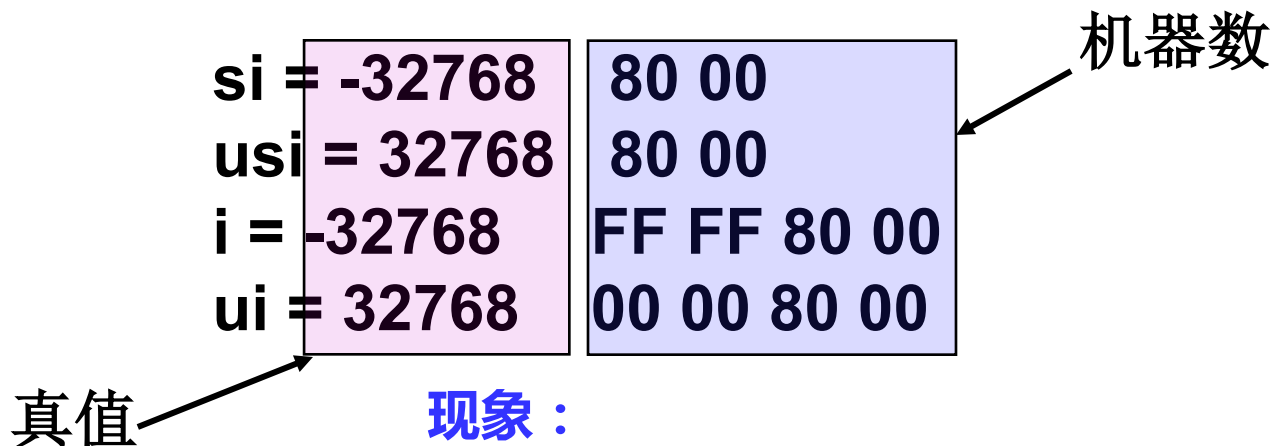
int i = si;

unsigned ui = usi ;

提示：

$32768 = 2^{15}$

= 1000 0000 0000 0000B



现象：

带符号整数：符号扩展

无符号数：0扩展

C语言程序中的整数

无符号数： `unsigned int (short / long)`；带符号整数： `int (short / long)`

常在一个数的后面加一个“u”或“U”表示无符号数

若同时有无符号和带符号整数，则C编译器将带符号整数强制转换为无符号数

假定以下关系表达式在32位用补码表示的机器上执行，结果是什么？

关系表达式	运算类型	结果	说明
0 == 0U			
-1 < 0			
-1 < 0U			
2147483647 > -2147483647-1			
2147483647U > -2147483647-1			
2147483647 > (int) 2147483648U			
-1 > -2			
(unsigned) -1 > -2			

C语言程序中的整数

关系表达式	类型	结果	说明
$0 == 0U$	无	1	$00...0B = 00...0B$
$-1 < 0$	带	1	$11...1B (-1) < 00...0B (0)$
$-1 < 0U$	无	0*	$11...1B (2^{32}-1) > 00...0B(0)$
$2147483647 > -2147483647 - 1$	带	1	$011...1B (2^{31}-1) > 100...0B (-2^{31})$
$2147483647U > -2147483647 - 1$	无	0*	$011...1B (2^{31}-1) < 100...0B(2^{31})$
$2147483647 > (int) 2147483648U$	带	1*	$011...1B (2^{31}-1) > 100...0B (-2^{31})$
$-1 > -2$	带	1	$11...1B (-1) > 11...10B (-2)$
$(unsigned) -1 > -2$	无	1	$11...1B (2^{32}-1) > 11...10B (2^{32}-2)$

带*的结果与常规预想的相反！

科学计数法(Scientific Notation)与浮点数

Example:

mantissa (尾数) \rightarrow 6.02 \times 10²¹ \leftarrow *exponent* (阶码、指数)
 \nwarrow *decimal point* \nwarrow *radix* (base, 基)

- **Normalized form (规格化形式)** : 小数点前只有一位非0数
- 同一个数有多种表示形式。例：对于数 1/1,000,000,000
 - Normalized (唯一的规格化形式): 1.0×10^{-9}
 - Unnormalized (非规格化形式不唯一) : 0.1×10^{-8} , 10.0×10^{-10}

for Binary Numbers:

mantissa (尾数) \rightarrow 0.101_{two} \times 2⁻¹⁰ \leftarrow *exponent* (指数)
 \nwarrow *binary point* \nwarrow 基为2

只要对尾数和指数分别编码，就可表示一个浮点数（即：实数）

浮点数(Floating Point)的表示范围

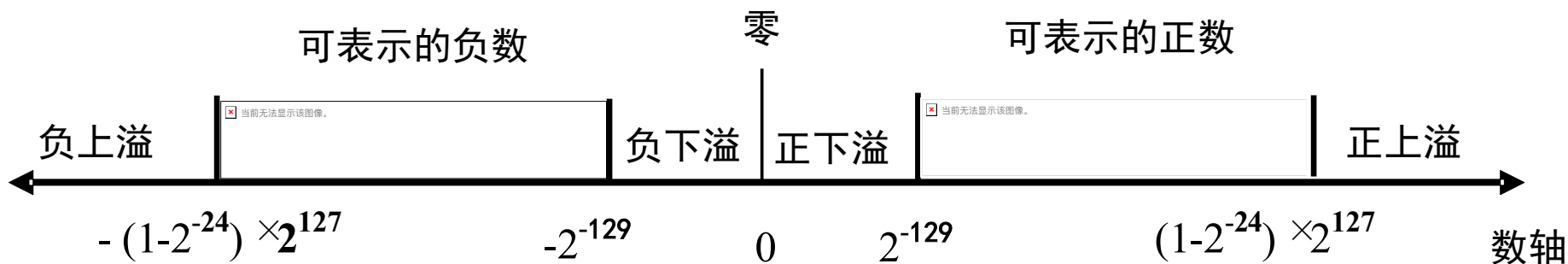
例：画出下述32位浮点数格式的规格化数的表示范围。



第0位数符S；第1～8位为8位移码表示阶码E（偏置常数为128）；第9～31位为24位二进制原码小数表示的尾数M。规格化尾数的小数点后第一位总是1，故规定第一位默认的“1”不明显表示出来。这样可用23个数位表示24位尾数。

最大正数：0.11...1 × 2^{11...1} = (1-2⁻²⁴) × 2¹²⁷ 最小正数：0.10...0 × 2^{00...0} = (1/2) × 2⁻¹²⁸

因为原码是对称的，所以其表示范围关于原点对称。



机器0：尾数为0 或 落在下溢区中的数

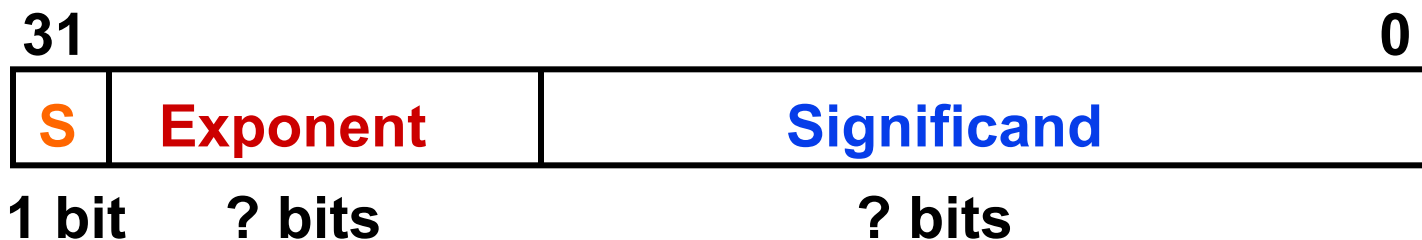
浮点数范围比定点数大，但数的个数没变多，故数之间更稀疏，且不均匀

浮点数的表示

- Normal format（规格化数形式）：

$$+/-1.\text{xxxxxxxxxx} \times 2^{\text{Exponent}}$$

- 32-bit 规格化数：



S 是符号位（Sign）

Exponent 用移码（增码）来表示

Significand 表示 **xxxxxxxxxxxx**，尾数部分

（基可以是 2 / 4 / 8 / 16，约定信息，无需显式表示）

- 早期的计算机，各自定义自己的浮点数格式

问题：浮点数表示不统一会带来什么问题？

规定：小数点前总是“1”，故可隐含表示

注意：和前面例子的规定不太一样，显然这里更合理！

“Father” of the IEEE 754 standard

直到80年代初，各个机器内部的浮点数表示格式还没有统一
因而相互不兼容，机器之间传送数据时，带来麻烦

1970年代后期，IEEE成立委员会着手制定浮点数标准

1985年完成浮点数标准IEEE 754的制定

现在所有计算机都采用IEEE 754来表示浮点数

This standard was primarily the work of one person, UC Berkeley math professor William Kahan.



www.cs.berkeley.edu/~wkahan/ieee754status/754story.html



Prof. William Kahan

IEEE 754 Floating Point Standard

规格化数 : $+/-1.\text{xxxxxxxxxx}_{\text{two}} \times 2^{\text{Exponent}}$

Single Precision : (Double Precision is similar)

S	Exponent	Significand
1 bit	8 bits	23 bits

- Sign bit: 1 表示negative ; 0表示 positive
- Exponent (阶码 / 指数) : 全0和全1用来表示特殊值!
 - SP规格化数阶码范围为0000 0001 (-126) ~ 1111 1110 (127)
 - bias为127 (single), 1023 (double)
- Significand (尾数) : 为什么用127? 若用128, 则阶码范围为多少?
 - 规格化尾数最高位总是1, 所以隐含表示, 省1位
 - 1 + 23 bits (single) , 1 + 52 bits (double)

SP: $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$ 0000 0001 (-127) ~ 1111 1110 (126)

DP: $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-1023)}$

Ex: Converting Binary FP to Decimal

BEE00000H is the hex. Rep. Of an IEEE 754 SP FP number

1 0111 1101 110 0000 0000 0000 0000 0000

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

- **Sign:** 1 => negative
- **Exponent:**
 - $0111\ 1101_{\text{two}} = 125_{\text{ten}}$
 - Bias adjustment: $125 - 127 = -2$
- **Significand:**
$$1 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + \dots$$
$$= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$$
- **Represents:** $-1.75_{\text{ten}} \times 2^{-2} = -0.4375$

Ex: Converting Decimal to FP

-12.75

1. Denormalize: -12.75

2. Convert integer part:

$$12 = 8 + 4 = 1100_2$$

3. Convert fractional part:

$$.75 = .5 + .25 = .11_2$$

4. Put parts together and normalize:

$$1100.11 = 1.10011 \times 2^3$$

5. Convert exponent: $127 + 3 = 128 + 2 = 1000\ 0010_2$

11000	0010	100	1100	0000	0000	0000	0000
-------	------	-----	------	------	------	------	------

The Hex rep. is C14C0000H

Normalized numbers (规格化数)

前面的定义都是针对规格化数 (normalized form)

How about other patterns?

Exponent	Significand	Object
1-254	anything implicit leading 1	Norms
0	0	?
0	nonzero	?
255	0	?
255	nonzero	?

Representation for 0

How to represent 0?

exponent: all zeros

significand: all zeros

What about sign? Both cases valid.

+0: 0 00000000 000000000000000000000000

-0: 1 00000000 000000000000000000000000

Representation for $+\infty/-\infty$

∞ : infinity

In FP, 除数为0的结果是 $\pm\infty$, 不是溢出异常. (整数除0为异常)

为什么要这样处理?

- 可以利用 $+\infty/-\infty$ 作比较。 例如: $X/0 > Y$ 可作为有效比较

How to represent $+\infty/-\infty$?

- **Exponent** : all ones (11111111B = 255)
- **Significand**: all zeros

$+\infty$: 0 11111111 000000000000000000000000

$-\infty$: 1 11111111 000000000000000000000000

Operations

$$5.0 / 0 = +\infty, \quad -5.0 / 0 = -\infty$$

$$5 + (+\infty) = +\infty, \quad (+\infty) + (+\infty) = +\infty$$

$$5 - (+\infty) = -\infty, \quad (-\infty) - (+\infty) = -\infty \quad \text{etc}$$

Representation for “Not a Number”

$\text{Sqrt}(-4.0) = ?$ $0/0 = ?$

- Called **Not a Number (NaN)** - “非数”

How to represent NaN

Exponent = 255

Significand: nonzero

NaNs can help with debugging

Operations

$\text{sqrt}(-4.0) = \text{NaN}$

$\text{op}(\text{NaN}, x) = \text{NaN}$

$+\infty - (+\infty) = \text{NaN}$

etc.

