南京大学
人工智能学院
SCHOOL OF ARTIFICIAL INTELLIGENCE, NANJING UNIVERSITY

第1章 二进制编码

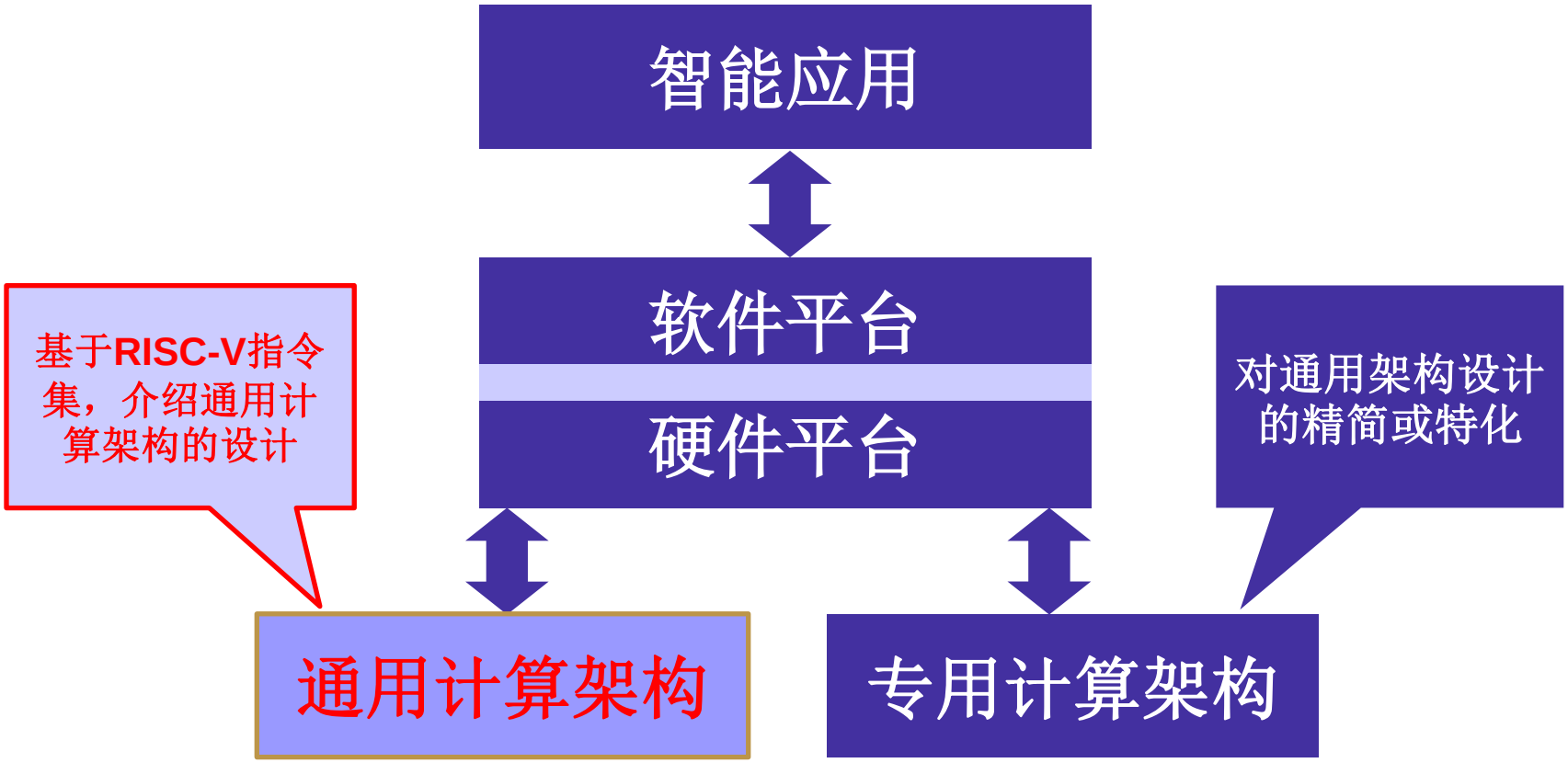
第一讲 计算机系统概述

第二讲 二进制数的表示

第三讲 数值数据的编码表示

第四讲 非数值数据的编码表示及数据的宽度和存储排列

第一讲 计算机系统概述


智能应用
软件平台
硬件平台
基于RISC-V指令集，介绍通用计算架构的设计
对通用架构设计的精简或特化
通用计算架构
专用计算架构

第一讲 计算机系统概述

- 冯.诺依曼结构计算机
  - 冯.诺依曼结构基本思想
  - 计算机硬件的基本组成
- 程序的表示和执行过程
  - 机器级语言和高级编程语言
  - 翻译程序：汇编、编译、解释
- 计算机系统抽象层
  - 计算机硬件和软件的接口：指令系统
  - 本课程内容在计算机系统中的位置

冯·诺依曼的故事

- 1944年，冯•诺依曼参加原子弹的研制工作，涉及到极为困难的计算。
- 1944年夏的一天，冯•诺依曼巧遇美国弹道实验室的军方负责人戈尔斯坦，他正参与ENIAC的研制工作。
- 冯•诺依曼被戈尔斯坦介绍加入ENIAC研制组，1945年，在共同讨论的基础上，冯•诺依曼以“关于EDVAC的报告草案”为题，起草了长达101页的总结报告，发表了全新的“存储程序通用电子计算机方案”。
- 一向专搞理论研究的普林斯顿高等研究院（the Institute for Advance Study at Princeton，IAS）批准让冯•诺依曼建造计算机，其依据就是这份报告。

Electronic
Discrete
Variable
Automatic
Computer

现代计算机的原型

1946年，普林斯顿高等研究院（the Institute for Advance Study at Princeton，IAS）开始设计“存储程序”计算机，被称为IAS计算机（1951年才完成，它并不是第一台存储程序计算机，1949年由英国剑桥大学完成的EDSAC是第一台）。

- 在那个报告中提出的计算机结构被称为冯•诺依曼结构。
- 冯•诺依曼结构最重要的思想是什么?

  “存储程序(Stored-program)” 工作方式：

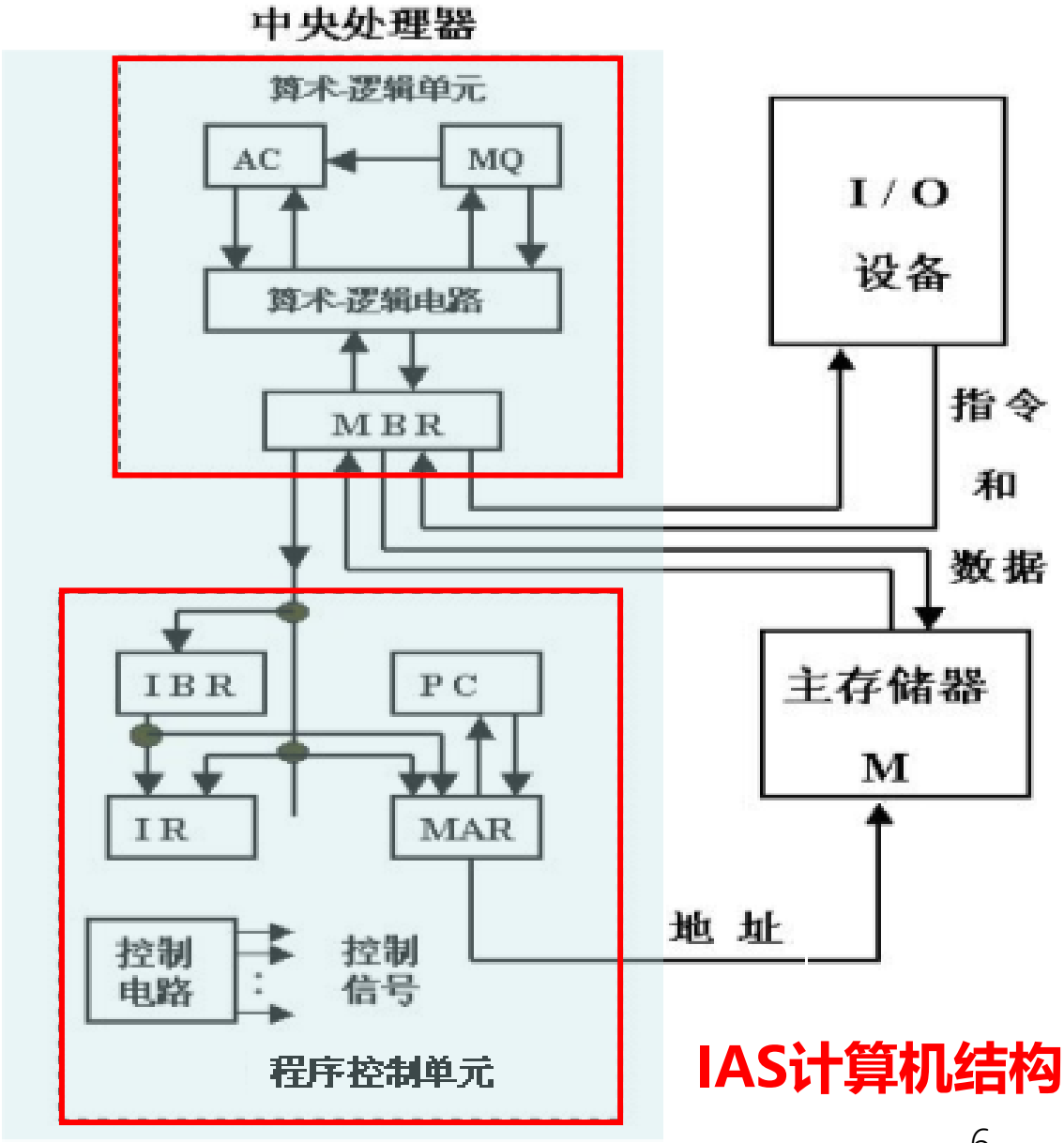
  任何要计算机完成的工作都要先被编写成程序（指令序列），存放在存储器中。一旦程序被启动，计算机应能在不需操作人员干预下，自动完成逐条取出指令和执行指令的任务。
- 冯•诺依曼结构计算机也称为冯•诺依曼机器（Von Neumann Machine）。
- 几乎现代所有的通用计算机大都采用冯•诺依曼结构，因此，IAS计算机是现代计算机的原型机。

你认为冯·诺依曼结构是怎样的？

- 应该有个主存，用来存放程序和数据
- 应该有一个自动逐条取出指令的部件
- 还应该有具体执行指令（即运算）的部件
- 程序由指令构成
- 指令描述如何对数据进行处理
- 应该有将程序和原始数据输入计算机的部件
- 应该有将运算结果输出计算机的部件