$0/0 = \text{NaN}$

$+\infty + (-\infty) = \text{NaN}$

$\infty/\infty = \text{NaN}$

Representation for Denorms(非规格化数)

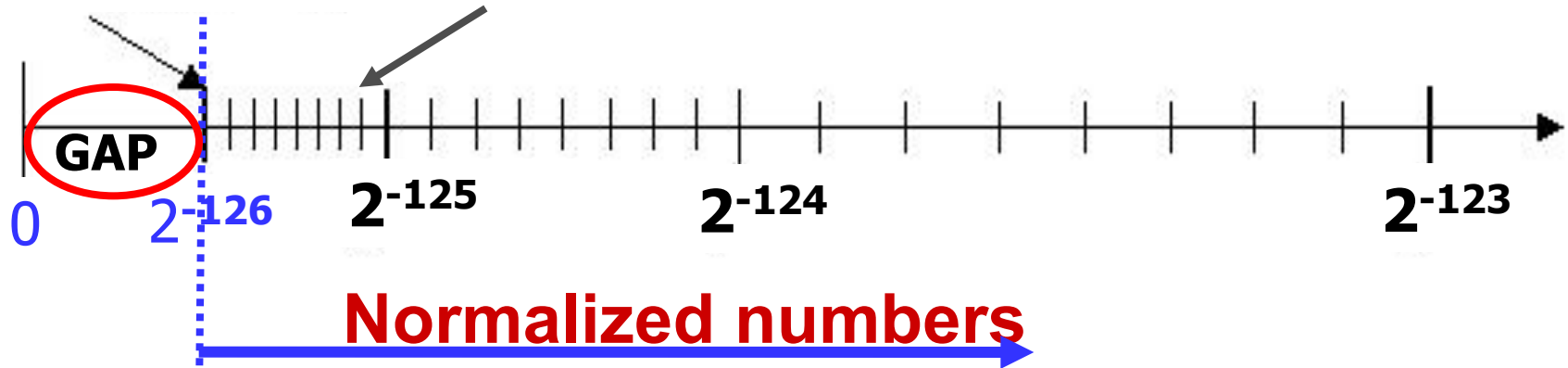
What have we defined so far? (for SP)

Exponent	Significand	Object
0	0	+/-0
0	nonzero	Denorms
1-254	anything implicit leading 1	Norms
255	0	+/- infinity
255	nonzero	NaN

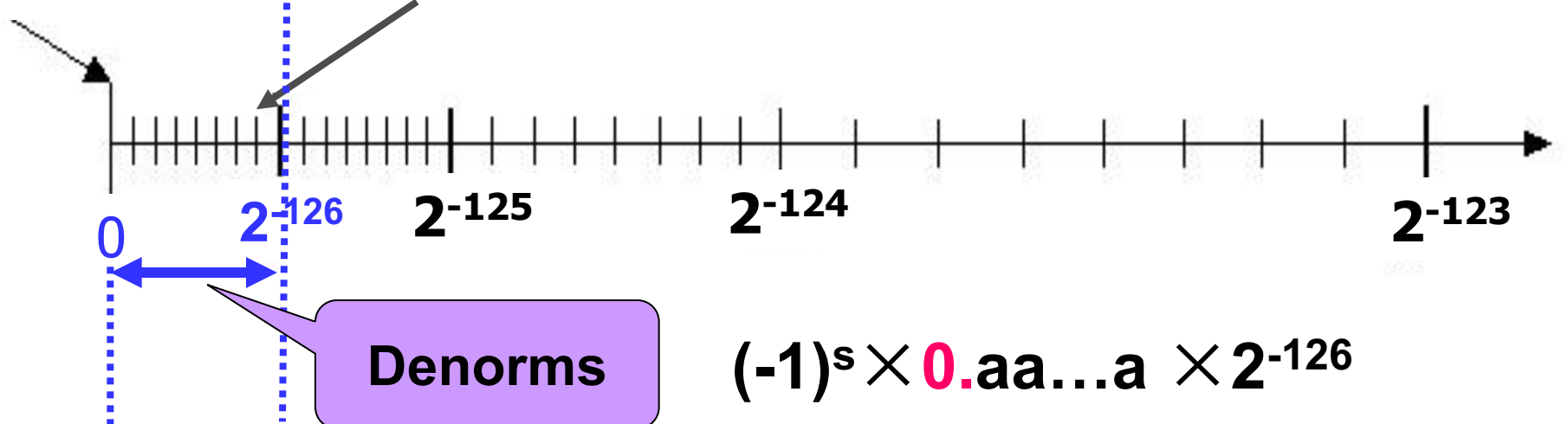


Representation for Denorms

$1.0...0 \times 2^{-126} \sim 1.1...1 \times 2^{-126}$



$0.0...0 \times 2^{-126} \sim 0.1...1 \times 2^{-126}$



$$(-1)^s \times 0.a \dots a \times 2^{-126}$$

Questions about IEEE 754

- ◆ What's the range of representable values?

The largest number for single: $+1.11...1 \times 2^{127}$ 约 $+3.4 \times 10^{38}$

How about double? 约 $+1.8 \times 10^{308}$

- ◆ What about following type converting: not always true!

```
if ( i == (int) ((float) i) ) {  
    printf ("true");  
}
```

How about
double?

True!

```
if ( f == (float) ((int) f) ) {  
    printf ("true");  
}
```

How about
double?

Not always
true!

- ◆ How about FP add associative? FALSE!

$x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, $z = 1.0$

$(x+y)+z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = 1.0$

$x+(y+z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = 0.0$

十进制数的表示

◆ 数值数据（numerical data）的两种表示

Binary (二进制数)

- 定点整数：Fixed-point number (integer)
 - Unsigned and signed int
- 浮点数：Floating-point number (real number)

Decimal (十进制数)

- 用ASCII码表示
- 用BCD（Binary coded Decimal）码表示

◆ 计算机中为什么要用十进制数表示数值？

- 日常使用的都是十进制数，所以，计算机外部都使用十进制数。在一些有大量数据输入/出的系统中，为减少二进制数和十进制数之间的转换，在计算机内部直接用十进制数表示数值。

用ASCII码表示十进制数

◆ 前分隔数字串

- 符号位单独用一个字节表示，位于数字串之前。
- 正号用 “+”的ASCII码(2BH)表示；负号用 “-”的ASCII码(2DH)表示
- 例：十进制数+236表示为: 2B 32 33 36H

0010 1011 0011 0010 0011 0011 0011 0110B

十进制数-2369表示为: 2D 32 33 36 39H

0010 1101 0011 0010 0011 0011 0011 0110 0011 1001B

◆ 后嵌入数字串

- 符号位嵌入最低位数字的ASCII码高4位中。比前分隔方式省一个字节。
- 正数不变；负数高4位变为0111.
- 例：十进制数+236表示为: 32 33 36H

0011 0010 0011 0011 0111 0110B

十进制数-2369表示为: 32 33 36 79H

0011 0010 0011 0011 0011 0110 0111 1001B

缺点：占空间大，且需转换成二进制数或BCD码才能计算。

用BCD码表示十进制数

- ◆ 编码思想： 每个十进数位至少有4位二进制表示。而4位二进制位可组合成16种状态，去掉10种状态后还有6种冗余状态。

- ◆ 编码方案

- 1. 十进制有权码

- 每个十进制数位的4个二进制位（称为基2码）都有一个确定的权。
8421码是最常用的十进制有权码。也称自然BCD（NBCD）码。

- 2. 十进制无权码

- 每个十进制数位的4个基2码没有确定的权。在无权码方案中，用的较多的是余3码和格雷码。

- 3. 其他编码方案（5中取2码、独热码等）

- ◆ 符号位的表示：

- “+”： 1100 ； “-”： 1101

- 例： +236=(1100 0010 0011 0110)₈₄₂₁ （占2个字节）

- 2369=(1101 0000 0010 0011 0110 1001)₈₄₂₁ （占3个字节）

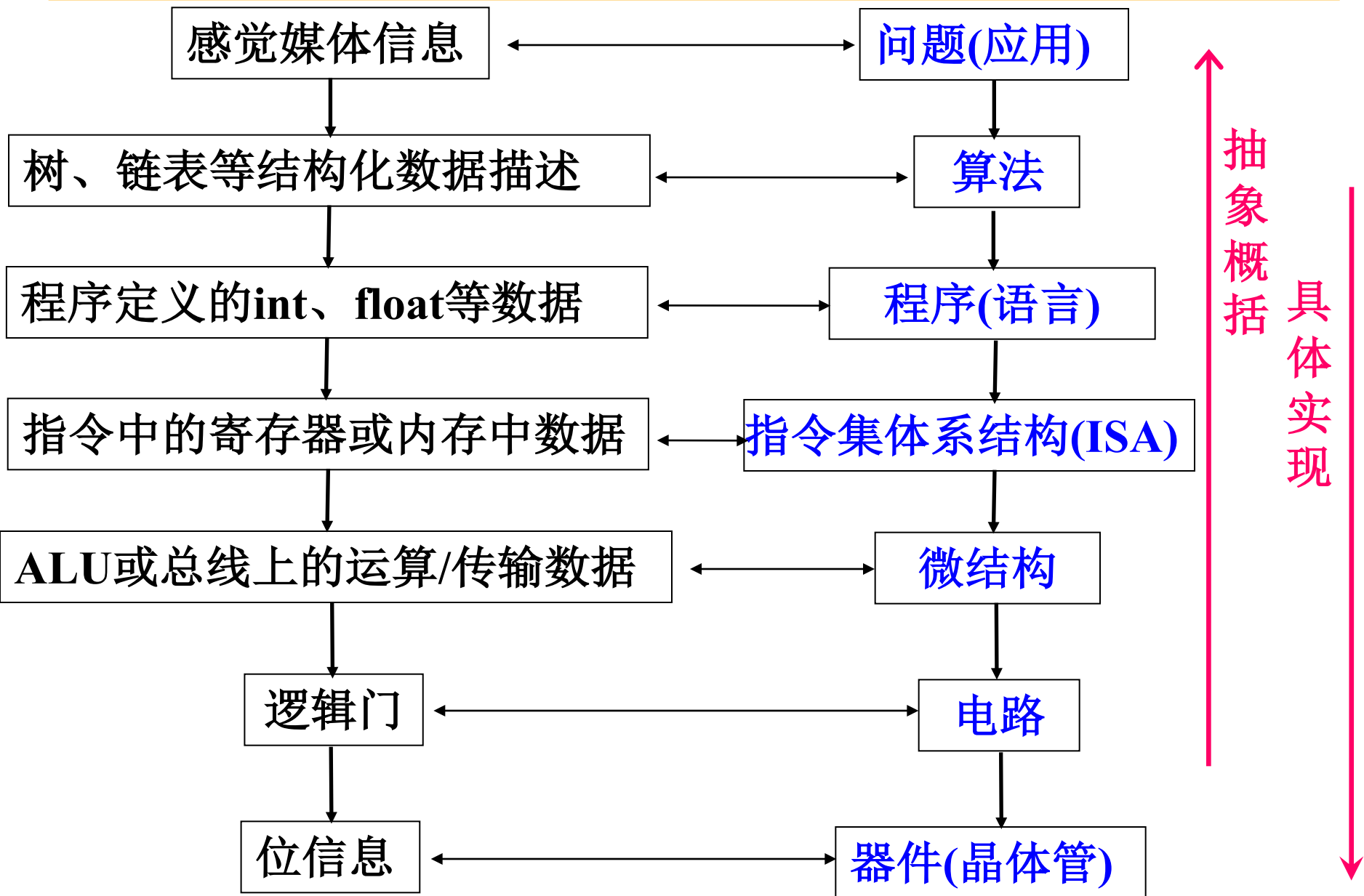
↑
补0以使数占满一个字节

第一讲小结

10在计算机中有几种可能的表示？
-10呢？

- ◆ 在机器内部编码后的数称为机器数，其值称为真值
- ◆ 定义数值数据有三个要素：进制、定点/浮点、编码
- ◆ 整数的表示
 - 无符号数：正整数，用来表示地址等；带符号整数：用补码表示
- ◆ C语言中的整数
 - 无符号数：unsigned int (short / long)；带符号数：int (short / long)
- ◆ 浮点数的表示
 - 符号；尾数：定点小数；指数（阶）：定点整数（基不用表示）
- ◆ 浮点数的范围
 - 正上溢、正下溢、负上溢、负下溢；与阶码的位数和基的大小有关
- ◆ 浮点数的精度：与尾数的位数和是否规格化有关
- ◆ 浮点数的表示（IEEE 754标准）：单精度SP（float）和双精度DP（double）
 - 规格化数(SP)：阶码1~254，尾数最高位隐含为1
 - “零”（阶为全0，尾为全0）
 - ∞ （阶为全1，尾为全0）
 - NaN（阶为全1，尾为非0）
 - 非规格化数（阶为全0，尾为非0，隐藏位为0）（P.42倒数第9行说明）
- ◆ 十进制数的表示：用ASCII码或BCD码表示

“转换”的概念在数据表示中的反映



第二讲 非数值数据、数据排列、纠/检错

主 要 内 容

◆非数值数据的表示

- 逻辑数据、西文字符、汉字

◆数据的宽度

◆数据的存储排列

- 大端方式、小端方式

◆数据的纠错和检错

- 奇偶校验、海明校验、循环冗余校验

逻辑数据的编码表示

◆表示

- 用一位表示。例如，真：1 / 假：0
- N位二进制数可表示N个逻辑数据，或一个位串

◆运算

- 按位进行
- 如:按位与 / 按位或 / 逻辑左移 / 逻辑右移 等

◆识别

- 逻辑数据和数值数据在形式上并无差别，也是一串0/1序列，机器靠指令来识别。

◆位串

- 用来表示若干个状态位或控制位（OS中使用较多）

例如，x86的标志寄存器含义如下：

				OF	DF	IF	TF	SF	ZF		AF		PF		CF
--	--	--	--	----	----	----	----	----	----	--	----	--	----	--	----

西文字符的编码表示

◆特点

- 是一种拼音文字，用有限几个字母可拼写出所有单词
- 只对有限个字母和数学符号、标点符号等辅助字符编码
- 所有字符总数不超过256个，使用7或8个二进位可表示

◆表示（常用编码为7位ASCII码）

- 十进制数字：0/1/2.../9
- 英文字母：A/B/.../Z/a/b/.../z
- 专用符号：+/-/%/*/&/.....
- 控制字符（不可打印或显示）

} 必须熟悉对应的ASCII码！

◆操作

- 字符串操作，如：传送/比较 等

汉字及国际字符的编码表示

◆特点

- 汉字是表意文字，一个字就是一个方块图形。
- 汉字数量巨大，总数超过6万字，给汉字在计算机内部的表示、汉字的传输与交换、汉字的输入和输出等带来了一系列问题。

◆编码形式

- 有以下几种汉字代码：
 - 输入码：对汉字用相应按键进行编码表示，用于输入
 - 内码：用于在系统中进行存储、查找、传送等处理
 - 字模点阵或轮廓描述：描述汉字字模点阵或轮廓，用于显示/打印

问题：西文字符有没有输入码？有没有内码？
有没有字模点阵或轮廓描述？

汉字的输入码

向计算机输入汉字的方式：

- ① 手写汉字联机识别输入，或者是印刷汉字扫描输入后自动识别，这两种方法现均已达到实用水平。
- ② 用语音输入汉字，虽然简单易操作，但离实用阶段还相差很远。
- ③ 利用英文键盘输入汉字：每个汉字用一个或几个键表示，这种对每个汉字用相应按键进行的编码称为汉字“输入码”，又称外码。输入码的码元为按键。是最简便、最广泛的汉字输入方法。

常用的方法有：搜狗拼音、五笔字型、智能ABC、微软拼音等

使用汉字输入码的原因：

- ① 键盘面向西文设计，一个或两个西文字符对应一个按键，非常方便。
- ② 汉字是大字符集，专门的汉字输入键盘由于键多、查找不便、成本高等原因而几乎无法采用。

字符集与汉字的内码

问题：西文字符常用的内码是什么？ **其内码就是ASCII码。**

对于汉字内码的选择，必须考虑以下几个因素：

- ① 不能有二义性，即不能和ASCII码有相同的编码。
- ② 尽量与汉字在字库中的位置有关，便于汉字查找和处理。
- ③ 编码应尽量短。

国标码（国标交换码）

1981年我国颁布了《信息交换用汉字编码字符集·基本集》(GB2312—80)。该标准选出6763个常用汉字，为每个汉字规定了标准代码，以供汉字信息在不同计算机系统间交换使用

可在汉字国标码的基础上产生汉字机内码

GB2312-80字符集

◆由三部分组成：

- ① 字母、数字和各种符号，包括英文、俄文、日文平假名与片假名、罗马字母、汉语拼音等共687个
- ② 一级常用汉字，共3755个，按汉语拼音排列
- ③ 二级常用汉字，共3008个，不太常用，按偏旁部首排列

◆汉字的区位码

- 码表由94行、94列组成，行号为区号，列号为位号，各占7位
- 指出汉字在码表中的位置，共14位，区号在左、位号在右

◆汉字的国标码

- 每个汉字的区号和位号各自加上32（20H），得到其“国标码”
- 国标码中区号和位号各占7位。在计算机内部，为方便处理与存储，前面添一个0，构成一个字节

汉字内码

◆至少需2个字节才能表示一个汉字内码。为什么？

•由汉字的总数决定！

◆可在GB2312国标码的基础上产生汉字内码

•为与ASCII码区别，将国标码的两个字节的第一位置“1”后得到一种汉字内码 区位码→国标码→内码

例如，汉字“大”在码表中位于第20行、第83列。因此区位码为0010100 1010011，国标码为00110100 01110011，即3473H。前面的34H和字符“4”的ASCII码相同，后面的73H和字符“s”的ASCII码相同，将每个字节的最高位各设为“1”后，就得到其内码：B4F3H (1011 0100 1111 0011B)，因而不会和ASCII码混淆。

国际字符集

国际字符集的必要性

- ◆ 不同地区使用不同字符集内码，如中文**GB2312 / Big5**、日文**Shift-JIS / EUC-JP**等。在安装中文系统的计算机中打开日文文件，会出现乱码。
- ◆ 为使所有国际字符都能互换，必须创建一种涵盖全部字符的多字符集。

国际多字符集

- ◆ 通过对各种地区性字符集规定使用范围来唯一定义各字符的编码。
- ◆ 国际标准**ISO/IEC 10646**提出了一种包括全世界现代书面语言文字所使用的所有字符的标准编码，有4个字节编码(**UCS-4**)和2字节编码(**UCS-2**)。
- ◆ 我国（包括香港、台湾地区）与日本、韩国联合制订了一个统一的汉字字符集（**CJK编码**），共收集了上述不同国家和地区共约2万多汉字及符号，采用2字节编码（即：**UCS-2**），已被批准为国家标准(**GB13000**)。
- ◆ Windows操作系统(中文版)已采用中西文统一编码，收集了中、日、韩三国常用的约2万汉字，称为“**Unicode**”，采用2字节编码，与**UCS-2**一致。

汉字的字模点阵码和轮廓描述

- ◆ 为便于打印、显示汉字，汉字字形必须预先存在机内
 - 字库 (font): 所有汉字形状的描述信息集合
 - 不同字体 (如宋体、仿宋、楷体、黑体等) 对应不同字库
 - 从字库中找到字形描述信息，然后送设备输出

问题：如何知道到哪里找相应的字形信息？

汉字内码与其在字库中的位置有关！！

- ◆ 字形主要有两种描述方法：**区位码←国标码←内码**
 - 字模点阵描述（图像方式）
 - 轮廓描述（图形方式）
 - 直线向量轮廓
 - 曲线轮廓（True Type字形）

数据的基本宽度

- ◆ 比特 (bit) 是计算机中处理、存储、传输信息的最小单位
- ◆ 二进制信息的计量单位是“字节” (Byte), 也称“位组”
 - 现代计算机中, 存储器按字节编址
 - 字节是最小可寻址单位 (*addressable unit*)
 - 如果以字节为一个排列单位, 则LSB表示最低有效字节, MSB表示最高有效字节
- ◆ 除比特和字节外, 还经常使用“字” (word)作为单位
- ◆ “字” 和 “字长” 的概念不同
 - IA-32中的“字” 有多少位? 字长多少位呢?
 - DWORD : 32位 16位 32位
 - QWORD: 64位

数据的基本宽度

◆ “字” 和 “字长” 的概念不同

- “字长” 指定点运算数据通路的宽度。

（数据通路指CPU内部数据流经的路径以及路径上的部件，主要是CPU内部进行数据运算、存储和传送的部件，这些部件的宽度基本上要一致，才能相互匹配。因此，“字长”等于CPU内部总线的宽度、运算器的位数、通用寄存器的宽度等。）

- “字” 表示被处理信息的单位，用来度量数据类型的宽度。
- 字和字长的宽度可以一样，也可不同。

例如，x86体系结构定义“字”的宽度为16位，但从386开始字长就是32位了。

数据量的度量单位

- ◆ 存储二进制信息时的度量单位要比字节或字大得多
- ◆ 容量经常使用的单位有：
 - “千字节” (KB), $1\text{KB}=2^{10}\text{字节}=1024\text{B}$
 - “兆字节” (MB), $1\text{MB}=2^{20}\text{字节}=1024\text{KB}$
 - “千兆字节” (GB), $1\text{GB}=2^{30}\text{字节}=1024\text{MB}$
 - “兆兆字节” (TB), $1\text{TB}=2^{40}\text{字节}=1024\text{GB}$
- ◆ 通信中的带宽使用的单位有：
 - “千比特/秒” (kb/s), $1\text{kbps}=10^3\text{ b/s}=1000\text{ bps}$
 - “兆比特/秒” (Mb/s), $1\text{Mbps}=10^6\text{ b/s}=1000\text{ kbps}$
 - “千兆比特/秒” (Gb/s), $1\text{Gbps}=10^9\text{ b/s}=1000\text{ Mbps}$
 - “兆兆比特/秒” (Tb/s), $1\text{Tbps}=10^{12}\text{ b/s}=1000\text{ Gbps}$

如果把b换成B，则表示字节而不是比特（位）

例如，10MBps表示 10兆字节/秒

程序中数据类型的宽度

- ◆ 高级语言支持多种类型、多种长度的数据

- 例如，C语言中char类型的宽度为1个字节，可表示一个字符（非数值数据），也可表示一个8位的整数（数值数据）
- 不同机器上表示的同一种类型的数据可能宽度不同

- ◆ 程序中的数据有相应的机器级表示方式和相应的处理指令

(在第五章指令系统介绍具体指令)

从表中看出：同类型数据并不是所有机器都采用相同的宽度，分配的字节数随机器字长和编译器的不同而不同。

C语言中数值数据类型的宽度 (单位：字节)

C声明	典型32位机器	Compaq Alpha机器
char	1	1
short int	2	2
int	4	4
long int	4	8
char*	4	8
float	4	4
double	8	8

Compaq Alpha是一个针对高端应用的64位机器，即字长为64位

数据的存储和排列顺序

$$65535 = 2^{16} - 1$$

◆ 80年代开始，几乎所有通用机器都用**字节编址** $[-65535]_{\text{补}} = \text{FFFF0001H}$

◆ ISA设计时要考虑的两个问题：

- 如何根据一个地址取到一个32位的字？ - 字的存放问题
- 一个字能否存放在任何地址边界？ - 字的边界对齐问题

例如，若 $\text{int } i = -65535$ ，存放在内存100号单元（即占100# ~ 103#），则用“取数”指令访问100号单元取出 i 时，必须清楚 i 的4个字节是如何存放的。

Word:	FF	FF	00	01	little endian word 100#
	103	102	101	100	
	msb			lsb	
	100	101	102	103	big endian word 100#

大端方式（**Big Endian**）：**MSB**所在的地址是数的地址

e.g. IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

小端方式（**Little Endian**）：**LSB**所在的地址是数的地址

e.g. Intel 80x86, DEC VAX

有些机器两种方式都支持，可通过特定控制位来设定采用哪种方式。

BIG Endian versus Little Endian

Ex1: Memory layout of a number ABCDH located in 1000

In Big Endian:	→	CD	1001
		AB	1000
In Little Endian:	→	AB	1001
		CD	1000

Ex2: Memory layout of a number 00ABCDEFH located in 1000

In Big Endian:	→	00	1000
		AB	1001
		CD	1002
		EF	1003
In Little Endian:	→	00	1003
		AB	1002
		CD	1001
		EF	1000

BIG Endian versus Little Endian

Ex3: Memory layout of a instruction located in 1000

假定小端机器中指令: `mov AX, 0x12345(BX)`

其中操作码mov为40H, 寄存器AX和BX的编号分别为0001B和0010B, 立即数占32位, 则存放顺序为:



若在大端机器上, 则存放顺序如何?



00	1005	45
01	1004	23
23	1003	01
45	1002	00
12	1001	12
40	1000	40

地址

只需要考虑指令中立即数的顺序!

Byte Swap Problem (字节交换问题)

78	3
56	2
34	1
12	0

Big Endian

↑
increasing
byte
address

12	3
34	2
56	1
78	0

Little Endian

上述存放在0号单元的数据（字）是什么？ 12345678H? 78563412H?

存放方式不同的机器间程序移植或数据通信时，会发生什么问题？

- ◆ 每个系统内部是一致的，但在系统间通信时可能会发生问题！
- ◆ 因为顺序不同，需要进行顺序转换

音、视频和图像等文件格式或处理程序都涉及到字节顺序问题

ex. Little endian: GIF, PC Paintbrush, Microsoft RTF, etc

Big endian: Adobe Photoshop, JPEG, MacPaint, etc

Alignment(对齐)

Alignment: 要求数据的地址是相应的边界地址

- ◆ 目前机器字长一般为32位或64位，而存储器地址按字节编址
- ◆ 指令系统支持对字节、半字、字及双字的运算，也有位处理指令
- ◆ 各种不同长度的数据存放时，有两种处理方式：

- 按边界对齐（假定存储字的宽度为32位，按字节编址）

- 字地址：4的倍数（低两位为0）

- 半字地址：2的倍数（低位为0）

- 字节地址：任意

每4个字节可同时读写



- 不按边界对齐

坏处：可能会增加访存次数！

（学了第四章存储器组织后会更明白！）

Alignment(对齐)