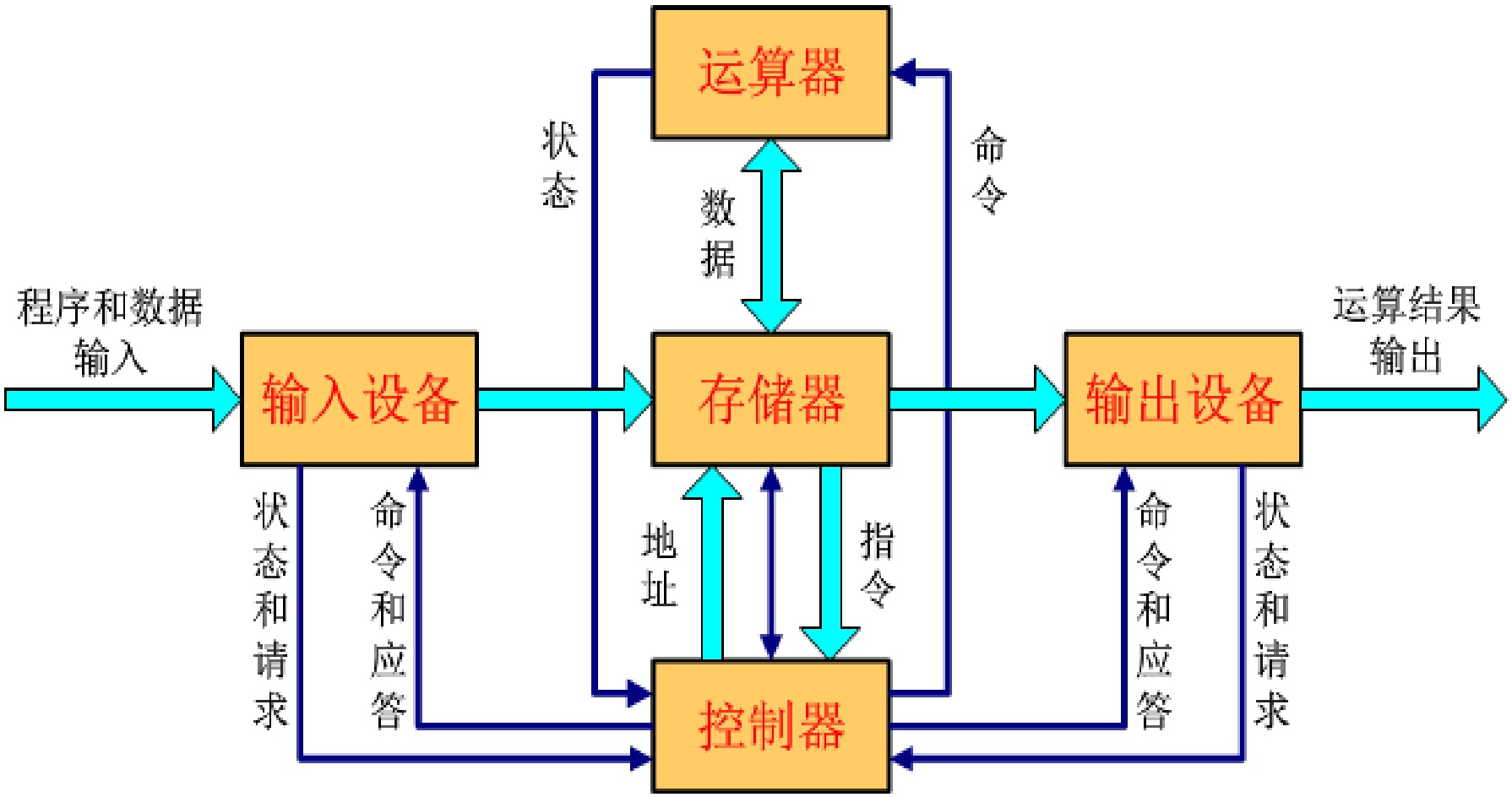
你还能想出更多吗？

你猜得八九不离十了☺



IAS计算机结构

冯.诺依曼结构计算机模型



早期，部件之间用分散方式相连

现在，部件之间大多用总线方式相连

趋势，点对点（分散方式）高速连接

冯·诺依曼结构的主要思想

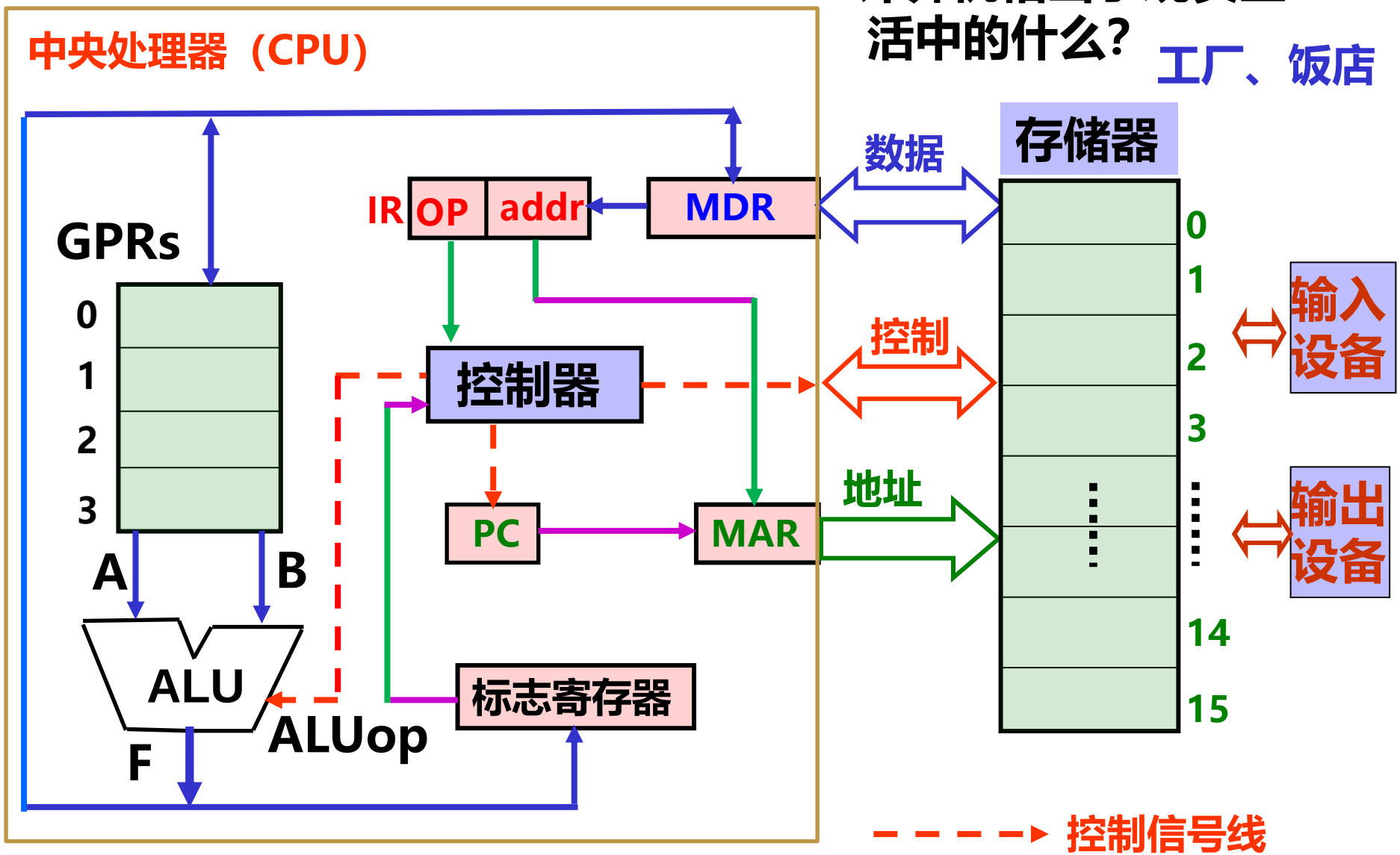
NJUAI

冯•诺依曼结构的主要思想是什么呢?

1. 计算机应由运算器、控制器、存储器、输入设备和输出设备五个基本部件组成。
2. 各基本部件的功能是:
   - 存储器不仅能存放数据，而且也能存放指令，形式上两者没有区别，给定场景条件下，才能区分数据还是指令;
   - 控制器应能自动取出指令来执行;
   - 运算器应能进行基本算术或逻辑运算等，如：加/减/乘/除，与或非逻辑操作; （完备性）
   - 操作人员可以通过输入设备、输出设备和主机进行交互。
3. 内部以二进制表示指令和数据。
   - 每条指令由操作码和地址码两部分组成。操作码指出操作类型，地址码指出操作数的地址。由一串指令组成程序。
4. 采用“存储程序”工作方式。

现代计算机结构模型

计算机相当于现实生活中的什么？ 工厂、饭店



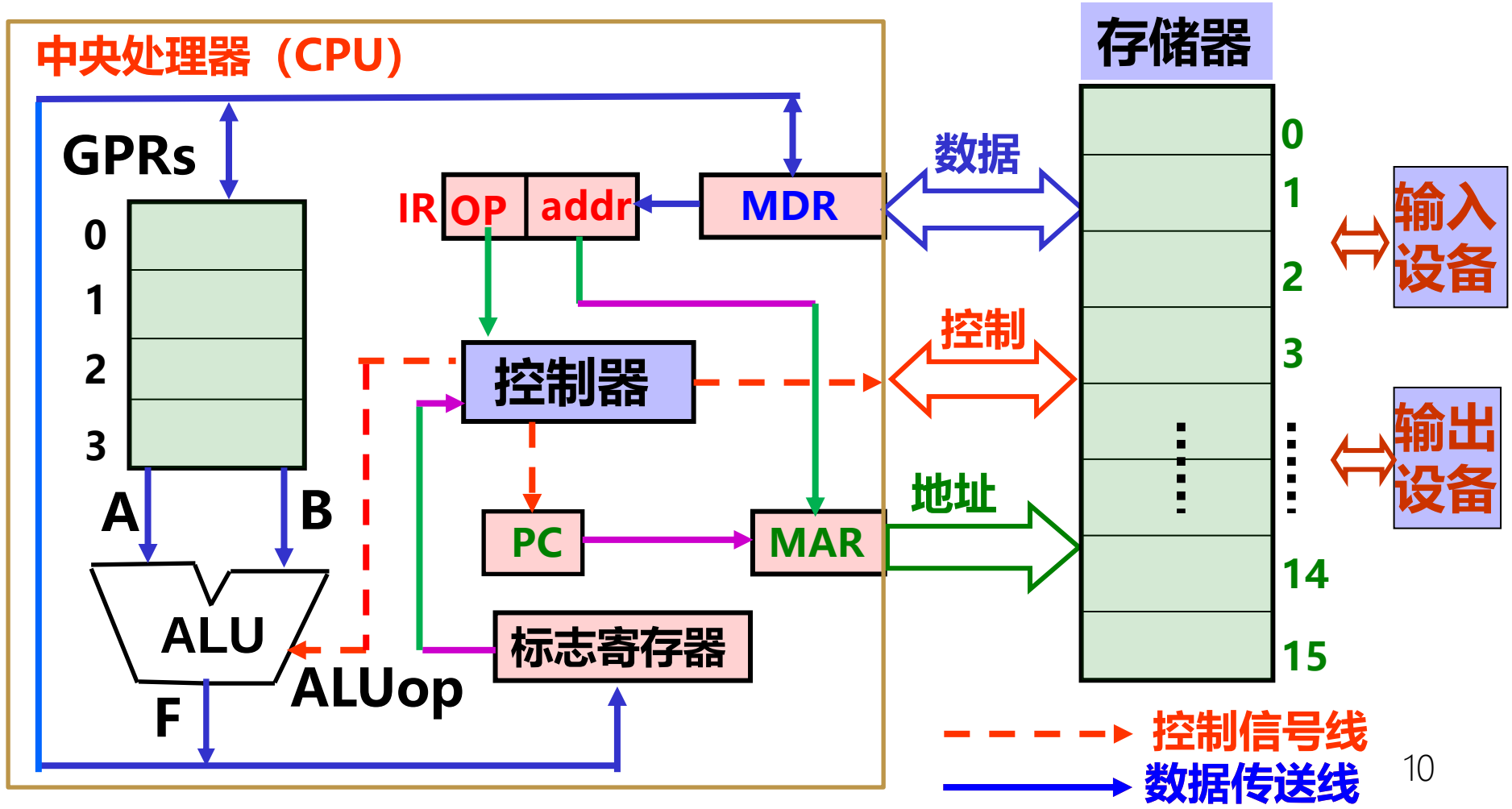
计算机是如何工作的呢？

认识计算机中最基本的部件

CPU：中央处理器；PC：程序计数器；MAR：存储器地址寄存器

ALU：算术逻辑部件；IR：指令寄存器；MDR：存储器数据寄存器

GPRs：通用寄存器组（由若干通用寄存器组成，早期就是累加器）



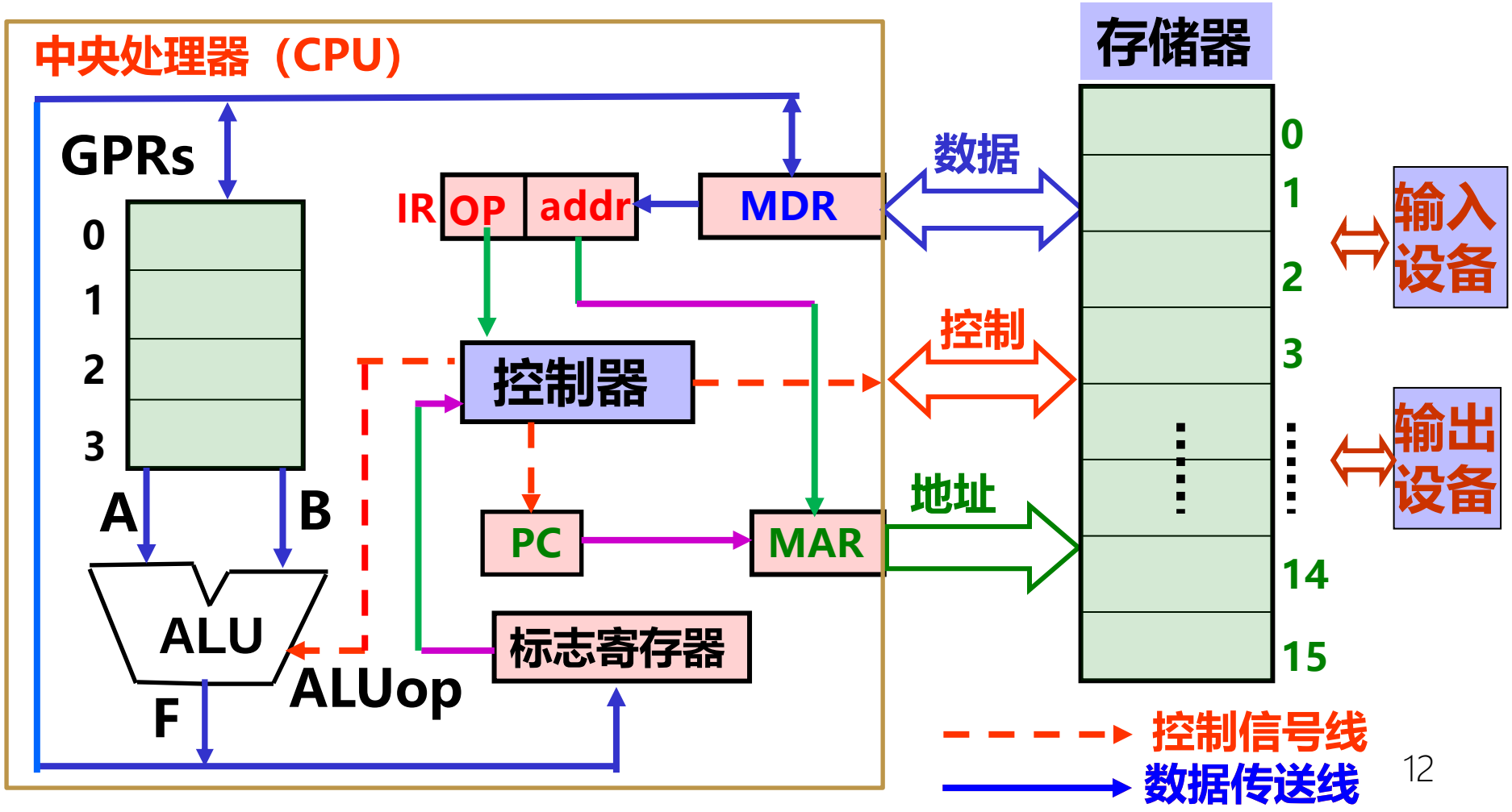
第一讲 计算机系统概述

- 冯.诺依曼结构计算机
  - 冯.诺依曼结构基本思想
  - 计算机硬件的基本组成
- 程序的表示和执行过程
  - 机器级语言和高级编程语言
  - 翻译程序：汇编、编译、解释
- 计算机系统抽象层
  - 计算机硬件和软件的接口：指令系统
  - 本课程内容在计算机系统中的位置

计算机是如何工作的？

先想象一下厨师是怎样做一桌你喜欢（指定）的菜的?

厨房-CPU，厨师-控制器，盘-GPRs，锅灶等-ALU ，架子-存储器



计算机是如何工作的？

NJUAI

类似“存储程序”工作方式

- 做菜前

  原材料（数据）和菜谱（指令）都按序放在厨房外的架子（存储器）上， 每个架子有编号（存储单元地址）。

  菜谱上信息：原料位置、做法、做好的菜放在哪里等

  例如，把10、11号架上的原料一起炒，并装入3号盘

  然后，厨师从第5个架上（起始PC=5）指定菜谱开始做

- 开始做菜

  第一步：从5号架上取菜谱（根据PC取指令）

  第二步：看菜谱（指令译码）

  第三步：从架上或盘中取原材料（取操作数）

  第四步：洗、切、炒等具体操作（指令执行）

  第五步：装盘或直接送桌（回写结果）

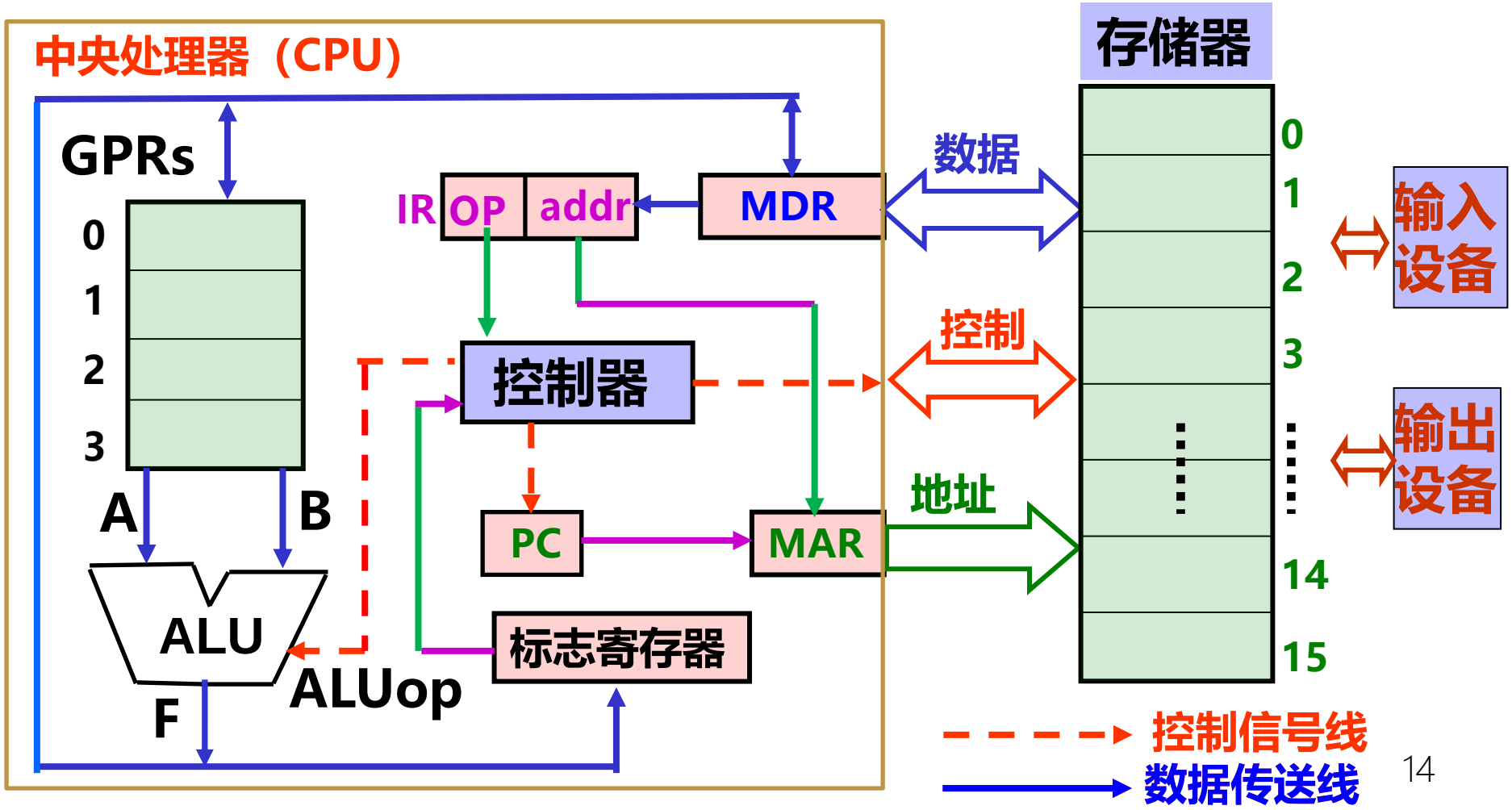
  第六步：算出下一菜谱所在架子号6=5+1（修改PC的值）

  继续做下一道菜（执行下一条指令）

计算机是如何工作的？

如果你也会做菜的，你就已经知道计算机是如何工作的！

“存储程序”工作方式！



计算机是如何工作的？

程序由指令组成，若所有指令执行完，则程序执行结束

- **程序在执行前**

  数据和指令事先存放在存储器中，每条指令和每个数据都有地址，指令按序存放，指令由OP、ADDR字段组成，程序起始地址置PC

  （原材料和菜谱都放在厨房外的架子上， 每个架子有编号。从第5个架上指定菜谱开始做）

- **开始执行程序**

  第一步：根据PC取指令（从5号架上取菜谱）

  第二步：指令译码（看菜谱）

  第三步：取操作数（从架上或盘中取原材料）

  第四步：指令执行（洗、切、炒等具体操作）

  第五步：回写结果（装盘或直接送桌）

  第六步：修改PC的值（算出下一菜谱所在架子号6=5+1）

  继续执行下一条指令（继续做下一道菜）

计算机是如何工作的？

- 程序启动前，指令和数据都存放在存储器中，形式上没有差别，都是0/1序列
- 采用”存储程序“工作方式：
  - 程序由指令组成，程序被启动后，计算机能自动取出一条一条指令执行，在执行过程中无需人的干预。
- 指令执行过程中，指令和数据被从存储器取到CPU，存放在CPU内的寄存器中，指令在IR中，数据在GPR中。

指令中需给出的信息：　　IR？ GPR?