如: `int i, short k, double x, char c, short j,.....`

存储器按字节

编址

每次只能读写

某个字地址开

始的4个单元中

连续的1个、2

个、3个或4个

字节

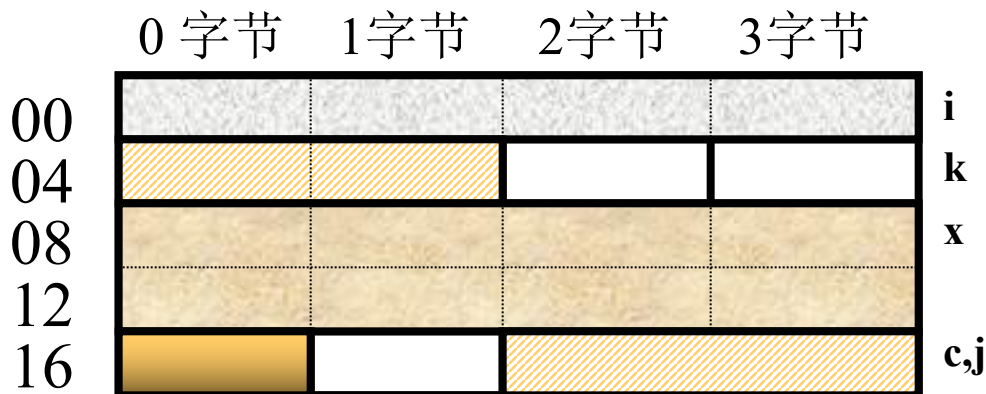
虽节省了空间，
但增加了访存次
数！

需要权衡，目前
来看，浪费一点
存储空间没有关
系！

按边界对齐

x: 2个周期

j: 1个周期

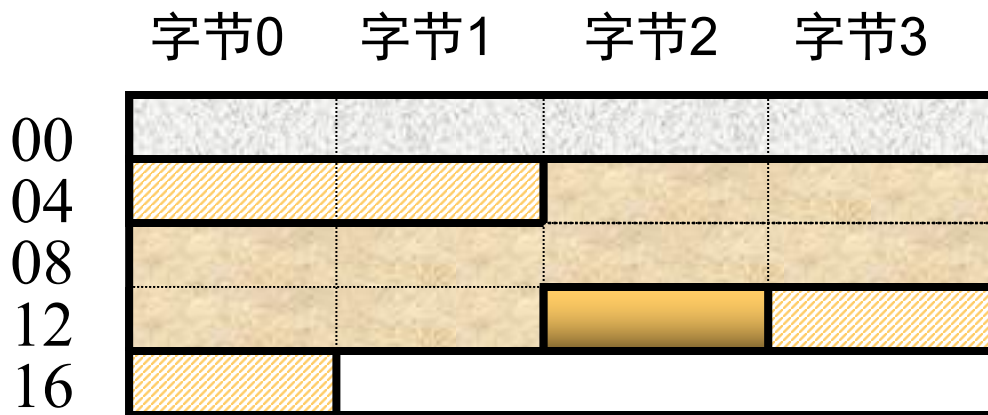


则: `&i=0; &k=4; &x=8; &c=16; &j=18;.....`

边界不对齐

x: 3个周期

j: 2个周期



则: `&i=0; &k=4; &x=6; &c=14; &j=15;.....`

Alignment(对齐) 举例

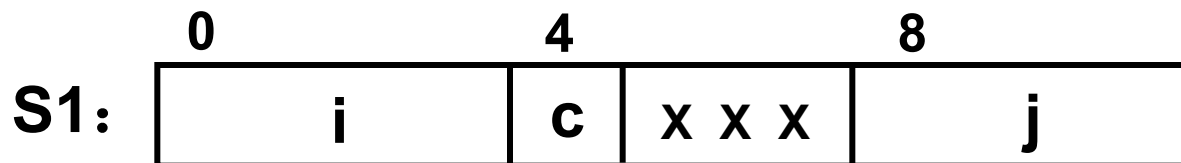
例如，考虑下列两个结构声明：

```
struct S1 {  
    int    i;  
    char   c;  
    int    j;  
};
```

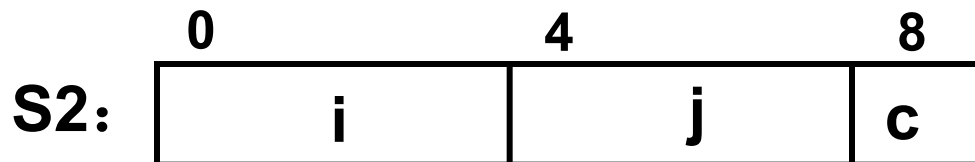
```
struct S2 {  
    int    i;  
    int    j;  
    char   c;  
};
```

在要求对齐的情况下，哪种结构声明更好？

S2比S1好

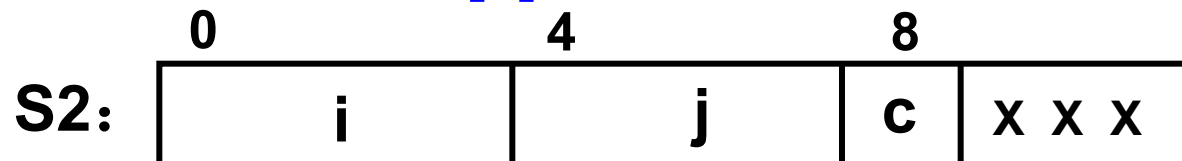


需要12个字节



只需要9个字节

对于 “struct S2 d[4]”，只分配9个字节能否满足对齐要求？ **不能！**



也需要12个字节

数据的检/纠错 (Error Detect/Correct)

◆ 为什么要进行数据的错误检测与校正？

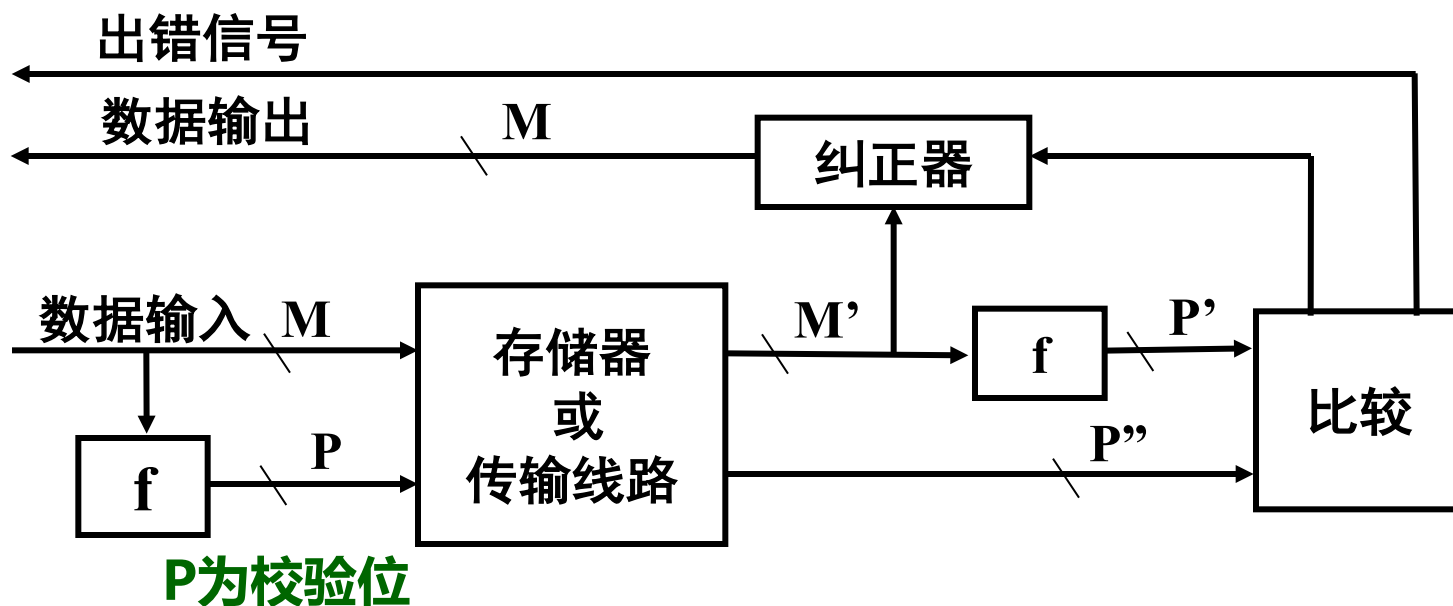
存取和传送时，由于元器件故障或噪音干扰等原因会出现差错。措施：

(1) 从计算机硬件本身的可靠性入手，在电路、电源、布线等各方面采取必要的措施，提高计算机的抗干扰能力；

(2) 采取相应的数据检错和校正措施，自动地发现并纠正错误。

◆ 如何进行错误检测与校正？

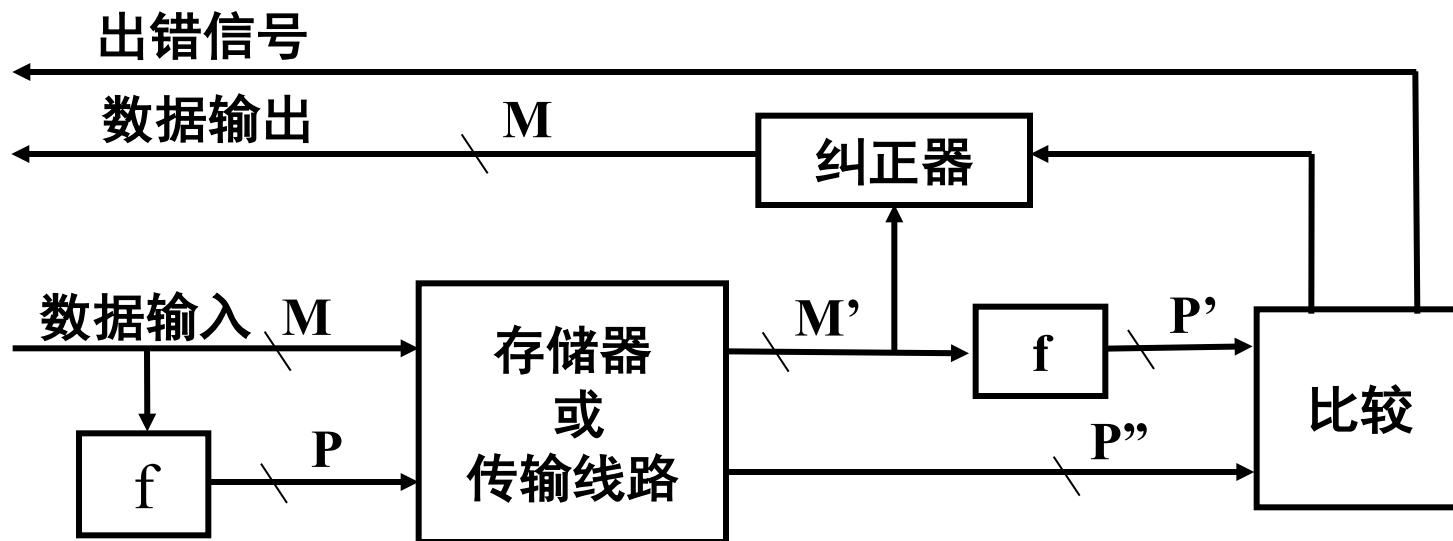
- 大多采用“冗余校验”思想，即除原数据信息外，还增加若干位编码，这些新增的代码被称为校验位。



数据的检/纠错

比较的结果为以下三种情况之一：

- ① 没有检测到错误，得到的数据位直接传送出去。
- ② 检测到差错，并可以纠错。数据位和比较结果一起送入纠错器，将正确数据位传送出去。
- ③ 检测到错误，但无法确认哪位出错，因而不能进行纠错处理，此时，报告出错情况。



码字和码距

◆ 什么叫码距？

- 由若干位代码组成的一个字叫“码字”
- 两个码字中具有不同代码的位的个数叫这两个码字间的“距离”
- 码制中各码字间最小距离为“码距”，它就是这个码制的距离。

问题：“8421”码的码距是几？

2 (0010) 和 3 (0011) 间距离为1，“8421”码制的码距为1。

◆ 数据校验中“码字”指数据位和校验位按某种规律排列得到的代码

◆ 码距与检错、纠错能力的关系（当 $d \leq 4$ ）

① 若码距 d 为奇数，则能发现 $d-1$ 位错，或能纠正 $(d-1)/2$ 位错。

② 若码距 d 为偶数，则能发现 $d/2$ 位错，并能纠正 $(d/2-1)$ 位错。

◆ 常用的数据校验码有：

奇偶校验码、海明校验码、循环冗余校验码。

奇偶校验码

基本思想：增加一位奇（偶）校验位并一起存储或传送，根据终部件得到的相应数据和校验位，再求出新校验位，最后根据新校验位确定是否发生了错误。

实现原理：假设数据 $B=b_{n-1}b_{n-2}\dots b_1b_0$ 从源部件传送至终部件。在终部件接收到的数据为 $B'=b_{n-1}'b_{n-2}'\dots b_1'b_0'$ 。

第一步：在源部件求出奇（偶）校验位 P 。

若采用奇校验，则 $P=b_{n-1}\oplus b_{n-2}\oplus\dots\oplus b_1\oplus b_0\oplus 1$ 。

若采用偶校验，则 $P=b_{n-1}\oplus b_{n-2}\oplus\dots\oplus b_1\oplus b_0$ 。

第二步：在终部件求出奇（偶）校验位 P' 。

若采用奇校验，则 $P'=b_{n-1}'\oplus b_{n-2}'\oplus\dots\oplus b_1'\oplus b_0'\oplus 1$ 。

若采用偶校验，则 $P'=b_{n-1}'\oplus b_{n-2}'\oplus\dots\oplus b_1'\oplus b_0'$ 。

第三步：计算最终的校验位 P^* ，并根据其值判断有无奇偶错。

假定 P 在终部件接受到的值为 P'' ，则 $P^*=P'\oplus P''$

① 若 $P^*=1$ ，则表示终部件接受的数据有奇数位错。

② 若 $P^*=0$ ，则表示终部件接受的数据正确或有偶数个错。

奇偶校验码

- 编码方法
 - 不管数据位长度多少，校验位只有一位。
 - 数据位和校验位一起所含“1”的个数，只能是奇数，称为奇校验。
 - 数据位和校验位一起所含“1”的个数，只能是偶数，称为偶校验。
 - 例：数据 奇校验的编码 偶校验的编码

00000000	100000000	000000000
01010100	001010100	101010100
01111111	001111111	101111111

缺点: 这种方案只能发现一位错或奇数个位错，但不能确定是哪一位错，也不能发现偶数个位错。

优点: 该方案还是有很好的实用价值。

奇偶校验法的特点

◆ 问题：奇偶校验码的码距是几？为什么？

- 码距 $d=2$ 。

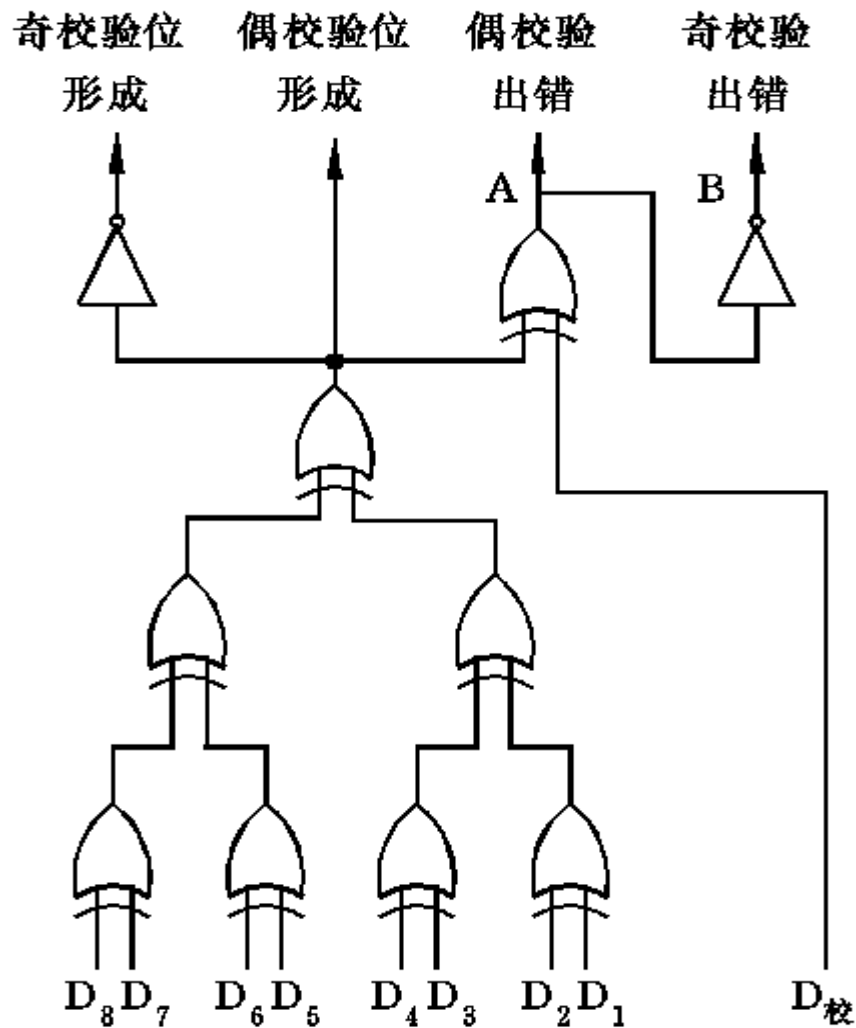
在奇偶校验码中，若两个数中有奇数位不同，则它们相应的校验位就不同；若有偶数位不同，则虽校验位相同，但至少有两数据位不同。因而任意两个码字之间至少有两位不同。

◆ 特点

- 根据码距和纠/检错能力的关系，它只能发现奇数位出错，不能发现偶数位出错，而且也不能确定发生错误的位置，不具有纠错能力。
- 开销小，适用于校验一字节长的代码，故常被用于存储器读写检查或按字节传输过程中的数据校验

因为一字节长的代码发生错误时，1位出错的概率较大，两位以上出错则很少，所以可用奇偶校验。

奇偶校验位的形成及校验



奇偶校验位的应用

在实际工作中还经常采用纵横都加校验奇偶校验位的编码系统--**分组奇偶校验码**。

现在考虑一个系统，它传输若干个长度为 m 位的信息。如果把这些信息都编成每组 n 个信息的分组，则在这些不同的信息间，也如对单个信息一样，能够作奇偶校验。图4中 n 个信息的一个分组排列成矩形式样，并以横向奇偶（HP）及纵向奇偶（VP）的形式编出奇偶校验位。

m位数字					横向奇偶位
a_1	a_2	...	a_{m-1}	a_m	HP_1
b_1	b_2	...	b_{m-1}	b_m	HP_2
c_1	c_2	...	c_{m-1}	c_m	HP_3
...
n_1	n_2	...	n_{m-1}	n_m	HP_n
VP_1	VP_2	...	VP_{m-1}	VP_m	HP_{n+1}

n个码字

纵向奇偶位

图 4 用纵横奇偶校验的分组奇偶校验码

奇偶校验位的应用

【例】 初级程序员试题

由 6 个字符的 7 位 ASCII 编码排列，再加上水平垂直奇偶校验位构成下列矩阵（最后一列为水平奇偶校验位，最后一行为垂直奇偶校验位）：

字符	7 位 ASCII 码							HP
3	0	X_1	X_2	0	0	1	1	0
Y_1	1	0	0	1	0	0	X_3	1
+	X_4	1	0	1	0	1	1	0
Y_2	0	1	X_5	X_6	1	1	1	1
D	1	0	0	X_7	1	0	X_8	0
=	0	X_9	1	1	1	X_{10}	1	1
VP	0	0	1	1	1	X_{11}	1	X_{12}

则 $X_1 X_2 X_3 X_4$ 处的比特分别为 __ (36) __ ；

$X_5 X_6 X_7 X_8$ 处的比特分别为 ____ ；

$X_9 X_{10} X_{11} X_{12}$ 处的比特分别为 __ (38) __ ； Y_1 和 Y_2 处的字符分别为 __ (39) __ 和 __ (40) __ 。

奇偶校验位的应用

[解]

从ASCII码左起第5列可知垂直为偶校验。则：

从第1列可知 $X_4=0$ ；从第3行可知水平也是偶校验。

从第2行可知 $X_3=1$ ；从第7列可知 $X_8=0$ ；从第8列可知 $X_{12}=1$ ；

从第7行可知 $X_{11}=1$ ；从第6列可知 $X_{10}=0$ ；从第6行可知 $X_9=1$ ；从第2列可知 $X_1=1$ ；

从第1行可知 $X_2=1$ ；从第3列可知 $X_5=1$ ；从第4行可知 $X_6=0$ ；

从第4列（或第5行）可知 $X_7=0$ ；整理一下：

(36) $X_1X_2X_3X_4 = 1110$

(37) $X_5X_6X_7X_8 = 1000$

(38) $X_9X_{10}X_{11}X_{12} = 1011$

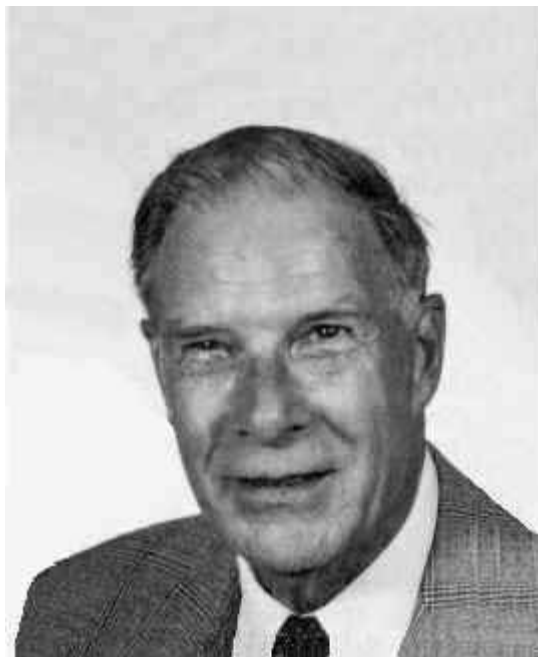
(39) 由字符 Y_1 的ASCII码1001001=49H知道， Y_1 即是“1”（由“D”的ASCII码是1000100=44H推得）

(40) 由字符 Y_2 的ASCII码0110111=37H知道， Y_2 即是“7”（由“3”的ASCII码是0110011=33H推得）

假如你能记住“0”的ASCII码是0110000=30H；“A”的ASCII码是1000001=41H，则解起来就更方便了。

海明校验码

- ◆ 由Richard Hamming于1950年提出，目前还被广泛使用。
 - ◆ 主要用于存储器中数据存取校验。
 - ◆ 基本思想：奇偶校验码对整个数据编码生成一位校验位。因此这种校验码检错能力差，并且没有纠错能力。如果将整个数据按某种规律分成若干组，对每组进行相应的奇偶检测，就能提供多位检错信息，从而对错误位置进行定位，并将其纠正。
 - 海明校验码实质上就是一种多重奇偶校验码。
 - ◆ 处理过程：
 - 最终比较时按位进行异或，以确定是否有差错。
 - 这种异或操作所得到的结果称为故障字（syndrome word）。显然，校验码和故障字的位数是相同。
- 每一组一个校验位，校验码位数等于组数！
- 每一组内采用一位奇偶校验！



Richard Wesley Hamming (02/11/1915—01/07/1998)

图灵奖获得时间：

1968年。第三位图灵奖获得者。

图灵奖引用 (Turing Award Citation)：

Citation : For his work on numerical methods, automatic coding systems, and error-detecting and error-correcting codes.

校验码位数的确定

- ◆ 假定数据位数为 n ，校验码为 k 位，则故障字位数也为 k 位。 k 位故障字所能表示的状态最多是 2^k ，每种状态可用来说明一种出错情况。
- ◆ 若只有一位错，则结果可能是：
 - 数据中某一位错 (n 种可能)
 - 校验码中有一位错 (k 种可能)
 - 无错 (1种可能)

1+n+k种情况

假定最多有一位错，则 n 和 k 必须满足下列关系：

$$2^k \geq 1+n+k, \quad \text{即: } 2^k - 1 \geq n+k$$

- 有效数据位数和校验码位数间的关系
- 当数据有8位时，校验码和故障字都应有4位。

说明：4位故障字最多可表示16种状态，而单个位出错情况最多只有12种可能（8个数据位和4个校验位），再加上无错的情况，一共有13种。所以，用16种状态表示13种情况应是足够了。

有效数据位数和校验码位数间的关系

n和k的关系： $2^k-1 \geq n+k$ （n和k分别为数据位数和校验位数）

	单纠错		单纠错/双检错	
数据位数	校验位数	增加率	校验位数	增加率
8	4	50	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

海明码的分组

- ◆ **基本思想：** n 位数据位和 k 位校验位按某种方式排列为一个（ $n+k$ ）位的码字，将该码字中每个出错位的位置与故障字的数值建立关系，通过故障字的值确定该码字中哪一位发生了错误，并将其取反来纠正。

根据上述基本思想，按以下规则来解释各故障字的值。

规则1： 若故障字每位全部是0，则表示没有发生错误。

规则2： 若故障字中有且仅有一位为1，则表示校验位中有一位出错，因而不需纠正。

规则3： 若故障字中多位为1，则表示有一个数据位出错，其在码字中的出错位置由故障字的数值来确定。纠正时只要将出错位取反即可。

海明码的分组

以8位数据进行单个位检错和纠错为例说明。

假定8位数据 $M = M_8M_7M_6M_5M_4M_3M_2M_1$ ，4位校验位 $P = P_4P_3P_2P_1$ 。根据规则将M和P按一定的规律排到一个12位码字中。

据规则1，故障字为0000时，表示无错。

据规则2，故障字中有且仅有一位为1时，表示校验位中有一位出错。此时，故障字只可能是0001、0010、0100、1000，将这四种状态分别代表校验位中 P_1 、 P_2 、 P_3 、 P_4 位发生错误，因此，它们分别位于码字的第1、2、4、8位。

据规则3，将其他多位为1的故障字依次表示数据位 $M_1 \sim M_8$ 发生错误的情况。因此，数据位 $M_1 \sim M_8$ 分别位于码字的第0、0111(7)、1001(9)、1010(10)、1011(11)、
是逻辑顺序，物理上
M和P是分开的！

列为： $M_8M_7M_6M_5P_4M_4M_3M_2P_3M_1P_2P_1$

这样，得到故障字 $S = S_4S_3S_2S_1$ 的各个状态和出错情况的对应关系表，可根据这种对应关系对整个数据进行分组。

海明校验码分组情况

故障字 $S_4S_3S_2S_1$ 每一位的值
反映所在组的奇偶性

码字: $M_8M_7M_6M_5P_4M_4M_3M_2P_3M_1P_2P_1$

序号 含义 分组	1	2	3	4	5	6	7	8	9	10	11	12	故障字	正 确	出错位											
	P ₁	P ₂	M ₁	P ₃	M ₂	M ₃	M ₄	P ₄	M ₅	M ₆	M ₇	M ₈			1	2	3	4	5	6	7	8	9	10	11	12
第4组								√	√	√	√	√	S ₄	0	0	0	0	0	0	0	1	1	1	1	1	
第3组				√	√	√	√					√	S ₃	0	0	0	1	1	1	1	0	0	0	0	1	
第2组		√	√			√	√			√	√		S ₂	0	0	1	1	0	0	1	1	0	0	1	1	
第1组	√		√		√		√		√		√		S ₁	0	1	0	1	0	1	0	1	0	1	0	1	