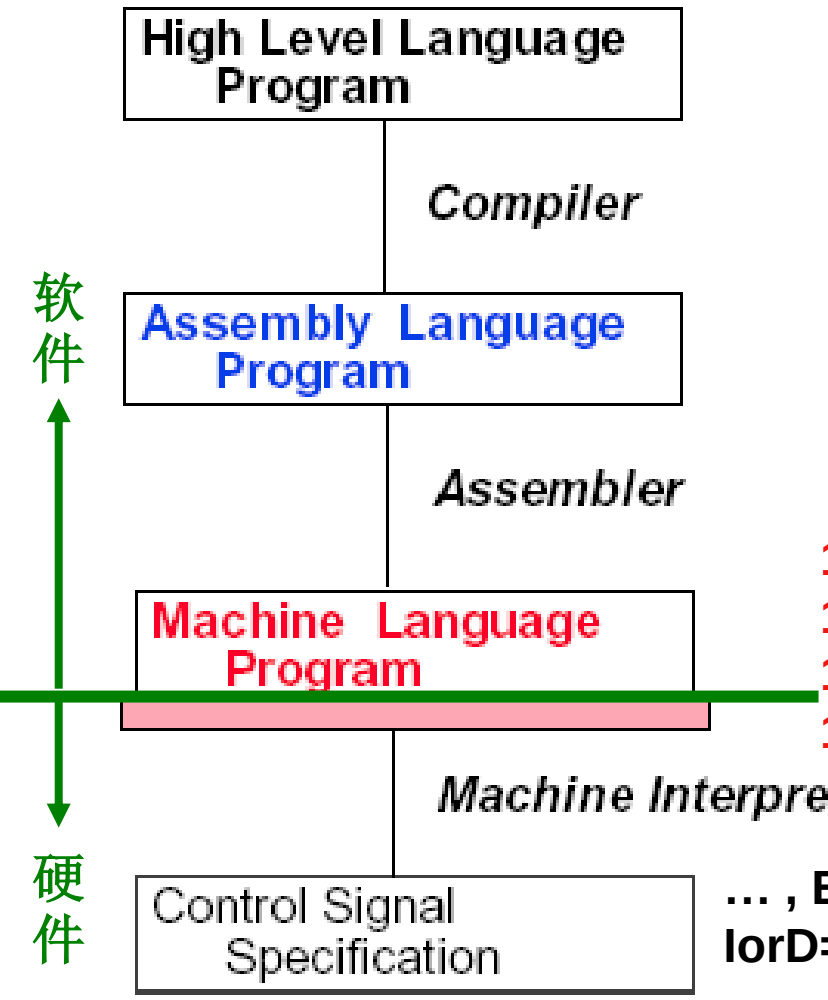
操作性质（操作码）

源操作数1 或/和 源操作数2　　（立即数、寄存器编号、存储地址）

目的操作数地址　　（寄存器编号、存储地址）

存储地址的描述与操作数的数据结构有关！

Hardware/Software Interface



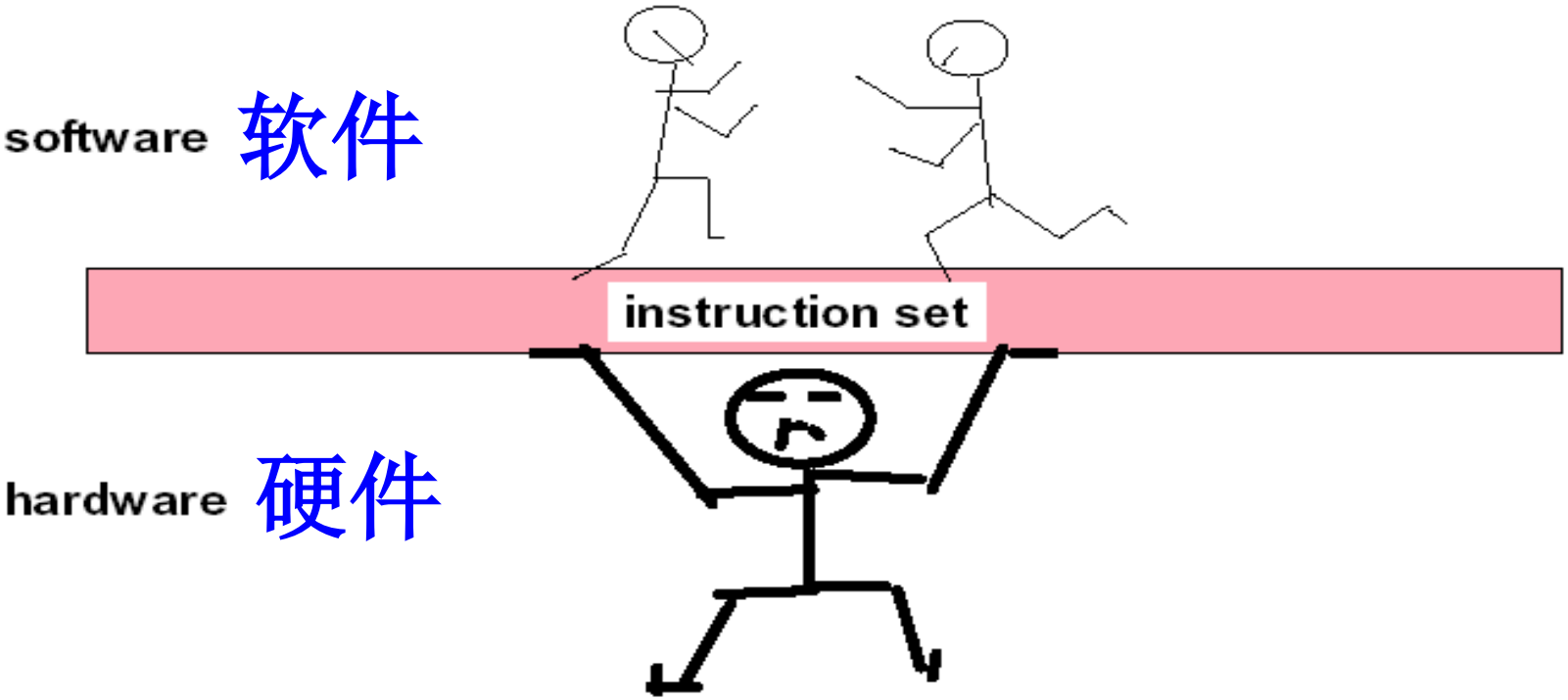

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
```

```
1000 1100 0100 1111 0000 0000 0000 0000
1000 1100 0101 0000 0000 0000 0000 0100
1010 1100 0101 0000 0000 0000 0000 0000
1010 1100 0100 1111 0000 0000 0000 0100
```

… , EXTop=1,ALUSelA=1,ALUSelB=11,ALUop=add, IorD=1,Read,MemtoReg=1,RegWr=1,.....

Hardware/Software Interface（界面）



软、硬件界面：指令集体系结构（Instruction Set Architecture，ISA）

有时简称系统结构、体系结构，指令系统，甚至简称“架构”

机器语言由指令代码构成，能被硬件直接执行。

指令集体系结构（ISA）

- ISA指Instruction Set Architecture，即指令集体系结构
- ISA是一种规约（Specification），它规定了如何使用硬件
  - 可执行的指令的集合，包括指令格式、操作种类以及每种操作对应的操作数的相应规定；
  - 指令可以接受的操作数的类型；
  - 操作数所能存放的寄存器组的结构，包括每个寄存器的名称、编号、长度和用途；
  - 操作数所能存放的存储空间的大小和编址方式；
  - 操作数在存储空间存放时按照大端还是小端方式存放；
  - 指令获取操作数的方式，即寻址方式；
  - 指令执行过程的控制方式，包括程序计数器、条件码定义等。
- ISA在计算机系统中是必不可少的一个抽象层，Why?
  - 没有它，软件无法使用计算机硬件！
  - 没有它，一台计算机不能称为“通用计算机”

软件（Software）

- System software(系统软件) - 简化编程，并使硬件资源被有效利用
  - 操作系统（Operating System）：硬件资源管理，用户接口
  - 语言处理系统：翻译程序+ Linker, Debug, etc …
    - 翻译程序(Translator)有三类：

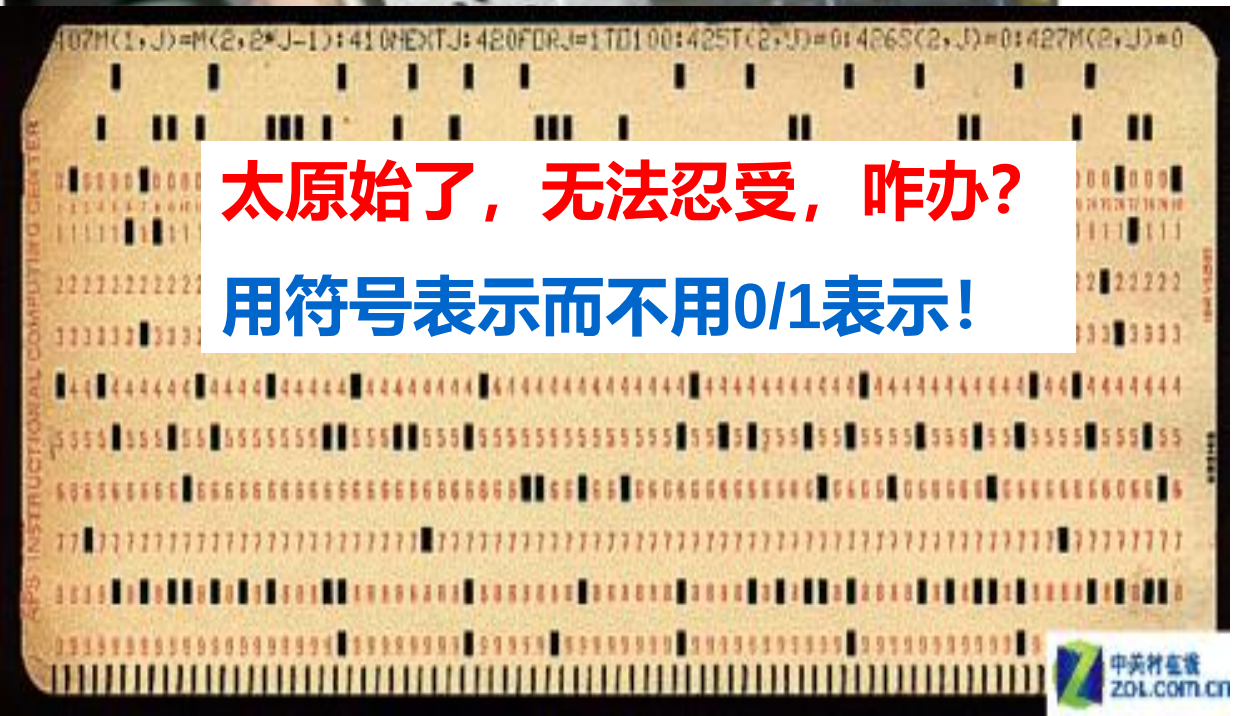
      汇编程序(Assembler)：汇编语言源程序→机器目标程序

      编译程序(Complier)：高级语言源程序→汇编/机器目标程序

      解释程序(Interpreter)：将高级语言语句逐条翻译成机器指令并立即执行,不生成目标文件。
  - 其他实用程序: 如：磁盘碎片整理程序、备份程序等
- Application software(应用软件) - 解决具体应用问题/完成具体应用
  - 各类媒体处理程序：Word/ Image/ Graphics/…
  - 管理信息系统 (MIS)
  - Game，…

最早的程序开发过程

- **用机器语言编写程序，并记录在纸带或卡片上**



输入：按钮、开关；所有信息都
输出：指示灯等　　是0/1序列！

假设：0010-jxx　转移指令

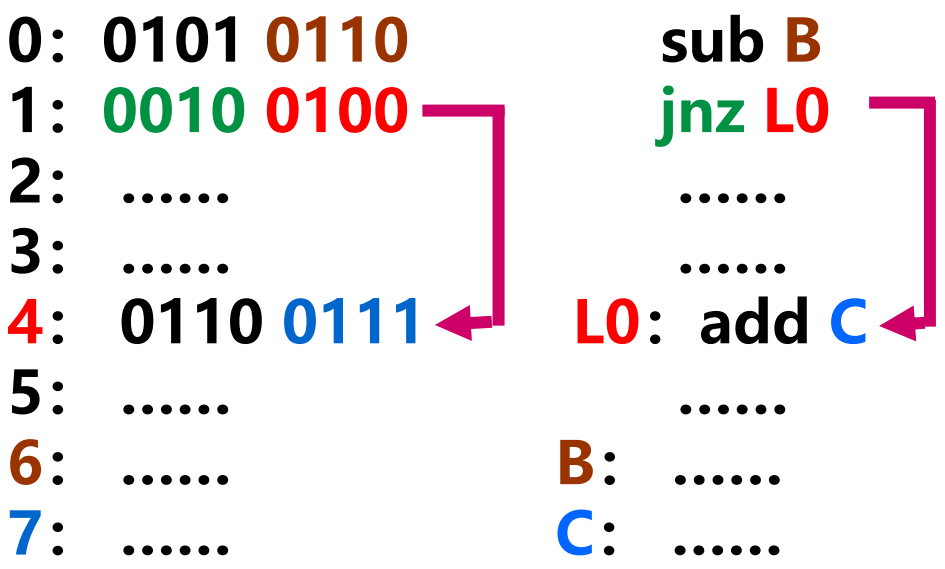
```
0: 0101 0110
1: 0010 0100
2: ......
3: ......
4: 0110 0111
5: ......
6: ......
```

若在第4条指令前加入指令，则需重新计算地址码（如jxx的目标地址），然后重新打孔。不灵活！

书写、阅读困难！

用汇编语言开发程序

- 若用符号表示跳转位置和变量位置，是否简化了问题?
- 于是，汇编语言出现
  - 用助记符表示操作码
  - 用标号表示位置
  - 用助记符表示寄存器
  - .....



你认为用汇编语言编写的优点是:

不会因为增减指令而需要修改其他指令

不需记忆指令码，编写方便

可读性比机器语言强

不过，这带来新的问题，是什么呢?

人容易了，可机器不认识这些指令了!

需将汇编语言转换为机器语言!

用汇编程序转换

在第4条指令前加指令时不用改变sub、jnz和add指令中的地址码!

进一步认识机器级语言

- 汇编语言源程序由汇编指令构成
- 你能用一句话描述什么是汇编指令吗?
  - 用助记符和标号来表示的指令（与机器指令一一对应）
- 指令又是什么呢?
  - 包含操作码和操作数或其地址码
    （机器指令用二进制表示，汇编指令用符号表示）
  - 可以描述：取（或存一个数）
    两个数加（或减、乘、除、与、或等）
    根据运算结果判断是否转移执行
- 想象用汇编语言编写复杂程序是怎样的情形?
  （例如，用汇编语言实现排序（sort）、矩阵相乘）
  - 需要描述的细节太多了！程序会很长很长！而且在不同结构的机器上就不能运行!

```
    sub B
    jnz L0
    ......
    ......
L0: add C
    ......
B:  ......
C:  ......
```

机器语言和汇编语言都是面向机器结构的语言，故它们统称为机器级语言

SKIP

结论：用汇编语言比机器语言好，但是，还是很麻烦!

指令所能描述的功能

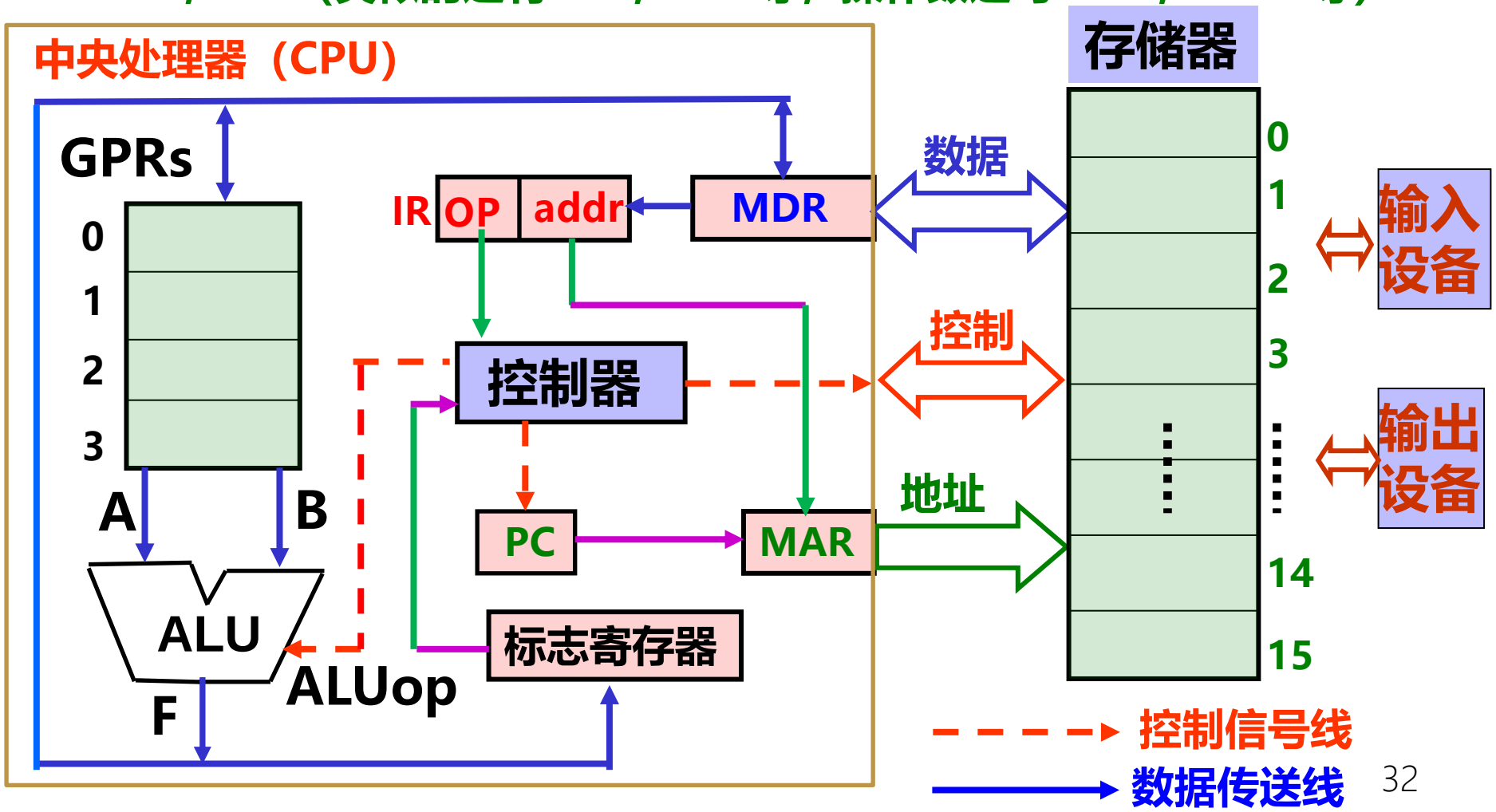
对于以下结构的机器，你能设计出几条指令吗?

Load M#，R#　　(将存储单元内容装入寄存器)

Store R#，M#　　(将寄存器内容装入存储单元)

Add R#，R#　(类似的还有Sub，Mul等；操作数还可“R#，M#”等)

BACK



用高级语言开发程序

- 随着技术的发展，出现了许多高级编程语言
  - 它们与具体机器结构无关
  - 面向算法描述，比机器级语言描述能力强得多
  - 高级语言中一条语句对应几条、几十条甚至几百条指令
  - 有“面向过程”和“面向对象”的语言之分
  - 处理逻辑分为三种结构
    - 顺序结构、选择结构、循环结构
  - 有两种转换方式：“编译”和“解释”
    - 编译程序(Complier)：将高级语言源程序转换为机器级目标程序，执行时只要启动目标程序即可
    - 解释程序(Interpreter)：将高级语言语句逐条翻译成机器指令并立即执行，不生成目标文件。

现在，几乎所有程序员都用高级语言编程，但最终要将高级语言转换为机器语言程序

一个典型程序的转换处理过程

经典的“ hello.c ”C-源程序

```
#include <stdio.h>

int main()
{
printf("hello, world\n");
}
```

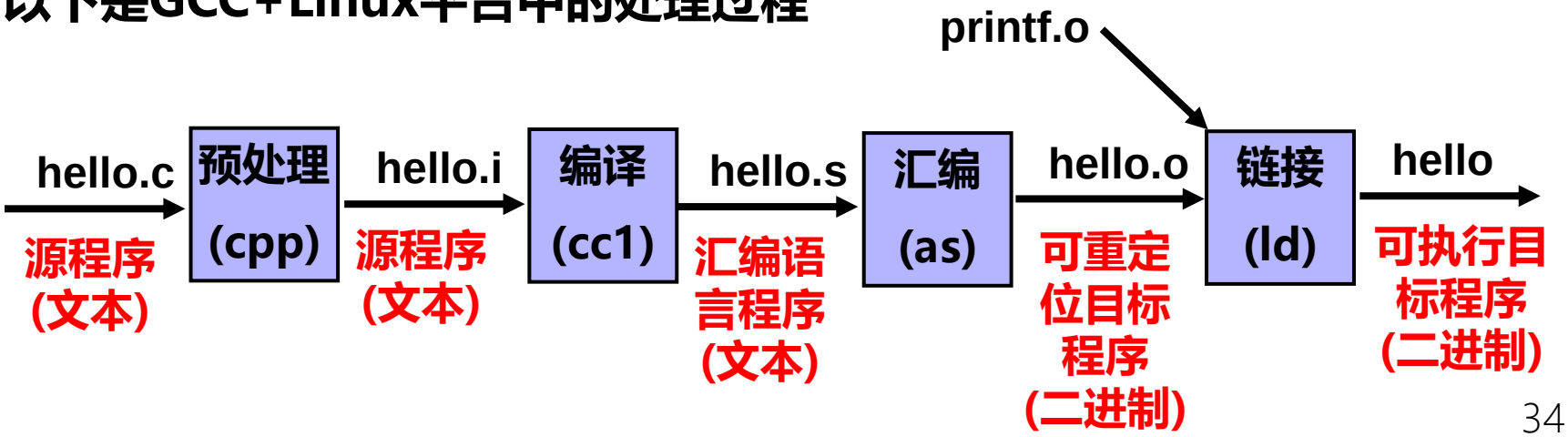
hello.c的ASCII文本表示

```
# i n c l u d e <sp> < s t d i o .
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46
h > \n \n i n t <sp> m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n <sp> <sp> <sp> <sp> p r i n t f ( " h e l
10 32 32 32 32 112 114 105 110 116 102 40 34 104 101 108
l o , <sp> w o r l d \ n " ) ; \n }
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```

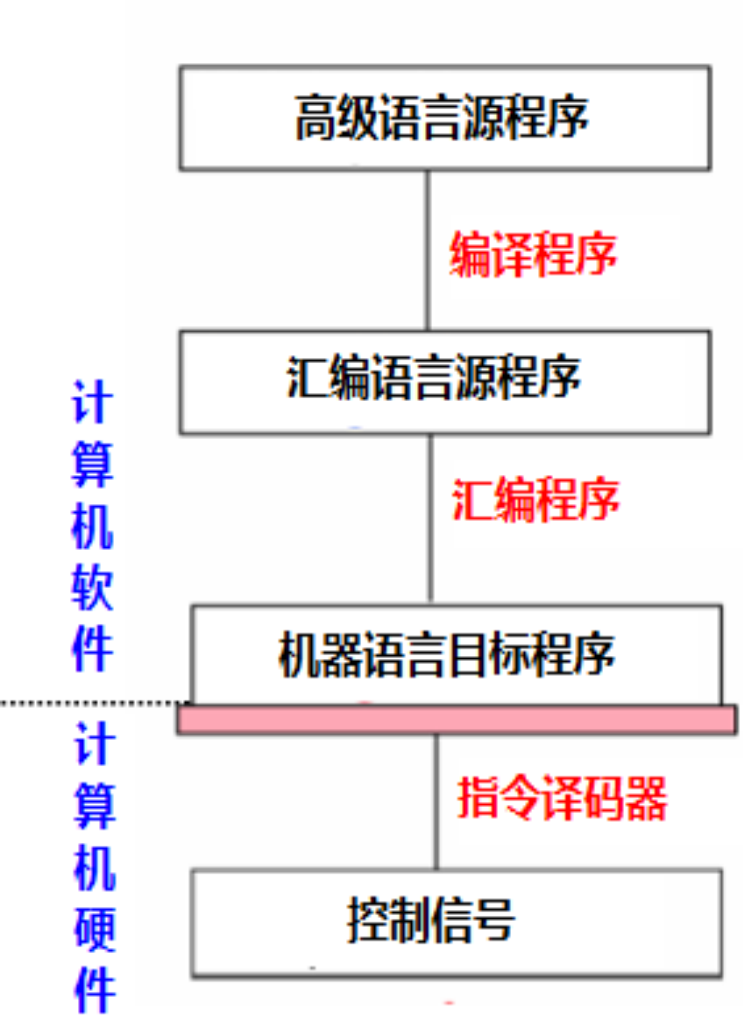
功能：输出“hello,world”

计算机不能直接执行hello.c!

以下是GCC+Linux平台中的处理过程



不同层次语言之间的等价转换




```
tmp = a[i];
a[i] = a[i+1];
a[i+1] = tmp;
```

```
lw $8, 0($2)
lw $9, 4($2)
sw $9, 0($2)
sw $8, 4($2)
```

每条指令由操作码和若干地址码组成

```
100011 00010 01000 0000000000000000
100011 00010 01001 0000000000000100
101011 00010 01001 0000000000000000
101011 00010 01000 0000000000000100
```

…, EXTop=1,ALUSelA=1,ALUSelB=11,ALUop=add, IorD=1,Read,MemtoReg=1,RegWr=1, …

任何高级语言程序最终通过执行若干条指令来完成！

开发和运行程序需什么支撑？

- 最早的程序开发很简单
  - 直接输入指令和数据，启动后把第一条指令地址送PC开始执行
- 用高级语言开发程序需要复杂的支撑环境
  - 需要编辑器编写源程序
  - 需要一套翻译转换软件处理各类源程序
    - 编译方式：预处理程序、编译器、汇编器、链接器
    - 解释方式：解释程序
  - 需要一个可以执行程序的界面（环境）
    - GUI方式：图形用户界面
    - CUI方式：命令行用户界面

语言处理程序

语言处理系统 ＋

人机接口 ＋

操作系统

- 语言的运行时系统
- 操作系统内核
- 指令集体系结构
- 计算机硬件

支撑程序开发和运行的环境由系统软件提供

最重要的系统软件是操作系统和语言处理系统

语言处理系统运行在操作系统之上，操作系统利用指令管理硬件

第一讲 计算机系统概述

- 冯.诺依曼结构计算机
  - 冯.诺依曼结构基本思想
  - 计算机硬件的基本组成
- 程序的表示和执行过程
  - 机器级语言和高级编程语言
  - 翻译程序：汇编、编译、解释
- 计算机系统抽象层
  - 计算机硬件和软件的接口：指令系统
  - 本课程内容在计算机系统中的位置