数据位或校验位出错一定会影响所在组的奇偶性。

例：若 M_2 出错，则故障字为0101，因而会改变 S_3 和 S_1 所在分组的奇偶性。
故 M_2 同时被分到第3(S_3)组和第1(S_1)组。

问题：若 P_1 出错，则如何？若 M_8 出错，则如何？

SKIP

$P_1 \sim 0001$ ，分在第1组； $M_8 \sim 1100$ ，分在第4组和第3组

校验位的生成和检错、纠错

- ◆ 分组完成后，就可对每组采用相应的奇（偶）校验，以得到相应的一个校验位。
- ◆ 假定采用偶校验（取校验位 P_i ，使对应组中有偶数个1），则得到校验位与数据位之间存在如下关系：

$$P_4 = M_5 \oplus M_6 \oplus M_7 \oplus M_8$$

$$P_3 = M_2 \oplus M_3 \oplus M_4 \oplus M_8$$

$$P_2 = M_1 \oplus M_3 \oplus M_4 \oplus M_6 \oplus M_7$$

$$P_1 = M_1 \oplus M_2 \oplus M_4 \oplus M_5 \oplus M_7$$

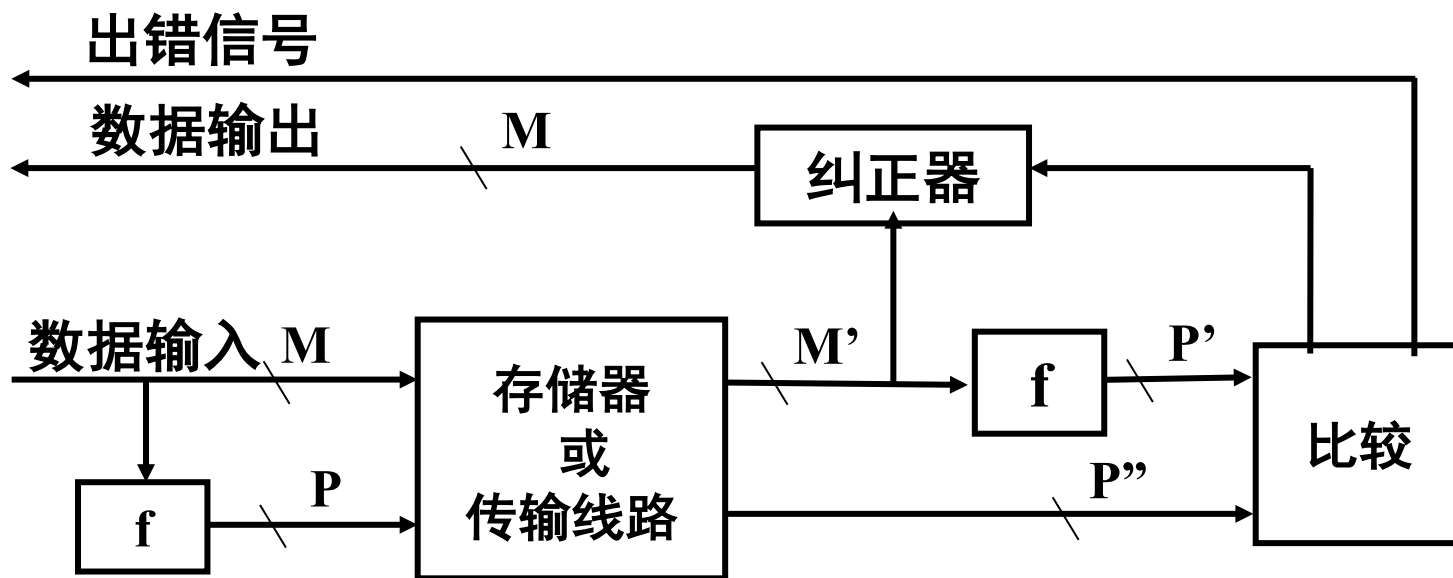
码字： $M_8 M_7 M_6 M_5 P_4 M_4 M_3 M_2 P_3 M_1 P_2 P_1$

海明校验过程

- ◆ 根据公式求出每一组对应的校验位 P_i ($i=1,2,3,4$)
- ◆ 数据 M 和校验位 P 一起被存储，根据读出数据 M' 得新校验位 P'
- ◆ 读出校验位 P'' 与新校验位 P' 按位进行异或操作，得故障字

$$S = S_4 S_3 S_2 S_1$$

- ◆ 根据 S 的值确定：无错、仅校验位错、某个数据位错



海明码举例

假定一个8位数据M为： $M_8M_7M_6M_5M_4M_3M_2M_1=01101010$ ，根据上述公式求出相应的校验位为：

$$P_4 = M_5 \oplus M_6 \oplus M_7 \oplus M_8 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$

$$P_3 = M_2 \oplus M_3 \oplus M_4 \oplus M_8 = 1 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_2 = M_1 \oplus M_3 \oplus M_4 \oplus M_6 \oplus M_7 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$$

$$P_1 = M_1 \oplus M_2 \oplus M_4 \oplus M_5 \oplus M_7 = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 1$$

假定12位码字 ($M_8M_7M_6M_5P_4M_4M_3M_2P_3M_1P_2P_1$) 读出后为：

(1) 数据位 $M'=M=01101010$ ，校验位 $P''=P=0011$

(2) 数据位 $M'=011\textcolor{red}{1}1010$ ，校验位 $P''=P=0011$

(3) 数据位 $M'=M=01101010$ ，校验位 $P''=\textcolor{red}{1}011$

要求分别考察每种情况的故障字。

(1) 数据位 $M'=M=01101010$ ，校验位 $P''=P=0011$ ，即无错。

因为 $M'=M$ ，所以 $P'=P$ ，因此 $S = P'' \oplus P' = P \oplus P = 0000$ 。

海明码举例

(2) 数据位M'= 011**1**1010, 校验位P''=P=0011, 即M₅错。

对M'生成新的校验位P'为:

$$P_4' = M_5' \oplus M_6' \oplus M_7' \oplus M_8' = 1 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$P_3' = M_2' \oplus M_3' \oplus M_4' \oplus M_8' = 1 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_2' = M_1' \oplus M_3' \oplus M_4' \oplus M_6' \oplus M_7' = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$$

$$P_1' = M_1' \oplus M_2' \oplus M_4' \oplus M_5' \oplus M_7' = 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 = 0$$

故障字S为:

$$S_4 = P_4' \oplus P_4'' = 1 \oplus 0 = 1$$

$$S_3 = P_3' \oplus P_3'' = 0 \oplus 0 = 0$$

$$S_2 = P_2' \oplus P_2'' = 1 \oplus 1 = 0$$

$$S_1 = P_1' \oplus P_1'' = 0 \oplus 1 = 1$$

因此, 错误位是第9位, 排列的是数据位M₅, 所以检错正确, 纠错时, 只要将码字的第9位 (M₅) 取反即可。

海明码举例

(3) 数据位 $M'=M=01101010$, 校验位 $P''=1011$,

即: 校验码第4位(P_4)错。

因为 $M'=M$, 所以 $P'=P$, 因此故障位 S 为:

$$S_4 = P_4' \oplus P_4'' = 0 \oplus 1 = 1$$

$$S_3 = P_3' \oplus P_3'' = 0 \oplus 0 = 0$$

$$S_2 = P_2' \oplus P_2'' = 1 \oplus 1 = 0$$

$$S_1 = P_1' \oplus P_1'' = 1 \oplus 1 = 0$$

错误位是第1000位(即第8位), 这位上排列的是校验位 P_4 , 所以检错时发现数据正确, 不需纠错。

单纠错和双检错码

◆ 单纠错码 (SEC -Single Error correcting Code)

- 问题：上述($n=8/k=4$)海明码的码距是几？
- 码距 $d=3$ 。因为，若有一位出错，则因该位至少要参与两组校验位的生成，因而至少引起两个校验位的不同。两个校验位加一个数据位等于3。

例如，若 M_1 出错，则故障字为0011，即 P_2 和 P_1 两个校验位发生改变，12位码字中有三位（ M_1 、 P_2 和 P_1 ）不同。

- 根据码距与检错、纠错能力的关系，知：这种码制能发现两位错，或对单个位出错进行定位和纠错。这种码称为单纠错码 (SEC)。

单纠错和双检错码

◆ 单纠错和双检错码（SEC-DED）

- 具有发现两位错和纠正一位错的能力，称为单纠错和双检错码（SEC-DED）。
- 若要成为SEC-DED，则码距需扩大到 $d=4$ 。为此，还需增加一位校验位 P_5 ，将 P_5 排列在码字的最前面，即： $P_5M_8M_7M_6M_5P_4M_4M_3M_2P_3M_1P_2P_1$ ，并使得数据中的每一位都参与三个校验位的生成。从表中可看出除了 M_4 和 M_7 外，其余位都只参与了两个校验位的生成。因此 P_5 按下式求值：

$$P_5 = M_1 \oplus M_2 \oplus M_3 \oplus M_5 \oplus M_6 \oplus M_8$$

当任意一个数据位发生错误时，必将引起三个校验位发生变化，所以码距为4。

海明校验码 最简单求法

	12	11	10	9	8	7	6	5	4	3	2	1
2^0		D_7		D_5		D_4		D_2		D_1		P_1
2^1		D_7	M_6			D_4	D_3			D_1	P_2	
2^2	D_8					D_4	D_3	D_2	P_3			
2^3	D_8	D_7	D_6	D_5	P_4							

$$P_1 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7$$

$$P_2 = D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7$$

$$P_3 = D_2 \oplus D_3 \oplus D_4 \oplus D_8$$

$$P_4 = D_5 \oplus D_6 \oplus D_7 \oplus D_8$$

8位二进制数,k=8,根据 $2^{r-1} \geq k+r$, r应为5,m=13

- 海明码可表示为: $H_{13}H_{12}.....H_2H_1$

1.校验位与数据位之和为m,每个校验位 P_i 被分在位号 2^{i-1} 的位置, $P_5 \sim P_1$ 对应 $H_{13}, H_8, H_4, H_2, H_1$

其余为数据位,形如:

$H_{13}H_{12}H_{11}H_{10}H_9H_8H_7H_6H_5H_4H_3H_2H_1$

$P_5D_8D_7D_6D_5P_4D_4D_3D_2P_3D_1P_2P_1$

每一位的位号等于校验它的各校验位位号之和.

表 出错的海明码位号和校验位位号的关系

海明码位号	数据位/校验位	参与校验的校验位位号	被校验位的海明码位号 =校验位位号之和
H ₁	P ₁	1	1=1
H ₂	P ₂	2	2=2
H ₃	D ₁	1, 2	3=1+2
H ₄	P ₃	4	4=4
H ₅	D ₂	1, 4	5=1+4
H ₆	D ₃	2, 4	6=2+4
H ₇	D ₄	1, 2, 4	7=1+2+4
H ₈	P ₄	8	8=8
H ₉	D ₅	1, 8	9=1+8
H ₁₀	D ₆	2, 8	10=2+8
H ₁₁	D ₇	1, 2, 8	11=1+2+8
H ₁₂	D ₈	4, 8	12=4+8
H ₁₃	P ₅	13	13=13

由此,可以找出有关数据位形成Pi值偶校验的结果。

$$P_1 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7$$

$$P_2 = D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7$$

$$P_3 = D_2 \oplus D_3 \oplus D_4 \oplus D_8$$

$$P_4 = D_5 \oplus D_6 \oplus D_7 \oplus D_8$$

如果要分清是两位出错还是一位出错, 还要补充一个 P_5 总校验位。使:

$$P_5 = D_1 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8 \oplus P_4 \oplus P_3 \\ \oplus P_2 \oplus P_1$$

由此,可以得出偶校验组。

$$S_1 = P_1 \oplus D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7$$

$$S_2 = P_2 \oplus D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7$$

$$S_3 = P_3 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_8$$

$$S_4 = P_4 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8$$

$$S_5 = P_5 \oplus P_4 \oplus P_3 \oplus P_2 \oplus P_1 \oplus \\ D_1 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8$$

$S_1 \sim S_5$ 能反映出13位海明码的出错情况.

$$S_1 = P_1 \oplus D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7$$

$$S_2 = P_2 \oplus D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7$$

$$S_3 = P_3 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_8$$

$$S_4 = P_4 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8$$

$$S_5 = P_5 \oplus P_4 \oplus P_3 \oplus P_2 \oplus P_1$$

$$D_1 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8$$

$S_4 \sim S_1$ 为全0,说明没错. $S_4 \sim S_1$ 不为全0,说明有错. $S_5=1$ 说明1位出错,而

$S_5=0$ 说明2位错,不再有效,且不能查出是哪2位出错。

$S_4 \sim S_1$ 的编码值正好等于出错的海明码的位号.如:1100和1011,分别表示 $H_{12}(D_8)$ 和 $H_{11}(D_7)$ 出错。