计算机系统抽象层的转换

功能**转换**：上层是下层的**抽象**，下层是上层的**实现**

底层为上层提供支撑环境！

程序执行结果

不仅依赖

算法、程序编写

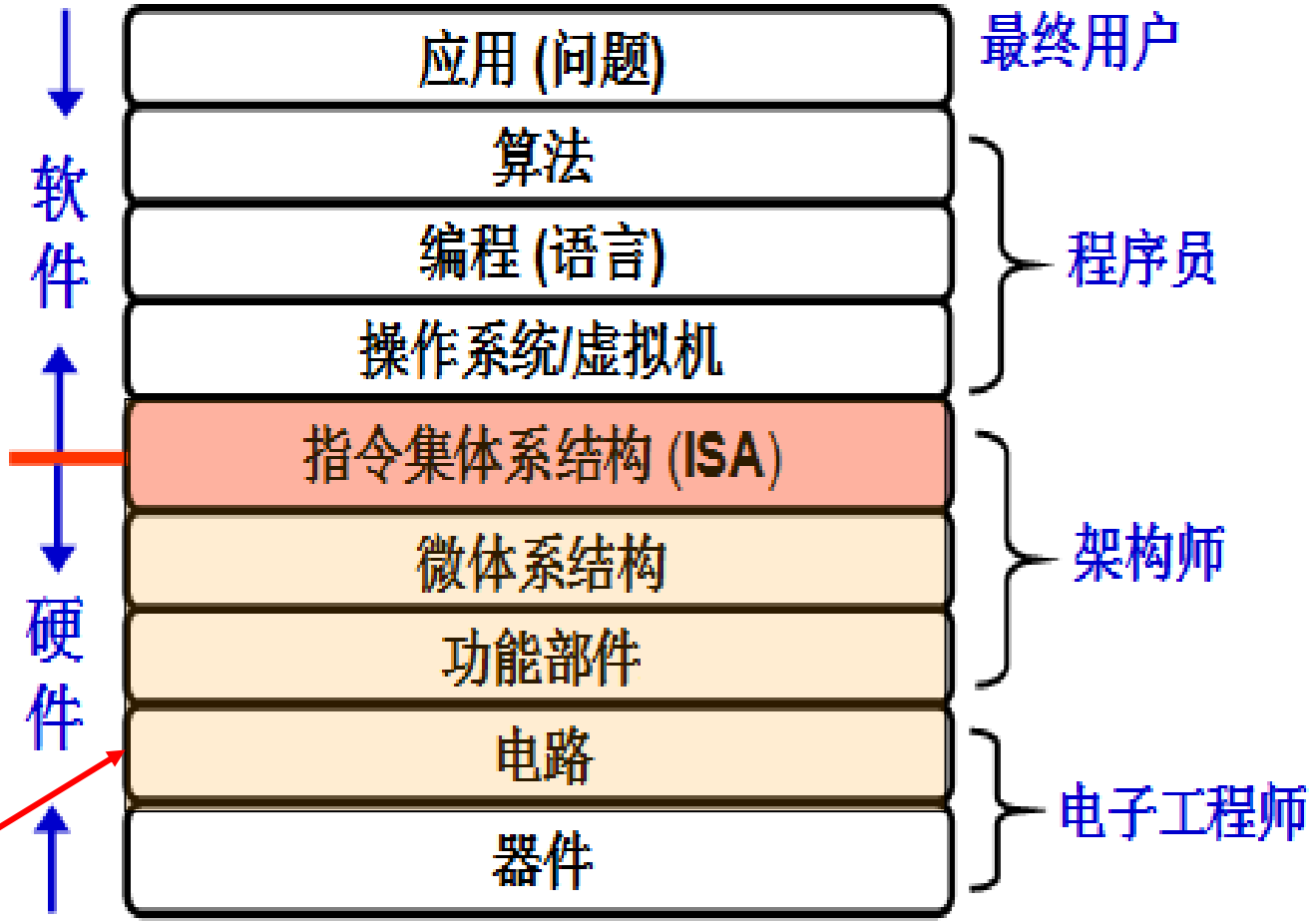
而且依赖

语言处理系统

操作系统

ISA

微体系结构



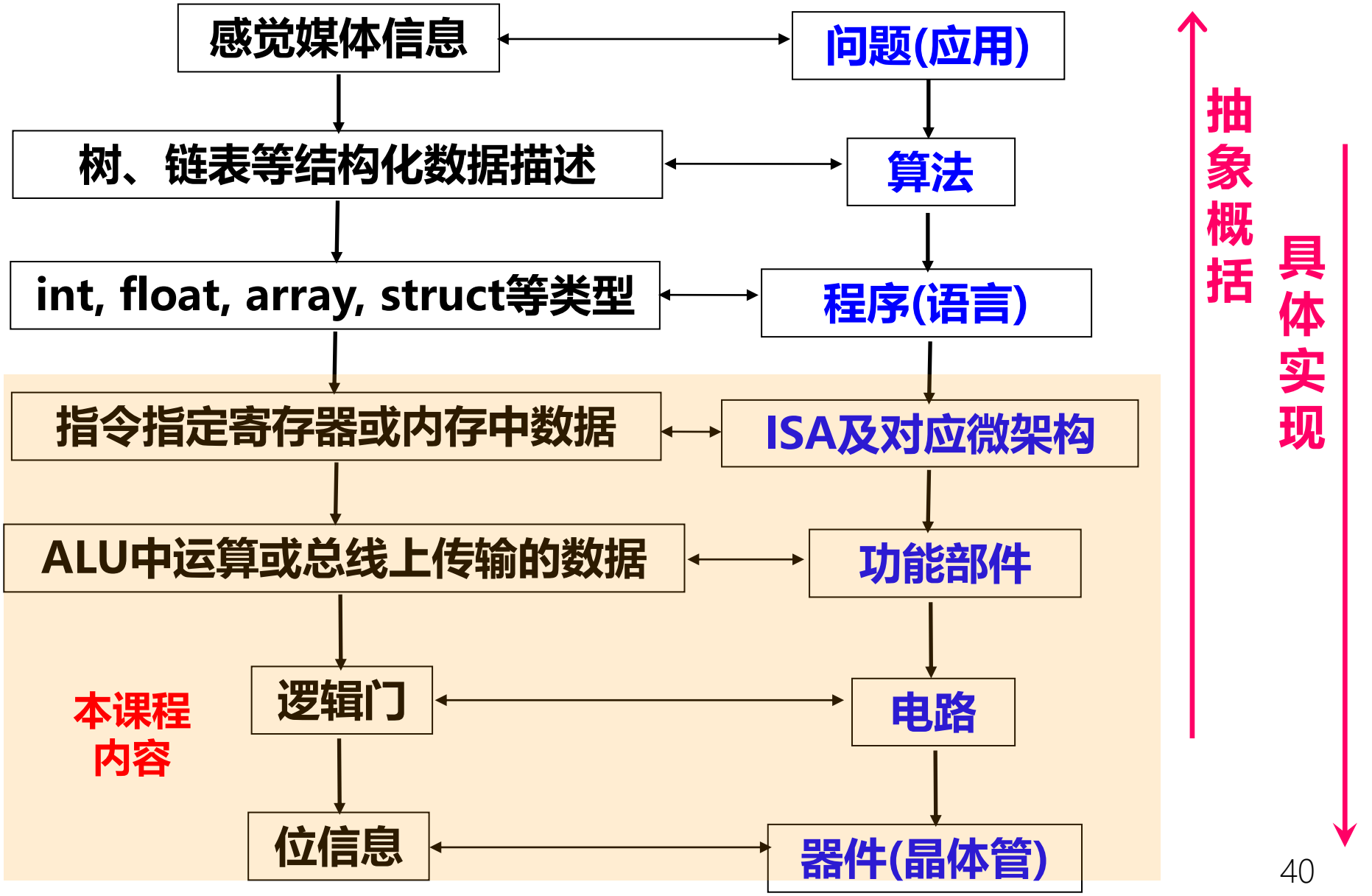
本课程教学内容：数字电路→功能部件→ISA →微架构（CPU、存储器、I/O）

第二讲：二进制数的表示

主　要　内　容

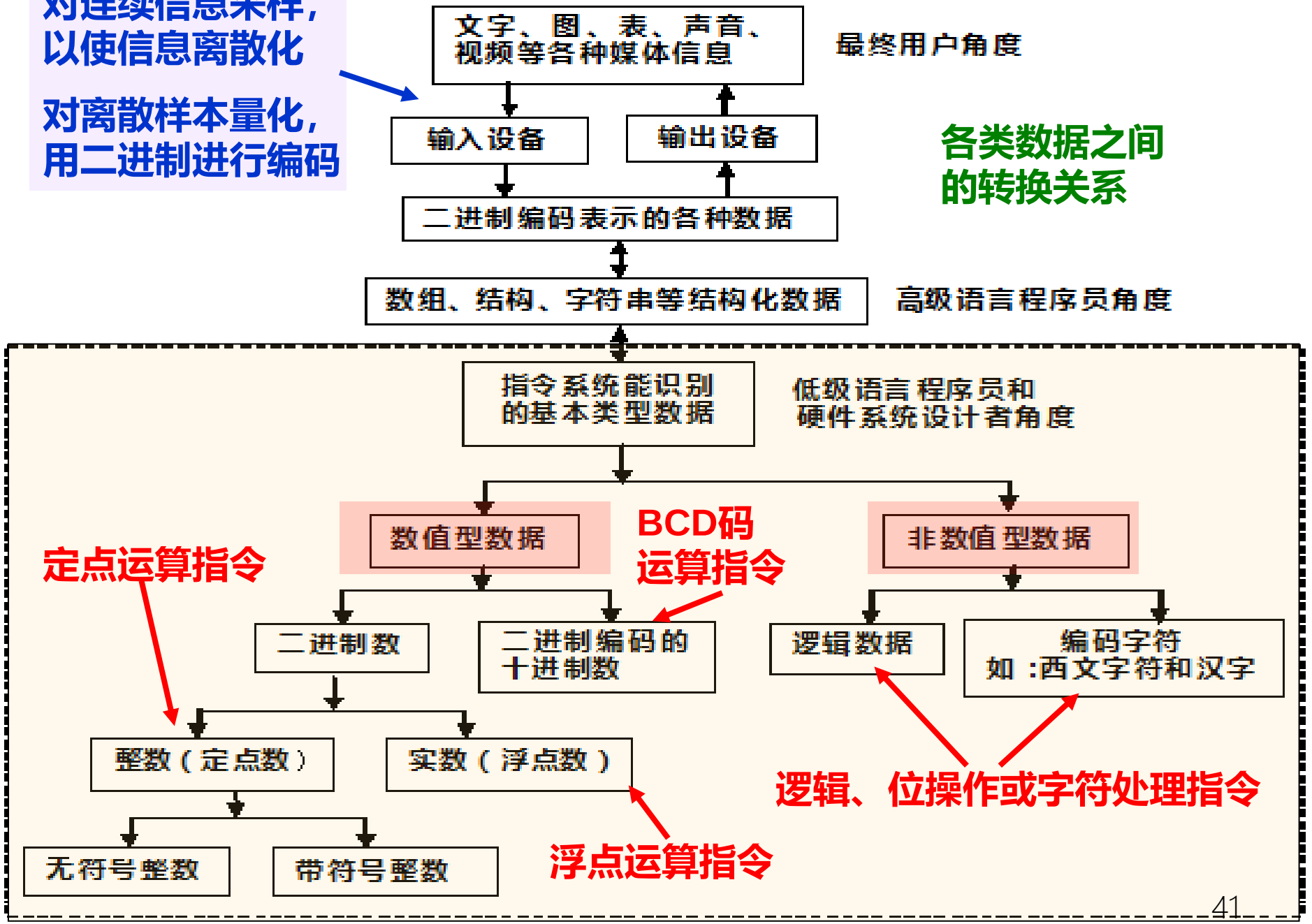
- 计算机的外部信息和内部数据
- 进位计数制
  - 十进制
  - 二进制
  - 八进制和十六进制
- 二进制数与其他计数制数之间的转换
  - R进制数与十进制数之间的转换
  - 二、十六进制数之间的转换
  - 十进制数→二进制数的简便方法

数据表示的抽象层转换


感觉媒体信息
问题(应用)
树、链表等结构化数据描述
算法
int, float, array, struct等类型
程序(语言)
指令指定寄存器或内存中数据
ISA及对应微架构
ALU中运算或总线上传输的数据
功能部件
逻辑门
电路
位信息
器件(晶体管)
本课程内容
抽象概括
具体实现

对连续信息采样，以使信息离散化

对离散样本量化，用二进制进行编码



信息的二进制编码

- 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

Decimal / Binary（十 / 二进制数）

- The decimal number 5836.47 in powers of 10:

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

- The binary number 11001 in powers of 2 :

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

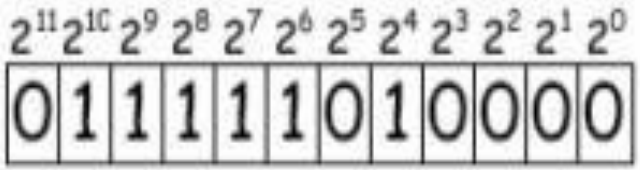
$$= 16 + 8 + 0 + 0 + 1 = 25$$

- 用一个下标表示数的基（ radix / base)

  或用后缀B-二进制（H-十六进制（前缀0x-）、O-八进制）

  $11001_2 = 25_{10}$、 11001B= 25

Octal / Hexadecimal (八 / 十六进制数)



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

$2^3=8$

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10进制或8/16进制数，可缩短长度

早期有用8进制数简便表示2进制数

现在基本上都用16进制数表示机器数

一个8进制数字用3位二进制数字表示

一个16进制数字用4位二进制数字表示

Conversions of numbers

(1) 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

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

Conversions of numbers

(2) R进制数 => 十进制数

按“权”展开 (a power of R)

例1: $(10101.01)_2=1x2^4+1x\ 2^2+1x2^0+1x2^{-2}=(21.25)_{10}$

例2: $(307.6)_8=3x8^2+7x8^0+6x8^{-1}=(199.75)_{10}$

例1: $(3A.\ 1)_{16}=3x16^1+10x16^0+1x16^{-1}=(58.0625)_{10}$

(3) 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”

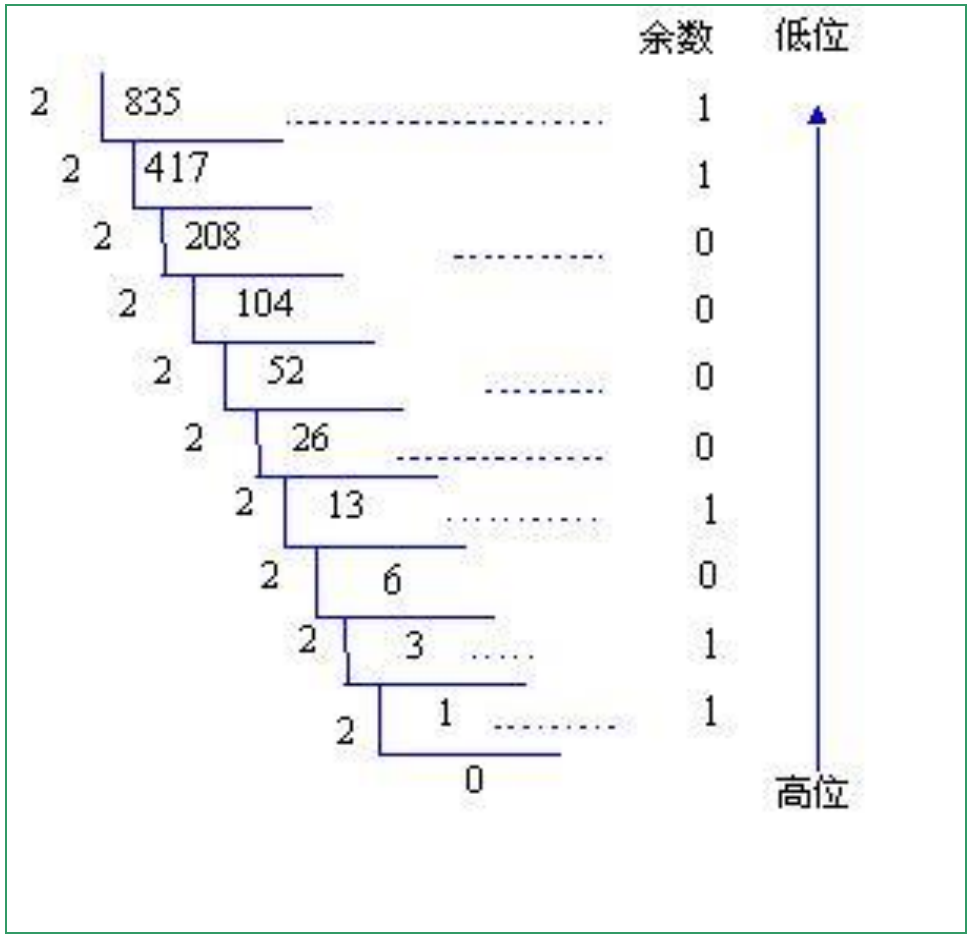
② 小数(fractional part)----“乘基取整，上左下右”

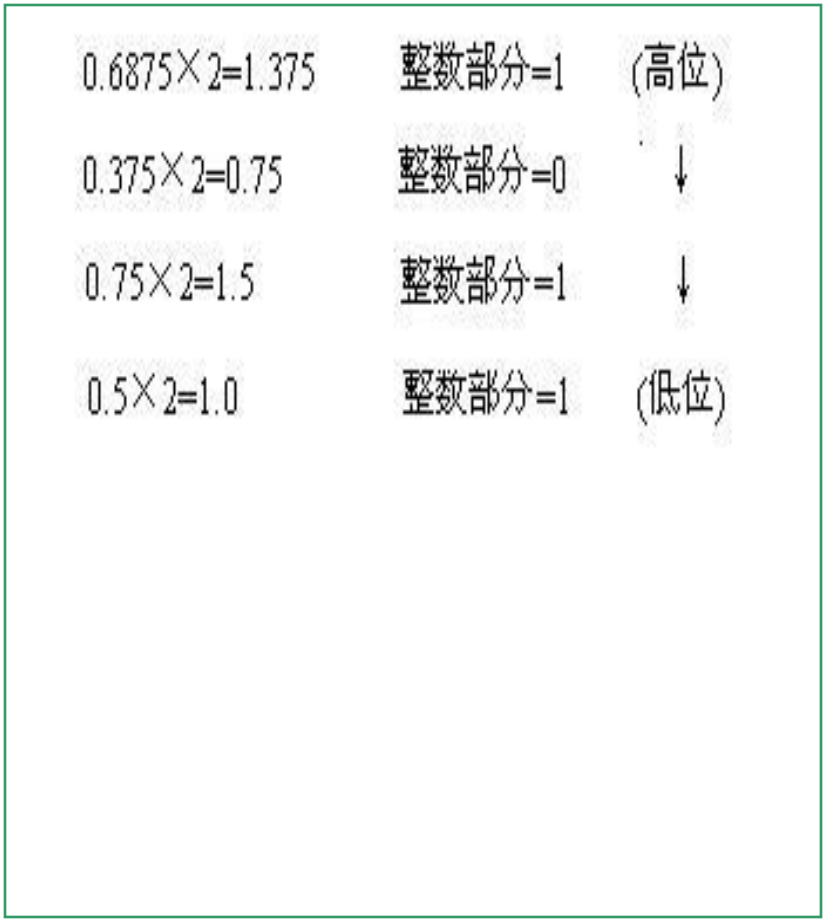
理论上的做法

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”



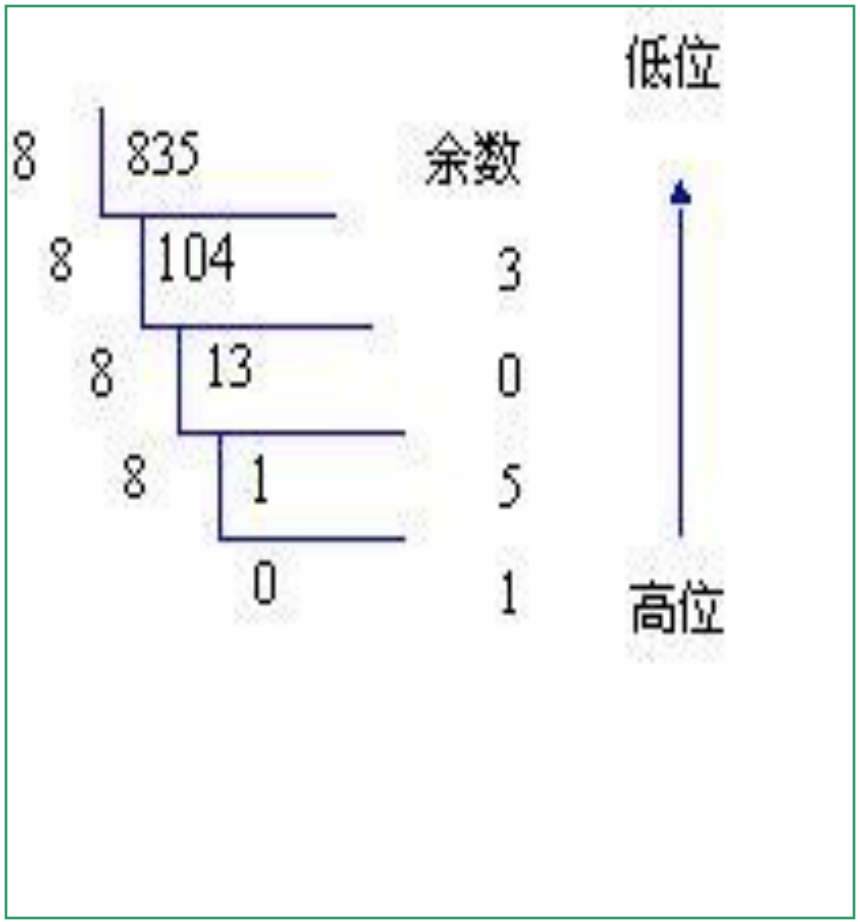


Decimal to Octal Conversions

例2: $(835.63)_{10}=(1503.50243\cdots)_8$

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。



0.63×8=5.04 整数部分=5 (高位)

0.04×8=0.32 整数部分=0

0.32×8=2.56 整数部分=2

0.56×8=4.48 整数部分=4

0.48×8=3.84 整数部分=3 (低位)

Conversions of numbers

(3) 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”

} 理论上的做法

实际按简便方法先转换为二进制数，再按需转换为8/16进制数

整数：2、4、8、16、…、512、1024、2048、4096、…、65536

小数：0.5、0.25、0.125、0.0625、0.03125、……

例：4123.25=4096+16+8+2+1+0.25=1 0000 0001 1011.01B

$=(101B.4)_{16}$

4023=(4096-1)-64-8=1111 1111 1111B-100 0000B-1000B

=1111 1011 0111B = = FB7H=$(FB7)_{16}$

第三讲：数值数据的编码表示

主 要 内 容

- 数值数据的表示方法
  - 定点表示法/浮点表示法
  - 定点数的二进制编码
    - 原码、补码、移码表示
- 整数的表示
  - 无符号整数、带符号整数
- 浮点数的表示
  - 浮点数格式和表示范围
  - 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
- 十进制数的二进制编码（BCD码）

数值数据的表示

- 问题：
  - 用有限个有效数字表示一个数值数据，例如：
    - 用8个二进制数表示数值数据，10101101，1.1101011
    - 用6个十进制数表示数值数据，123456，1234.56

- 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！

数值数据的表示

- 定/浮点表示法

数值数据

定点表示法：指定有效数字序列中的某两位之间是小数点位。 1234.56

浮点表示法：不固定小数点位，数值采用科学计数法表示：$f*r^e$ $1.23456*10^3$

定点表示法：

- 指定在有效数字最右边表示整数，**定点整数**。
- 指定在有效数字最左边表示小数，**定点小数**。

浮点表示法：

- r是进位计数制的基，缺省。
- f和e都可以用**定点表示法**记录。

定点表示法的编码（定点数编码）：

- 增加符号位，解决有符号数表示问题。

具体应用：根据数据类型，选择不同编码方式。

- 整数：定点表示法，定点整数（整型类：int，long，unsigned）
- 实数（浮点数）：定点整数+定点小数（浮点类：float，double）

数值数据的表示

- 定点数的编码
  - 定点整数编码
    - 原码
    - 补码
    - 反码（很少用）
    - 移码
  - 定点小数编码
    - 原码

- 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　　（编码）
  - 真值：机器数真正的值，即：现实中带正负号的数　（数值）

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

格式：符号位+数值位

范围：$-2^{n-1}+1$—$2^{n-1}-1$

- 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

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。

格式：符号位+变换后数值位

范围：-2^{n-1}—$2^{n-1}-1$

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（相当于10+（-4）=6）

② 顺拨8格：10+8 = 18 ≡ 6 (mod 12)

模12系统中： 10- 4 ≡ 10+8 (mod 12)

- 4 ≡ 8 (mod 12)

则，称8和- 4对模12运算是等价的，可以执行同样运算。

同样有 -3 ≡ 9 （mod 12）

-5 ≡ 7 （mod 12）等

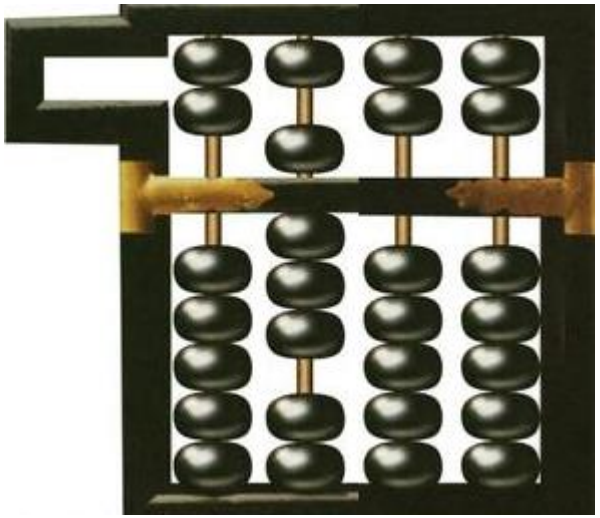
结论1： 一个负数的补码等于模减该负数的绝对值。

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？

$10-4 = 10+(12-4) = 10+8 = 6 \quad (\text{mod } 12)$

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？

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

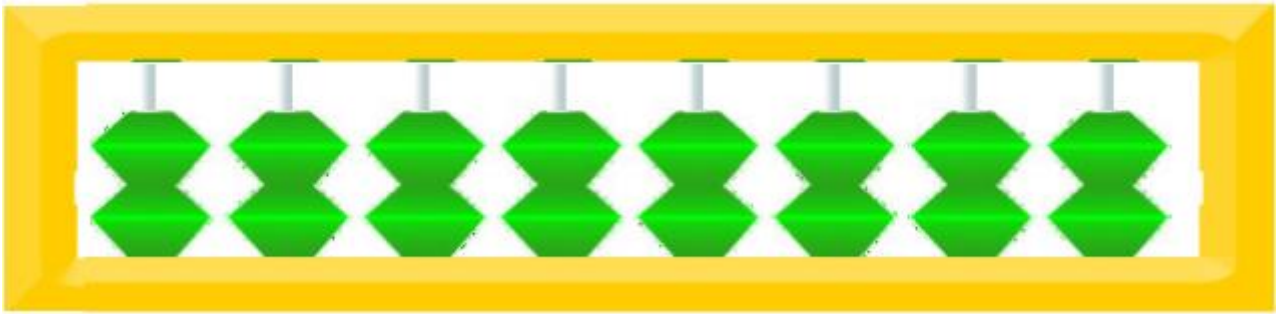
$=9828+8072$

$=\boxed{1}7900$

$=7900 \ (\text{mod } 10^4)$

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 = 1 0011 1111 (mod 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的“各位取反、末位加1”

运算器是一个模运算系统

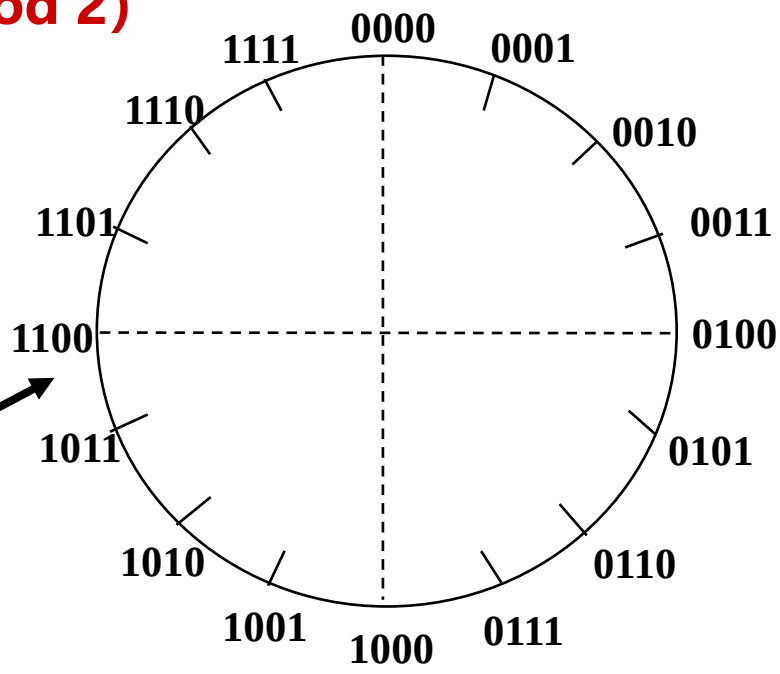
计算机中运算器只有有限位。假定为n位，则运算结果只能保留低n位，故可看成是个只有n档的二进制算盘。所以，其模为2^n。

补码的定义　假定补码有n位，则：

定点整数：$[X]_{补}= 2^n + X$　$(-2^{n-1} \le X < 2^{n-1}, \mod 2^n)$

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

注：实际上在计算机中并不使用补码的定点小数表示！不需要掌握这个知识点。



当n=4时，共有16个机器数：

0000 ~ 1111，可看成是模为2^4的钟表系统。真值的范围为

-8 ~ +7

求特殊数的补码

假定机器数有n位

① $[-2^{n-1}]_{补} = 2^n - 2^{n-1} = 10\ldots0$（n-1个0） （mod 2^n）

② $[-1]_{补} = 2^n - 0\ldots01 = 11\ldots1$（n个1） （mod 2^n）

③ $[-1.0]_{补} = 2 - 1.0 = 1.00\ldots0$（n-1个0） （mod 2）

④ $[+0]_{补} = [-0]_{补} = 00\ldots0$（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 = 01111111B - 100B = 01111011B

-123= - 01111011B

$[01111011]_{补}= 2^8 + 01111011 = 100000000 + 01111011$

$= 01111011 \pmod{2^8}$，即 7BH。

$[-01111011]_{补}= 2^8 – 01111011 = 10000\ 0000 - 01111011$

= 1111 1111 – 0111 1011 +1

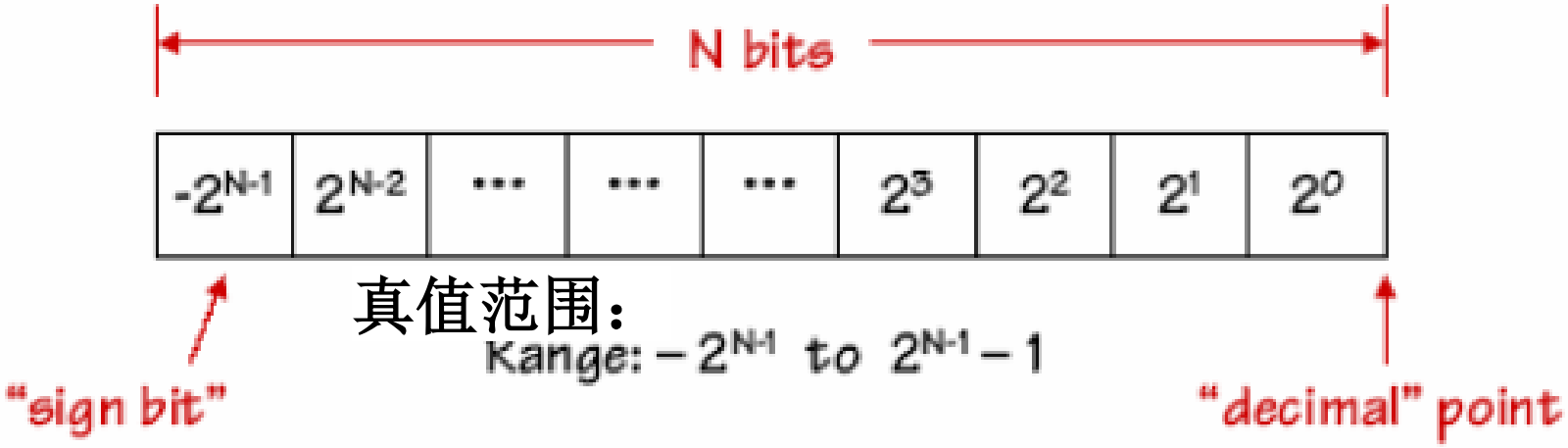
= 1000 0100 +1 ⟵ 各位取反，末位加1

= 1000 0101，即 85H。

如何求补码的真值

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

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



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，数值部分相同

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: $E_{biased} = E + 2^3$ （bias= 2^3 =1000B）

  -8 (+8) ~ 0000B

  -7 (+8) ~ 0001B

  …

  0 (+8) ~ 1000B

  …

  +7 (+8) ~ 1111B

格式：符号位+数值位

范围：-2^{n-1}—$2^{n-1}-1$

0的移码表示唯一

移码和补码仅第一位不同

移码主要用来表示浮点数阶码！

- 为什么要用移码来表示指数（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)

第三讲：数值数据的编码表示

主 要 内 容

- 数值数据的表示方法
  - 定点表示法/浮点表示法
  - 定点数的二进制编码
    - 原码、补码、移码表示
- 整数的表示
  - 无符号整数、带符号整数
- 浮点数的表示
  - 浮点数格式和表示范围
  - 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
- 十进制数的二进制编码（BCD码）

具体应用：根据数据类型，选择不同编码方式。

- 整数：定点表示法，定点整数
- 实数（浮点数）：定点整数+定点小数

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: $0\ldots01011_2$）
  - 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场合下，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）