图 3.11 是 11~12, 数据位 8, 校验位 4 的海明校验线路, 记作 (12, 8) 分组码。

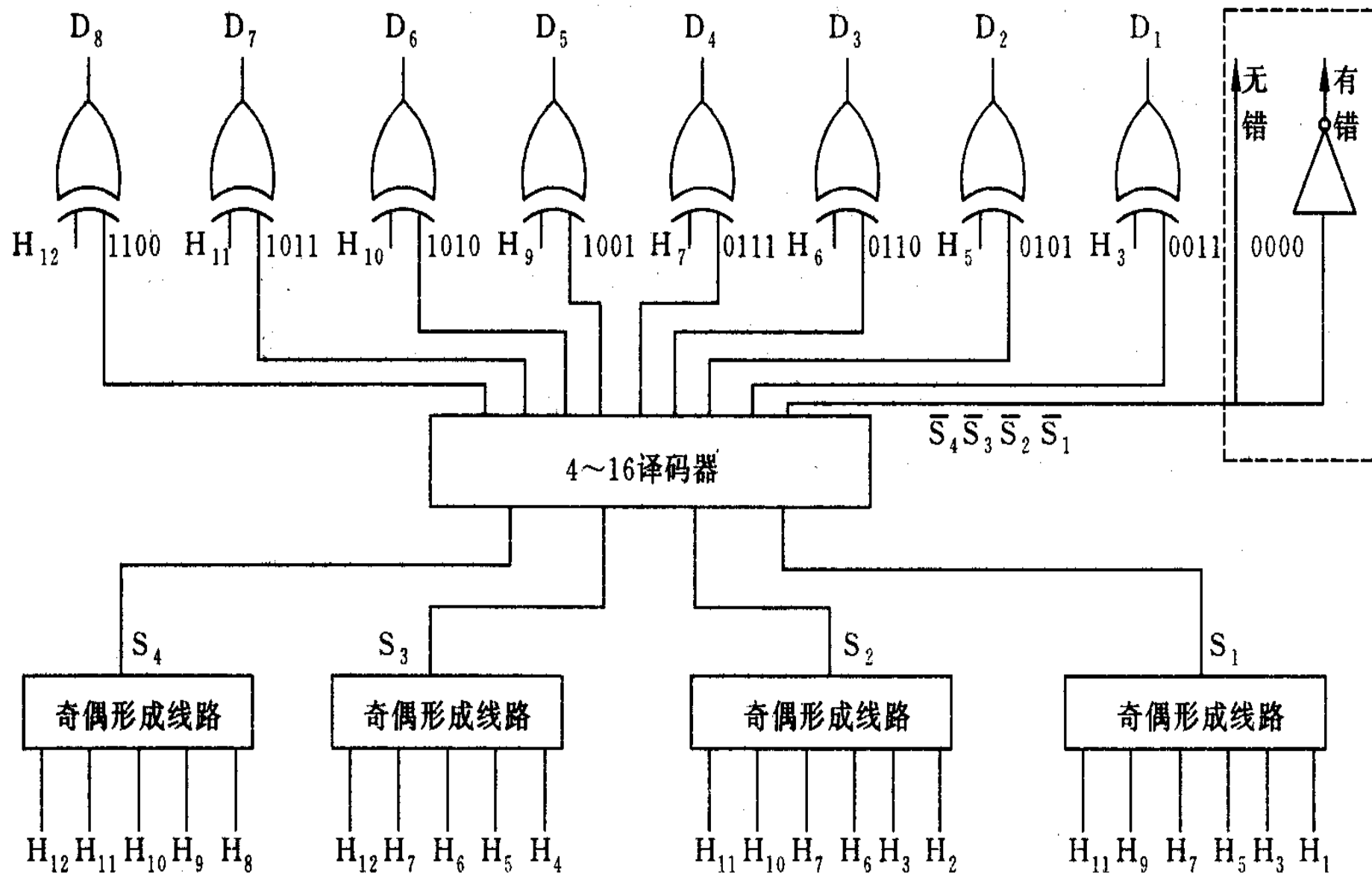


图 3.11 (12, 8) 分组码海明校验线路

若要判断是一位出
错还是两位出错,再
增加一个 P_5 , (H_{13})并
用右图来判断。

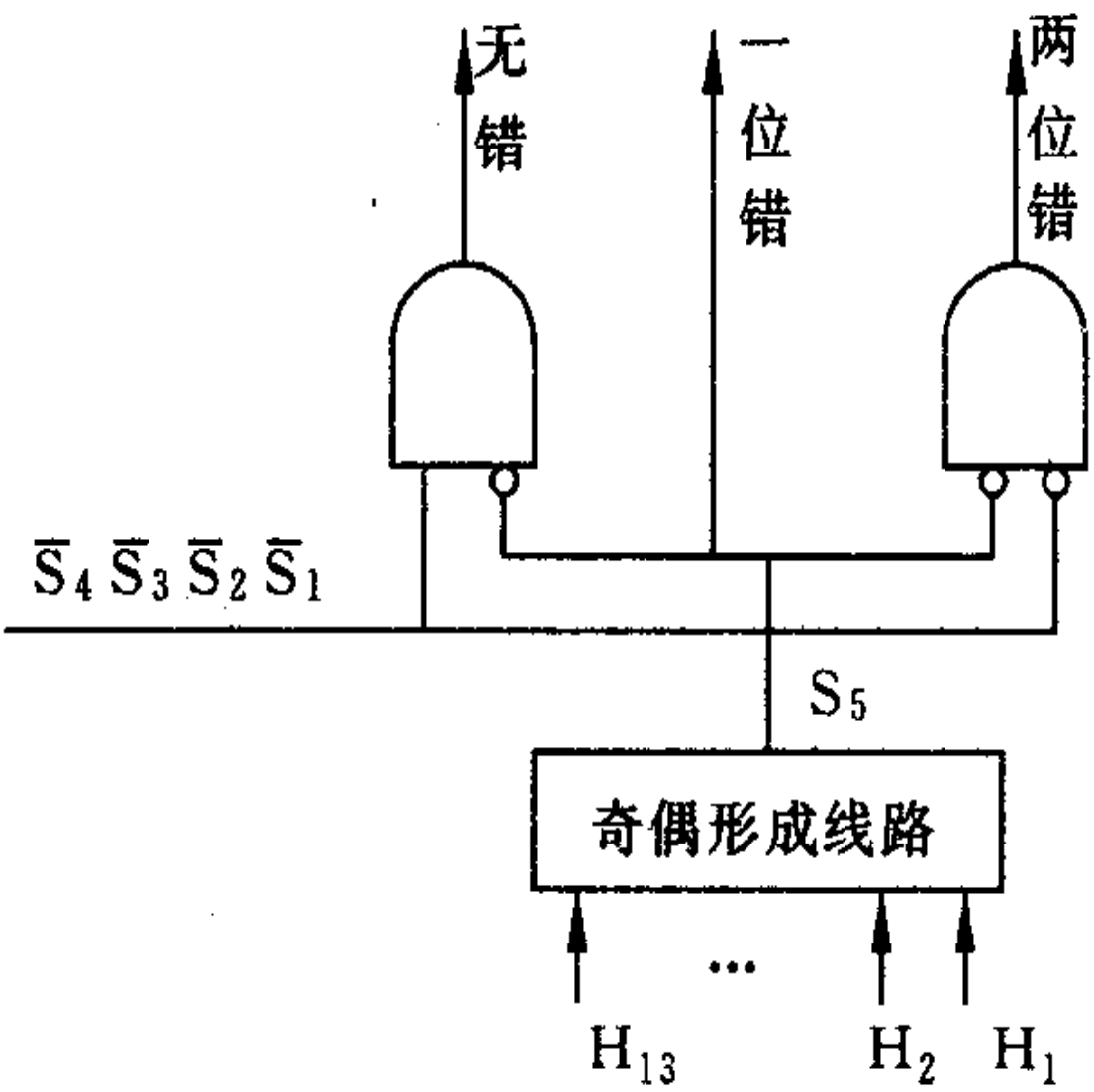


图 3.12 判 1 位/2 位错的附加线路

循环冗余码

循环冗余校验码（Cyclic Redundancy Check），简称CRC码

- 具有很强的检错、纠错能力。
- 用于大批量数据存储和传送(如：外存和通信)中的数据校验。

为什么大批量数据不用奇偶校验？

在每个字符后增加一位校验位会增加大量的额外开销；尤其在网络通信中，对传输的二进制比特流没有必要再分解成一个个字符，因而无法采用奇偶校验码。

- 通过某种数学运算来建立数据和校验位之间的约定关系。

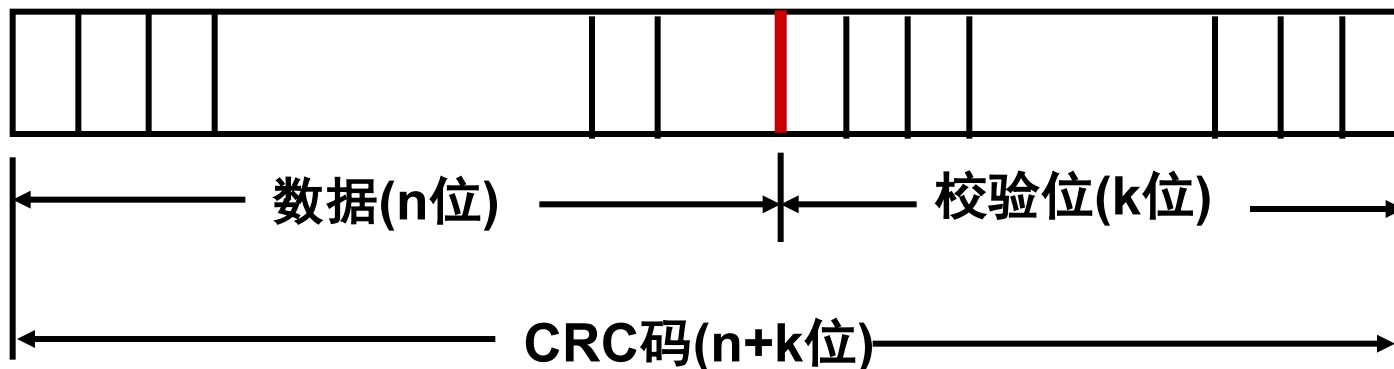
奇偶校验码和海明校验码都是以奇偶检测为手段的。

网络或通信课程中会学到。

CRC码的检错方法

基本思想：

- 数据信息 $M(x)$ 为一个 n 位的二进制数据，将 $M(x)$ 左移 k 位后，用一个约定的“生成多项式” $G(x)$ 相除， $G(x)$ 是一个 $k+1$ 位的二进制数，相除后得到的 k 位余数就是校验位。校验位拼接到 $M(x)$ 后，形成一个 $n+k$ 位的代码，称该代码为循环冗余校验 (CRC) 码，也称 $(n+k, n)$ 码。
- 一个CRC码一定能被生成多项式整除，当数据和校验位一起送到接受端后，只要将接受到的数据和校验位用同样的生成多项式相除，如果正好除尽，表明没有发生错误；若除不尽，则表明某些数据位发生了错误。通常要求重传一次。



CRC码

k位的有效信息位表达为多项式 $M(x)$:

$$M(x) = C_{k-1}x^{k-1} + C_{k-2}x^{k-2} + \dots + C_i x^i + \dots + C_1 x^1 + C_0$$

将信息位组左移r位, 则可表示为多项式 $M(x) \cdot x^r$ 。

信息位组 k位

左移r位 k位 r位

$$n = (k+r) \text{位}$$

CRC码是用 $M(x) \cdot x^r$ 除以 $G(x)$ (生成多项式) 所得余数作为校验位的。 (要求 $G(x)$ 必须是r+1位的)

CRC码的格式为: $M(x) \cdot x^r + R(x)$

CRC码

- 例: $M(x) = x^3 + x^2 = 1100$

$$M(x) \cdot x^3 = x^6 + x^5 = 1100000$$

$$G(x) = x^3 + x + 1 = 1011$$

$$\frac{M(x) \cdot x^3}{G(x)} = \frac{1100000}{1011} = 1110 + \frac{010}{1011}$$

$$\begin{aligned} M(x) \cdot x^3 + R(x) &= 1100000 + 010 \\ &= 1100010 \end{aligned}$$

编好的循环校验码称为(7,4)码,即 $n=7, k=4$

CRC码

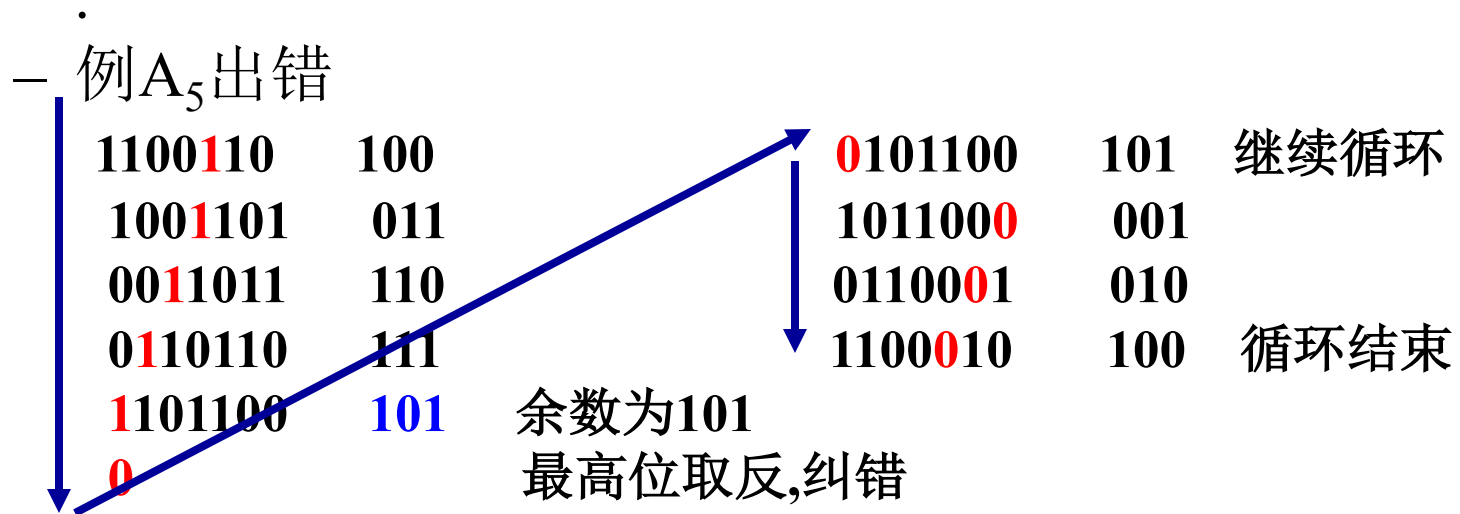
CRC的译码与纠错

	A ₁	A ₂	A ₃	A ₄	A ₅	A ₆	A ₇	余数
正确	1	1	0	0	0	1	0	0 0 0
A ₇ 错	1	1	0	0	0	1	1	0 0 1
A ₆ 错	1	1	0	0	0	0	0	0 1 0
A ₅ 错	1	1	0	0	1	1	0	1 0 0
A ₄ 错	1	1	0	1	0	1	0	0 1 1
A ₃ 错	1	1	1	0	0	1	0	1 1 0
A ₂ 错	1	0	0	0	0	1	0	1 1 1
A ₁ 错	0	1	0	0	0	1	0	1 0 1

- 将收到的CRC码用约定的生成多项式G(x)去除,如果码字无误则余数应位0,如果有某一位出错,则余数不为0,不同位数出错余数不同.
- 如果对余数补个0继续除下去,我们将发现一个现象:各次余数将按上表顺序循环.