- 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）

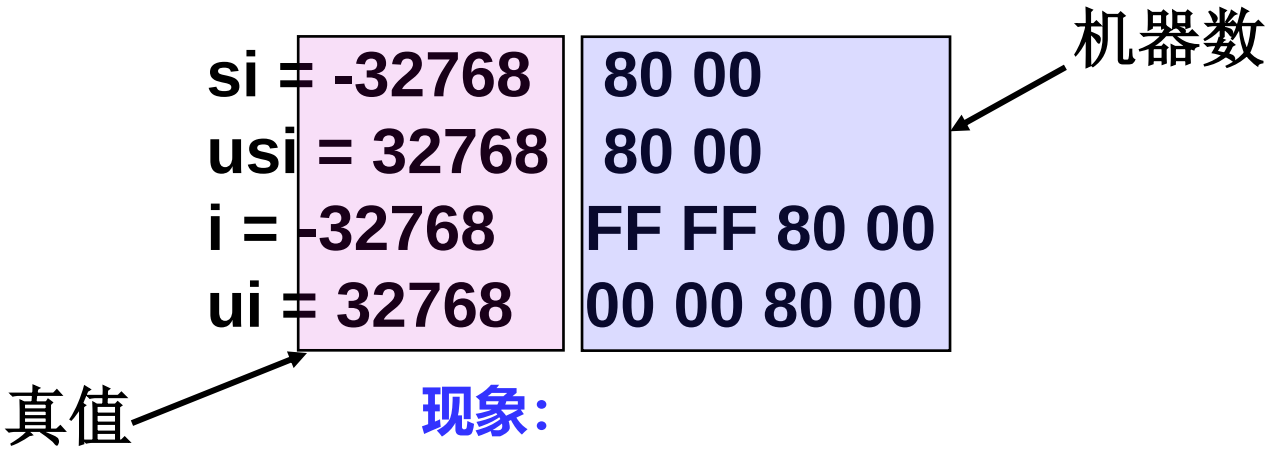
    50年代以来，所有计算机都用补码来表示定点整数
- 为什么用补码表示带符号整数？
  - 补码运算系统是模运算系统，加、减运算统一
  - 数0的表示唯一，方便使用
  - 比原码和反码多表示一个最小负数
  - 与移码相比，其符号位和真值的符号对应关系清楚

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;
unsingned  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！

第三讲：数值数据的编码表示

主 要 内 容

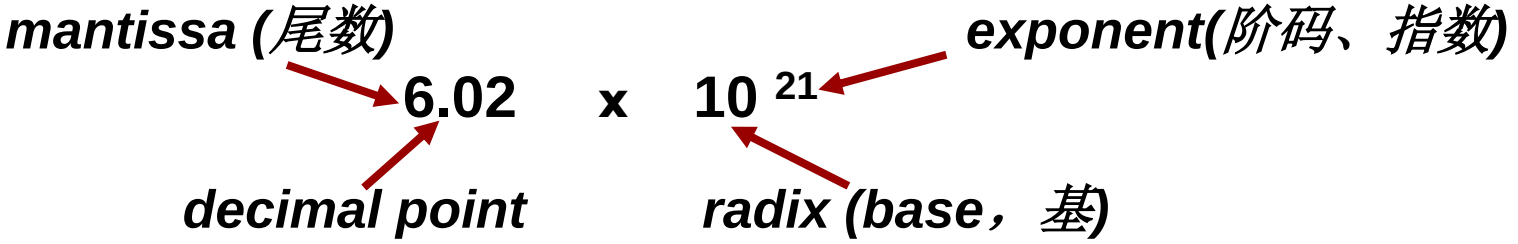
- 数值数据的表示方法
  - 定点表示法/浮点表示法
  - 定点数的二进制编码
    - 原码、补码、移码表示
- 整数的表示
  - 无符号整数、带符号整数
- 浮点数的表示
  - 浮点数格式和表示范围
  - IEEE754浮点数标准
    - 单精度浮点数、双精度浮点数
    - 特殊数的表示形式
- 十进制数的二进制编码（BCD码）

具体应用：根据数据类型，选择不同编码方式。

- 整数：定点表示法，定点整数
- 实数（浮点数）：定点整数+定点小数

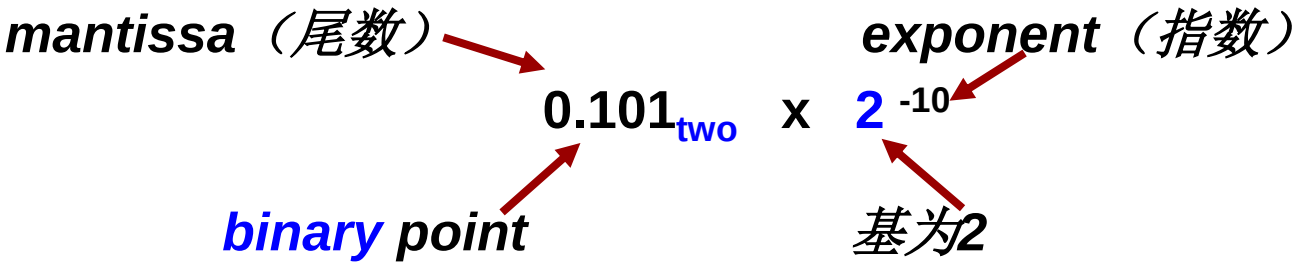
科学计数法(Scientific Notation)与浮点数

Example:



- **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}**

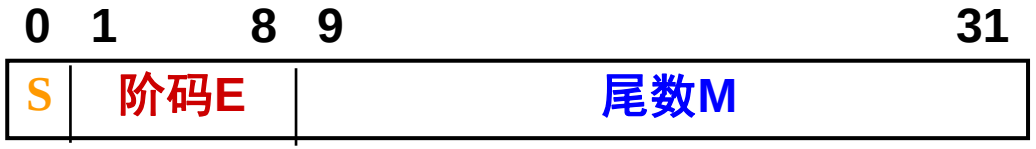
for Binary Numbers:



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。

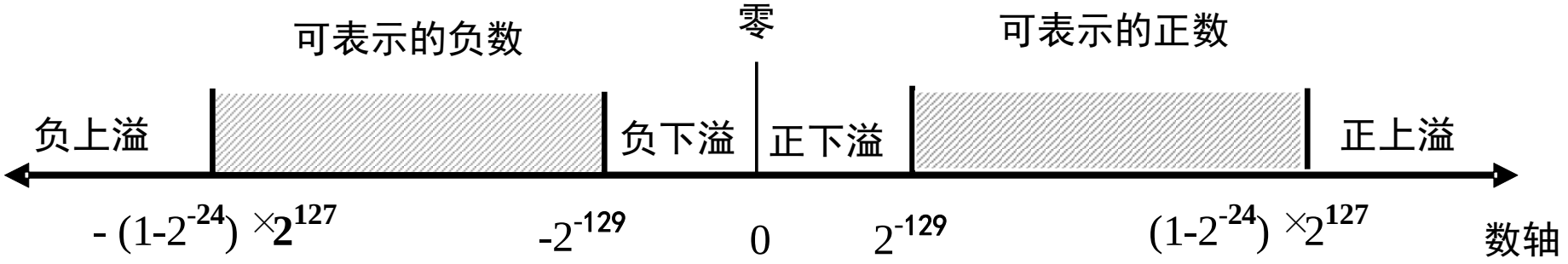


$+/-0.1xxxxx \times 2^{E}$

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：$0.11\ldots1 \times 2^{11\ldots1} = (1-2^{-24}) \times 2^{127}$　最小正数：$0.10\ldots0 \times 2^{00\ldots0} = (1/2) \times 2^{-128}$

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，也不连续

浮点数的表示

- Normal format（规格化数形式）：

  +/-1.xxxxxxxxxx × $2^{Exponent}$

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！

- 32-bit 规格化数：

S 是符号位（Sign）

Exponent用移码（增码）来表示

Significand 表示 xxxxxxxxxxxxx，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？

“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.

1989
ACM Turing
Award Winner!

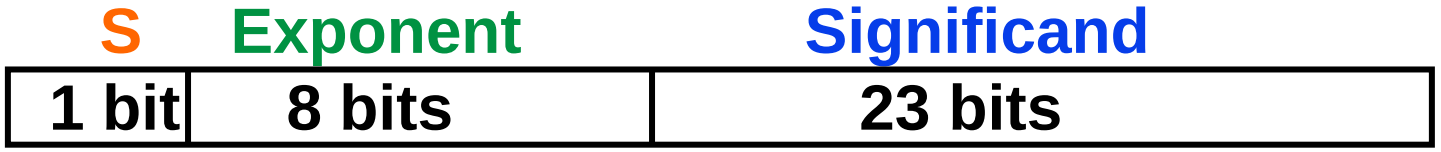
www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Prof. William Kahan

IEEE 754 Floating Point Standard

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

Single Precision： (Double Precision is similar)



- 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，8位), 1023 (double，11位)
- Significand（尾数）：
  - 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)

为什么用127？若用128，则阶码范围为多少？

0000 0001 (-127) ~ 1111 1110 (126)

SP: $(-1)^S$ x (1 + Significand) x $2^{(Exponent-127)}$

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

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:**
  - **$0111\ 1101_{two} = 125_{ten}$**
  - **Bias adjustment: 125 - 127 = -2**
- **Significand:**

  $1 + 1\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 0\times2^{-4} + 0\times2^{-5} + \ldots$

  $= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$
- **Represents: $-1.75_{ten} \times 2^{-2} = -0.4375$**

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

1	1000 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 00000000000000000000000

-0: 1 00000000 00000000000000000000000

Representation for +∞/-∞

∞ : infinity

In FP, 除数为0的结果是 +/- ∞, 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+∞/-∞作比较。 例如：X/0>Y可作为有效比较

How to represent +∞/-∞?

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

+∞ : 0 11111111 00000000000000000000000

-∞ : 1 11111111 00000000000000000000000

Operations

5.0 / 0 = +∞, -5.0 / 0 = -∞

5+(+∞) = +∞, (+∞)+(+∞) = +∞

5 - (+∞) = -∞, (-∞) - (+∞) = -∞ etc

浮点数除0的问题

这是网上的一个帖子

```
#include <conio.h>
#include <stdio.h>
int    main()
{
        int    a=1,b=0;
        printf( "Division    by    zero:%d\n ",a/b);
        getchar();
        return    0;
}
int    main()
{
        double    x=1.0,y=-1.0,z=0.0;
        printf( "division    by    zero:%f    %f\n ",x/z,y/z);
        getchar();
        return    0;
}
```

为什么整数除0会发生异常?

为什么浮点数除0不会出现异常?

浮点运算中，一个有限数除以0，
结果为正无穷大（负无穷大）

问题一:为什么整除int型会产生错误? 是什么错误?

二:用double型的时候结果为1.#INF00和-1.#INF00，作何解释???

Representation for “Not a Number”

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

sqrt (-4.0) = NaN 0/0 = NaN

op (NaN,x) = NaN +∞+(-∞) = NaN

+∞- (+∞) = NaN ∞/∞ = NaN

etc.

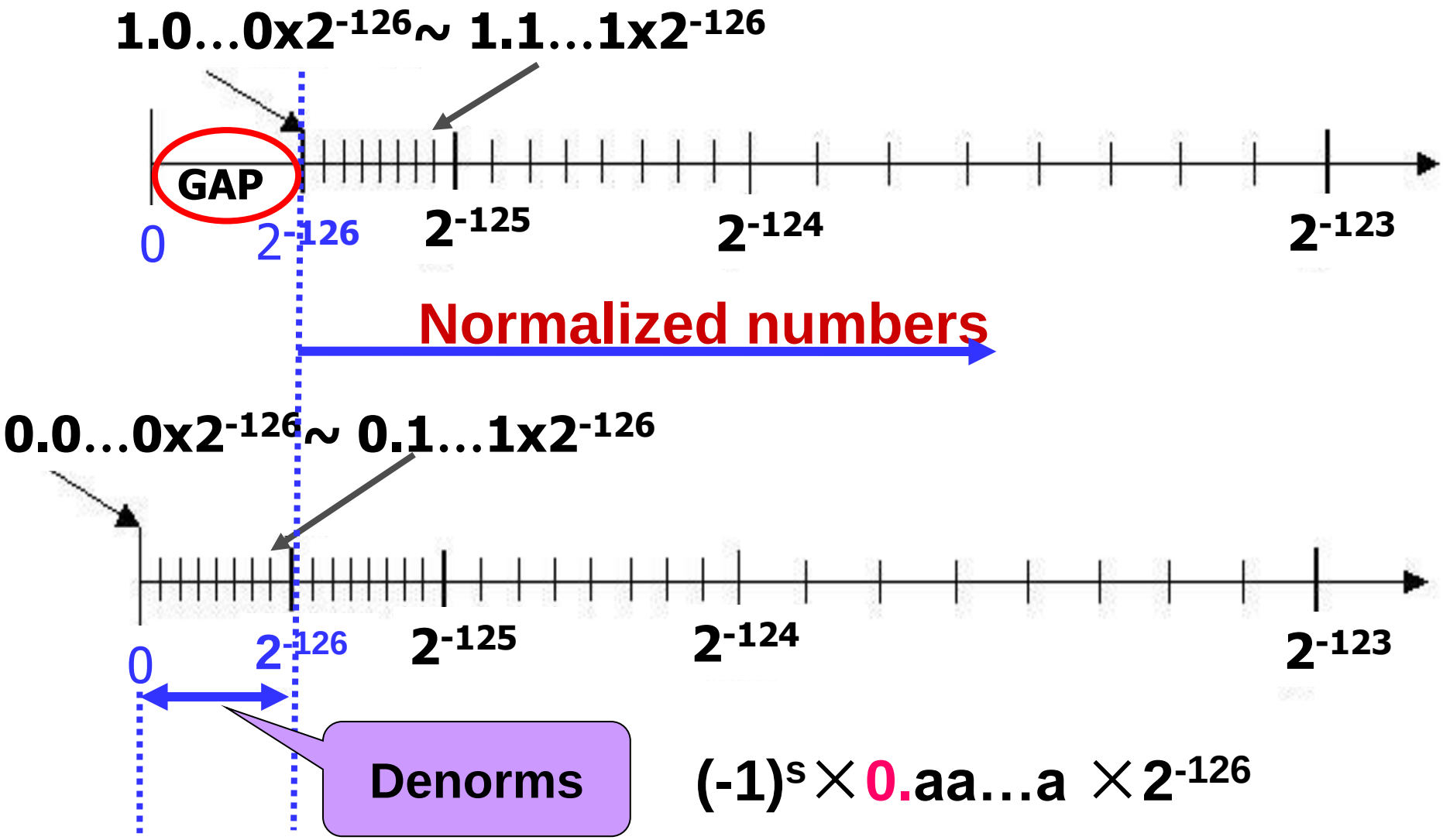
Representation for Denorms(非规格化数)

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

Used to represent Denormalized numbers

Representation for Denorms



非规格化浮点数举例

当结果为 0.1×2^{-126} 时，是用非规格化数表示还是近似为0?

以下程序试图计算 $2^{-63}/2^{64}=2^{-127}$

```
#include <stdio.h>

main()
{ float x=1.084202172485504e-19;
  float y=1.844674407370955e+19;
  float z=x/y;
  printf("x=%f %x\n",x,x);
  printf("y=%f %x\n",y,y);
  printf("z=%f %x\n",z,z);
}
```

```
user@debian:~/Templates$ ./denom
x=0.000000 0
y=18446744073709551616.000000 0
z=0.000000 0
```

计算器

2^{-63}

2^{64}

讨论问题:

1. **计算器上算的准确吗?**
2. **为什么 x 输出为 0?**
3. **为什么 y 的输出发生变化?**
4. **为什么x、y、z用%x输出为0?**
5. **z 输出为 0 说明了什么?**
6. **如下赋初值对否?**

   float x=0x40000000;

   float y=0x5f800000;

非规格化浮点数举例

```
#include <stdio.h>

main()
{  float x=0.000015258789062 5;
   float y=0.000030517578125;
   float z=x*x*x*y;
   float l=1.84467440737095516 16e+19;
   float m=z/l;
   printf("z=%e \n",z);
   printf("m=%1.38e \n",m);
   //printf("z=%f %x\n",z,z);
}
```

2^{-16}

2^{-15}

当结果为 $0.1x2^{-126}$ 时，用非规格化数表示，而不是近似表示成0!

z和m的输出结果说明了什么?

```
user@debian:~/Templates$ gcc -o denom denom.c
user@debian:~/Templates$ ./denom
z=1.084202e-19
m=5.87747175411143753984368268611122838909e-39
```

Questions about IEEE 754

- What's the range of representable values?

  The largest number for single: +1.11…1x 2^{127} 约 +3.4 x 10^{38}

  How about double? 约 +1.8 x 10^{308}

- What about following type converting: not always true!