CRC码

- 根据不同的余数来纠正不同的出错位
- 循环除法:
 - 余数添0继续除法,同时信息部分进行循环移位.
 - 当最高位变成101时,则最高位取反,纠错.
 - 继续余数除法,直至循环一遍,余数变成第一次的余数



循环冗余码举例

校验位的生成：用一个例子来说明校验位的生成过程。

- 假设要传送的数据信息为：100011，即报文多项式为：

$$M(x) = x^5 + x + 1。 \text{数据信息位数 } n=6。$$

- 若约定的生成多项式为： $G(x) = x^3 + 1$ ，则生成多项式位数为4位，所以校验位位数 $k=3$ ，除数为1001。
- 生成校验位时，用 $x^3 \cdot M(x)$ 去除以 $G(x)$ ，即：
 $100011000 \div 1001$ 。
- 相除时采用 “模2运算” 的多项式除法。

循环冗余校验 (CRC) 码 (模2运算)

- CRC码的编码方法

- 模2加减: 异或逻辑— $0 \oplus 0=0$; $0 \oplus 1=1$;
 $1 \oplus 0=1$; $1 \oplus 1=0$.

- 模2乘:

$$\begin{array}{r} 1010 \\ \times \quad 101 \\ \hline 1010 \\ 0000 \\ 1010 \\ \hline 100010 \end{array}$$

- 模2除:

$$\begin{array}{r} 101 \overline{) 10000} \\ \underline{101} \\ 010 \\ \underline{000} \\ 100 \\ \underline{101} \\ 01 \end{array} \quad \begin{array}{l} \dots \text{商} \\ \\ \\ \\ \\ \dots \text{余数} \end{array}$$

循环冗余码举例

$$X^3 \cdot M(x) \div G(x) = (x^8 + x^4 + x^3) \div (x^3 + 1)$$

$$\begin{array}{r} 1001 \overline{) 100011000} \\ \underline{1001} \\ 0011 \\ \underline{0000} \\ 0111 \\ \underline{0000} \\ 1110 \\ \underline{1001} \\ 1110 \\ \underline{1001} \\ 111 \end{array}$$

111 余数

(模2运算不考虑加法进位和减法借位，上商的原则是当部分余数首位是1时商取1，反之商取0。然后按模2相减原则求得最高位后面几位的余数。这样当被除数逐步除完时，最后的余数位数比除数少一位。这样得到的余数就是校验位，此例中最终的余数有3位。)

校验位为111，CRC码为100011 111。如果要校验CRC码，可将CRC码用同一个多项式相除，若余数为0，则说明无错；否则说明有错。例如，若在接收方的CRC码也为100011 111时，用同一个多项式相除后余数为0。若接收方CRC码不为100011 111时，余数则不为0。

第二讲小结

◆ 非数值数据的表示

- 逻辑数据用来表示真/假或N位位串，按位运算
- 西文字符：用ASCII码表示
- 汉字：汉字输入码、汉字内码、汉字字模码

◆ 数据的宽度

- 位、字节、字（不一定等于字长），k/K/M/G/...有不同的含义

◆ 数据的存储排列

- 数据的地址：连续若干单元中最小的地址，即：从小地址开始存放数据
 - 问题：若一个short型数据si存放在单元0x08000100和0x08000101中，那么si的地址是什么？
- 大端方式：用MSB存放的地址表示数据的地址
- 小端方式：用LSB存放的地址表示数据的地址
- 按边界对齐可减少访存次数

◆ 数据的纠错和检错

- 奇偶校验：适应于一字节长数据的校验，如内存
- 海明校验：各组内用奇偶校验，用于内存储器数据的校验
- 循环冗余校验：用在通信和外存中，适合于大批量数据校验

附录： Decimal / Binary（十 / 二进制数）

- ◆ The **decimal** number 5836.47 in **powers of 10**:

$$\begin{aligned} &5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 \\ &+ 4 \times 10^{-1} + 7 \times 10^{-2} \end{aligned}$$

- ◆ The **binary** number 11001 in **powers of 2**:

$$\begin{aligned} &1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 0 + 0 + 1 = 25 \end{aligned}$$

- ◆ 用一个下标表示数的基（**radix / base**）

$$11001_2 = 25_{10}$$

附录： Octal / Hexadecimal (八 / 十六进制数)

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \hline 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline \end{array} = 2000_{10}$$

$$v = \sum_{i=0}^{n-1} 2^i b_i$$

$$2^3=8$$

03720

Octal - base 8

000 - 0

001 - 1

010 - 2

011 - 3

100 - 4

101 - 5

110 - 6

111 - 7

$$2^4=16$$

0x7d0

Hexadecimal - base 16

0000 - 0 1000 - 8

0001 - 1 1001 - 9

0010 - 2 1010 - a

0011 - 3 1011 - b

0100 - 4 1100 - c

0101 - 5 1101 - d

0110 - 6 1110 - e

0111 - 7 1111 - f

计算机用二进制表示所有信息！

为什么要引入 8 / 16进制？

8 / 16进制是二进制的简便表示。便于阅读和书写！

它们之间对应简单，转换容易。

在机器内部用二进制，在屏幕或其他外部设备上表示时，转换为8/16进制数，可缩短长度

附录： Conversions of numbers

(1) R进制数 => 十进制数

按“权”展开 (a power of R)

例1: $(10101.01)_2 = 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-2} = (21.25)_{10}$

例2: $(307.6)_8 = 3 \times 8^2 + 7 \times 8^0 + 6 \times 8^{-1} = (199.75)_{10}$

例1: $(3A.1)_{16} = 3 \times 16^1 + 10 \times 16^0 + 1 \times 16^{-1} = (58.0625)_{10}$

(2) 十进制数 => R进制数

整数部分和小数部分分别转换

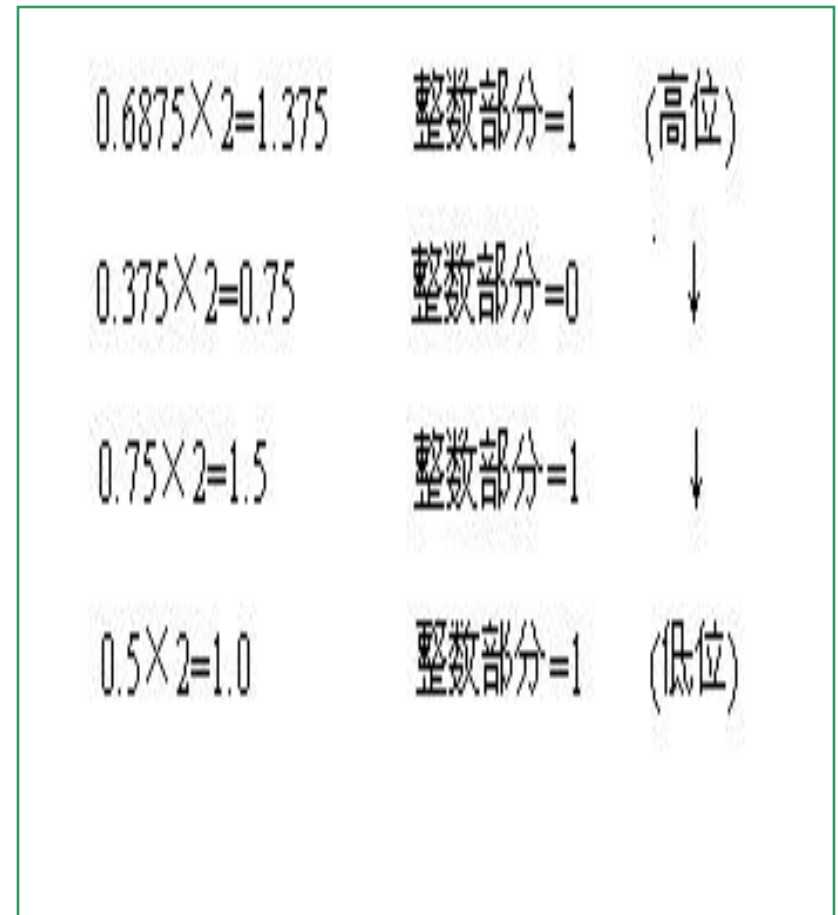
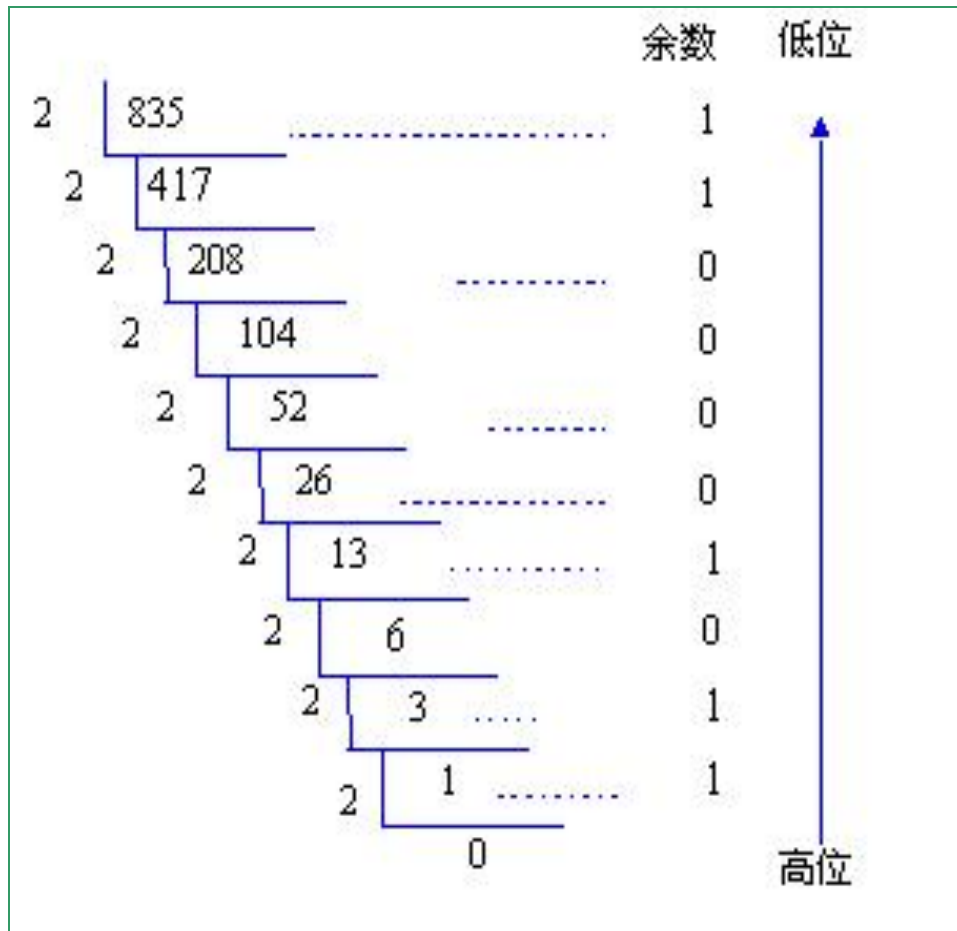
- ① 整数(integral part)----“除基取余，上右下左”
- ② 小数(fractional part)----“乘基取整，上左下右”

附录： Decimal to Binary Conversions

例1: $(835.6785)_{10} = (1101000011.1011)_2$

整数——“除基取余，上右下左”

小数——“乘基取整，上左下右”



附录： Decimal to Binary Conversions

例2: $(835.63)_{10} = (1503.50243...)_{8}$

整数——“除基取余，上右下左” 小数——“乘基取整，上左下右”

有可能乘积的小数部分总得不到0，此时得到一个近似值。

		低位
8	835	余数
8	104	3
8	13	0
8	1	5
	0	1
		高位

$0.63 \times 8 = 5.04$	整数部分=5	(高位)
$0.04 \times 8 = 0.32$	整数部分=0	
$0.32 \times 8 = 2.56$	整数部分=2	
$0.56 \times 8 = 4.48$	整数部分=4	
$0.48 \times 8 = 3.84$	整数部分=3	(低位)

附录: Conversions of numbers

(3) 二/八/十六进制数的相互转换

① 八进制数转换成二进制数

$$(13.724)_8 = (001\ 011\ .\ 111\ 010\ 100)_2 = (1011.1110101)_2$$

② 十六进制数转换成二进制数

$$(2B.5E)_{16} = (00101011\ .\ 01011110)_2 = (101011.0101111)_2$$

③ 二进制数转换成八进制数

$$(0.10101)_2 = (000\ .\ 101\ 010)_2 = (0.52)_8$$

④ 二进制数转换成十六进制数

$$(11001.11)_2 = (0001\ 1001\ .\ 1100)_2 = (19.C)_{16}$$