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

How about double? True!

How about double? Not always true!

- How about FP add associative? FALSE!

  x = – 1.5 x 10^{38}, y = 1.5 x 10^{38}, z = 1.0

  (x+y)+z = (–1.5x10^{38}+1.5x10^{38}) +1.0 = 1.0

  x+(y+z) = –1.5x10^{38}+ (1.5x10^{38}+1.0) = 0.0

浮点数舍入举例

例：将同一实数分别赋值给单精度和双精度类型变量，然后打印输出。

```
#include <stdio.h>
main()
{
        float a;
        double b;
        a = 123456.789e4;
        b = 123456.789e4;
        printf( “%f/n%f/n” ,a,b);
}
```

运行结果如下：

```
1234567936.000000
1234567890.000000
```

为什么float情况下输出的结果会比原来的大？这到底有没有根本性原因还是随机发生的？为什么会出现这样的情况？

float可精确表示7个十进制有效数位，后面的数位是舍入后的结果，舍入后的值可能会更大，也可能更小

问题：为什么同一个实数赋值给float型变量和double型变量，输出结果会有所不同呢？

第三讲：数值数据的编码表示

主 要 内 容

- 数值数据的表示方法
  - 定点表示法/浮点表示法
  - 定点数的二进制编码
    - 原码、补码、移码表示
- 整数的表示
  - 无符号整数、带符号整数
- 浮点数的表示
  - 浮点数格式和表示范围
  - 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
- 十进制数的二进制编码（BCD码）

用BCD码表示十进制数

- 编码思想： 每个十进数位至少需要4位二进制表示。而4位二进制位可组合成16种编码，去掉10种编码后还有6种冗余编码。
- 编码方案
  1. 十进制有权码
     - 每个十进制数字的4个二进制位（称为基2码）都有确定的权。
       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。
     - 用的较多的是余3码和格雷码。
     - 余3码方案：由8421码加上0011形成。当两个十进制数字之和是10时，其二进制编码的值正好是16，而且0和9，1和8，…，5和4的余3码互为反码。
     - 格雷码（Gray Code）方案：任意两个相邻的编码只有一位二进位不同。格雷码有多种编码形式。
  3. 其他编码方案 （5中取2码、独热码等）

第三讲小结

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
- 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)
- 十进制数的二进制表示（BCD码）
  - 有权BCD码（8421码）、无权BCD码（余3码、格雷码等）

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

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，机器靠指令来认定。
- 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：

西文字符的编码表示

- 特点：{字符集，编码方案，字形}
  - 是一种拼音文字，用有限几个字母可拼写出所有单词
  - 只对
  - 所有
- 表示（常
  - 十进
  - 英文
  - 专用
  - 控制
- 操作
  - 字符串操作，如:传送/比较 等

$b_3b_2b_1b_0$	Row (hex)	$b_6b_5b_4$ (column) 000 0	001 1	010 2	011 3	100 4	101 5	110 6	111 7
0000	0	NUL	DLE	SP	0	@	P	`	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[	k	{
1100	C	FF	FS	,	<	L	\	l	|
1101	D	CR	GS	-	=	M	]	m	}
1110	E	SO	RS	.	>	N	^	n	~
1111	F	SI	US	/	?	O	_	o	DEL

汉字及国际字符的编码表示

NJUAI

- 特点:{字符集，编码方案，字形}
  - 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
  - 字模点阵码或轮廓描述: 描述汉字字模的点阵或轮廓，用于输出

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，已经逐步进入实用程度。

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

其内码就是ASCII码。

问题：西文字符常用的内码是什么?

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

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ACSII码相同，后面的73H和字符“s”的ACSII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)，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。

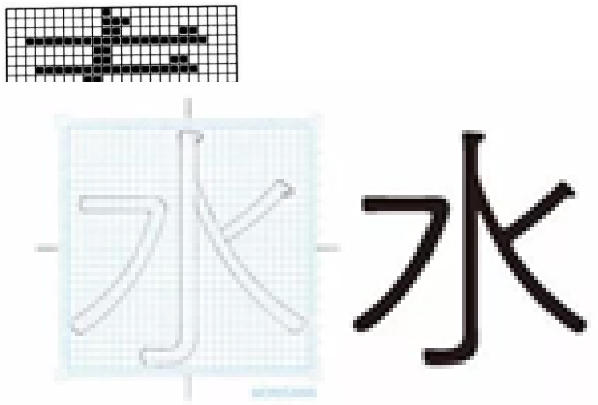
汉字的字模点阵码和轮廓描述

- 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：
  - 字模的点阵描述（图像方式）
  - 字模的轮廓描述（图形方式）
    - 直线向量轮廓
    - 曲线轮廓（True Type字形）

区位码←国标码←内码



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内部定点运算部件的位数、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
- 主存容量经常使用的单位，如：
  - “千字节”（KB），1KB=2^{10}字节=1024B
  - “兆字节”（MB），1MB=2^{20}字节=1024KB
  - “千兆字节”（GB），1GB=2^{30}字节=1024MB
  - “兆兆字节”（TB），1TB=2^{40}字节=1024GB
- 主频和带宽使用的单位，如：
  - “千比特/秒”（kb/s），1kbps=10^3 b/s=1000 bps
  - “兆比特/秒”（Mb/s），1Mbps=10^6 b/s =1000 kbps
  - “千兆比特/秒”（Gb/s），1Gbps=10^9 b/s =1000 Mbps
  - “兆兆比特/秒”（Tb/s），1Tbps=10^{12} b/s =1000 Gbps
- 硬盘和文件使用的单位
  - 不同的硬盘制造商和操作系统用不同的度量方式，因而比较混乱
  - 为避免歧义，国际电工委员会（IEC）在1998年给出了表示2的幂次的二进制前缀字母定义

数据量的度量单位

- 硬盘和文件使用的单位
  - 不同的硬盘制造商和操作系统用不同的度量方式，因而比较混乱
  - 为避免歧义，国际电工委员会（IEC）给出了二进制前缀字母定义，可用不同的前缀表示所采用的度量方式

十进制前缀			IEC 定义的二进制前缀			值差
单词	前缀	值	单词	前缀	值	（%）
kilobyte	KB/kB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

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)

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

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。

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

- 80年代开始，几乎所有通用机器都用字节编址
- 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

$65535=2^{16}-1$

$[-65535]_{补}=FFFF0001H$

例如，若 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
103	102	101	100
msb			lsb
100	101	102	103

little endian word 100#

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

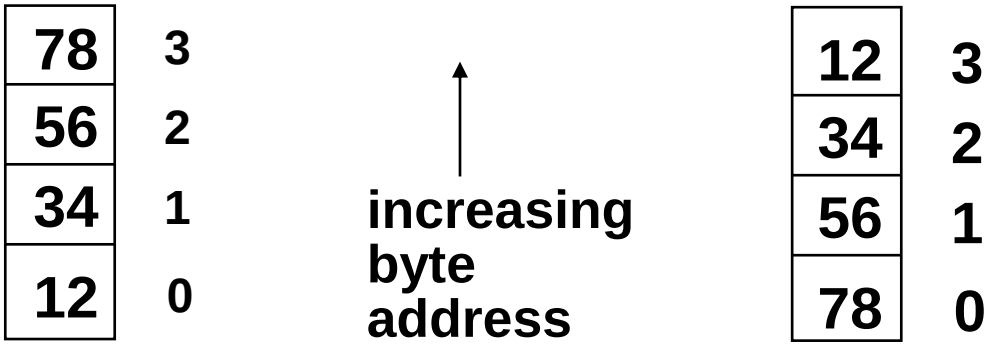
In Big Endian: →

00	1000
AB	1001
CD	1002
EF	1003

In Little Endian: →

00	1003
AB	1002
CD	1001
EF	1000

Byte Swap Problem（字节交换问题）



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

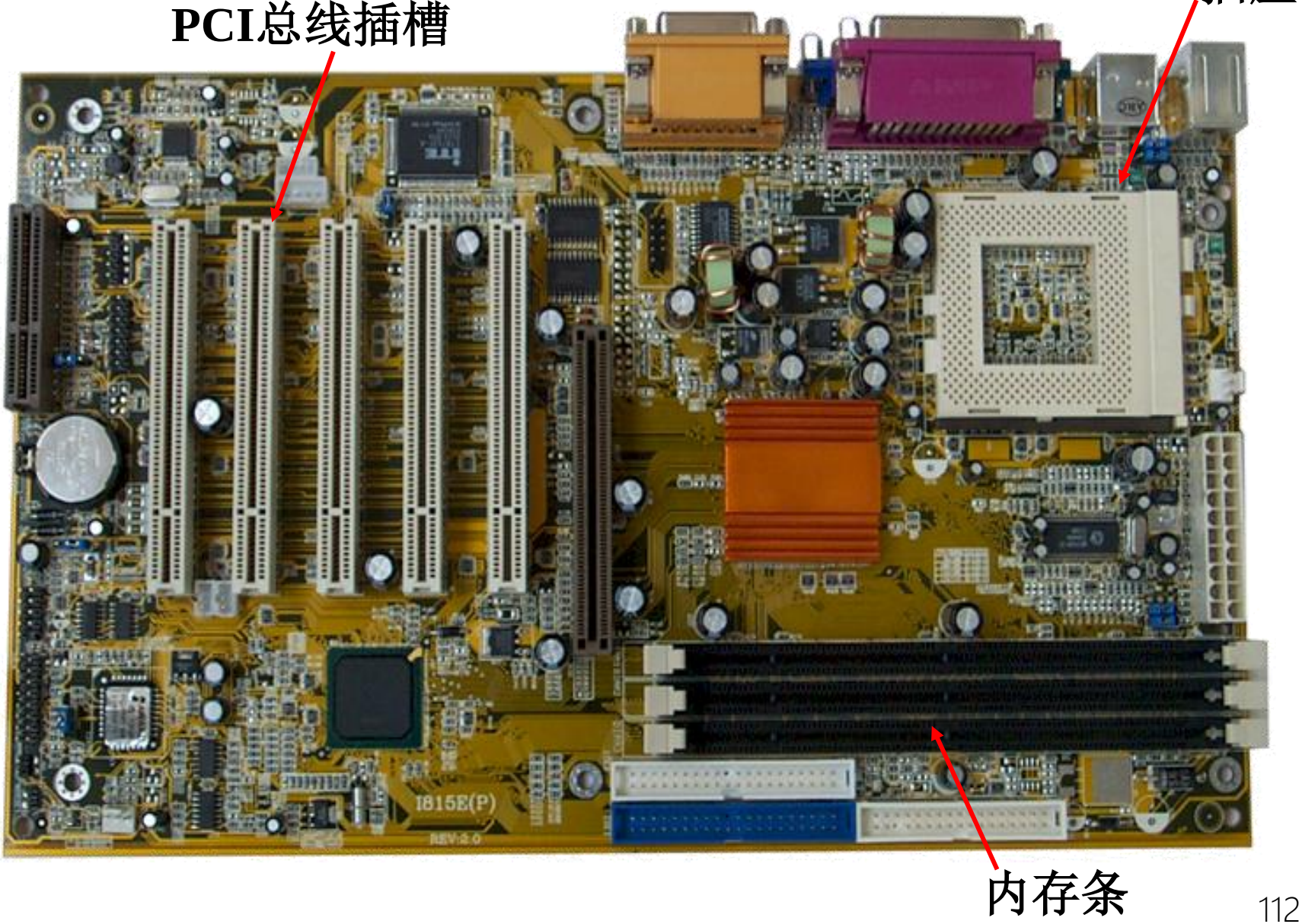
Big endian: Adobe Photoshop, JPEG, MacPaint, etc

第四讲小结

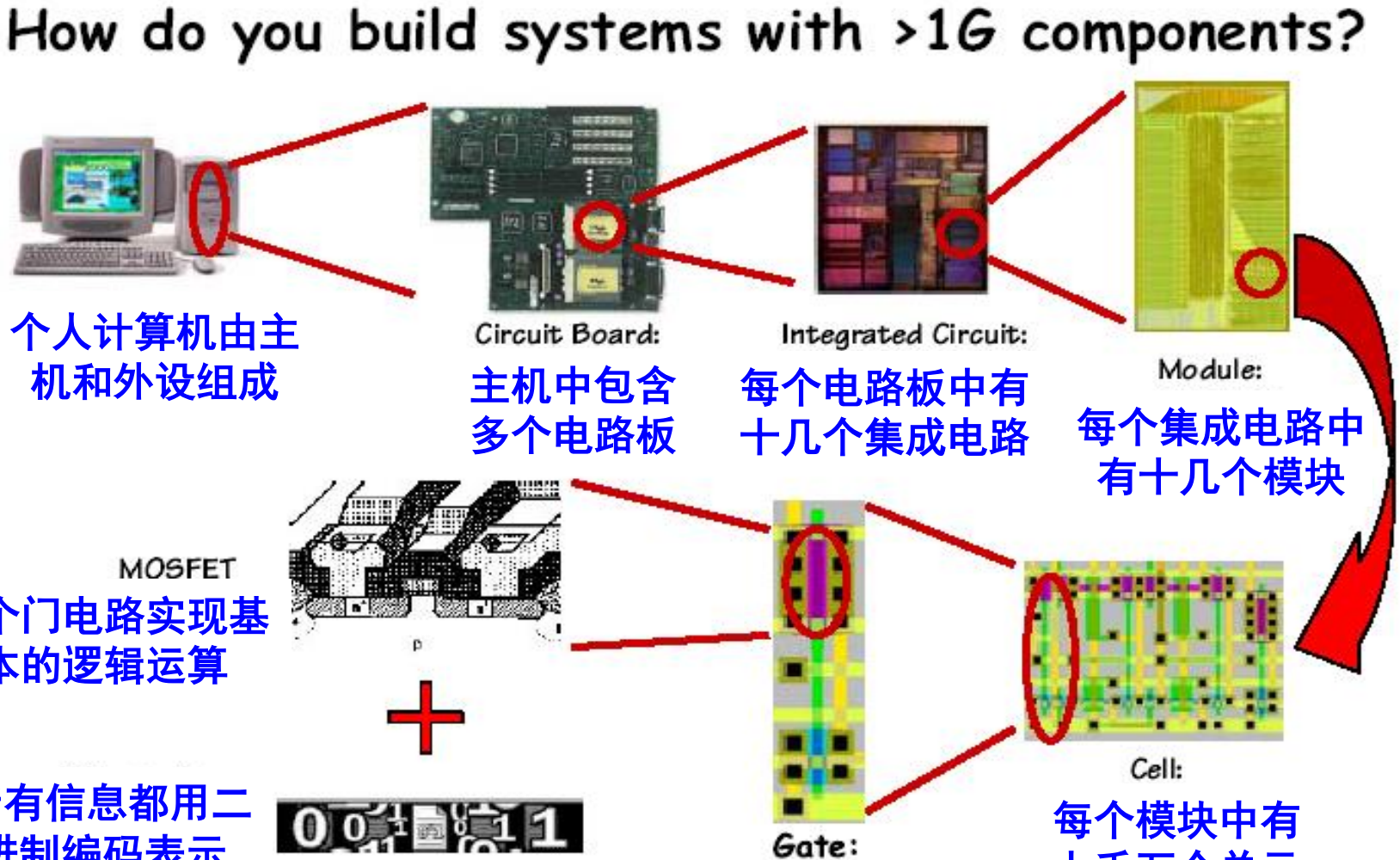
- 非数值数据的表示
  - 逻辑数据用来表示真/假或N位位串，按位运算
  - 西文字符：用ASCII码表示
  - 汉字：汉字输入码、汉字内码、汉字字模码
- 数据的宽度
  - 位、字节、字（不一定等于字长），k/K/M/G/…有不同的含义
- 数据的存储排列
  - 数据的地址：连续若干单元中最小的地址，即：从小地址开始存放数据
    - 问题：若一个short型数据si存放在单元0x08000100和0x08000101中，那么si的地址是什么?
  - 大端方式：用MSB存放的地址表示数据的地址
  - 小端方式：用LSB存放的地址表示数据的地址

计算机硬件：打开PC来看看

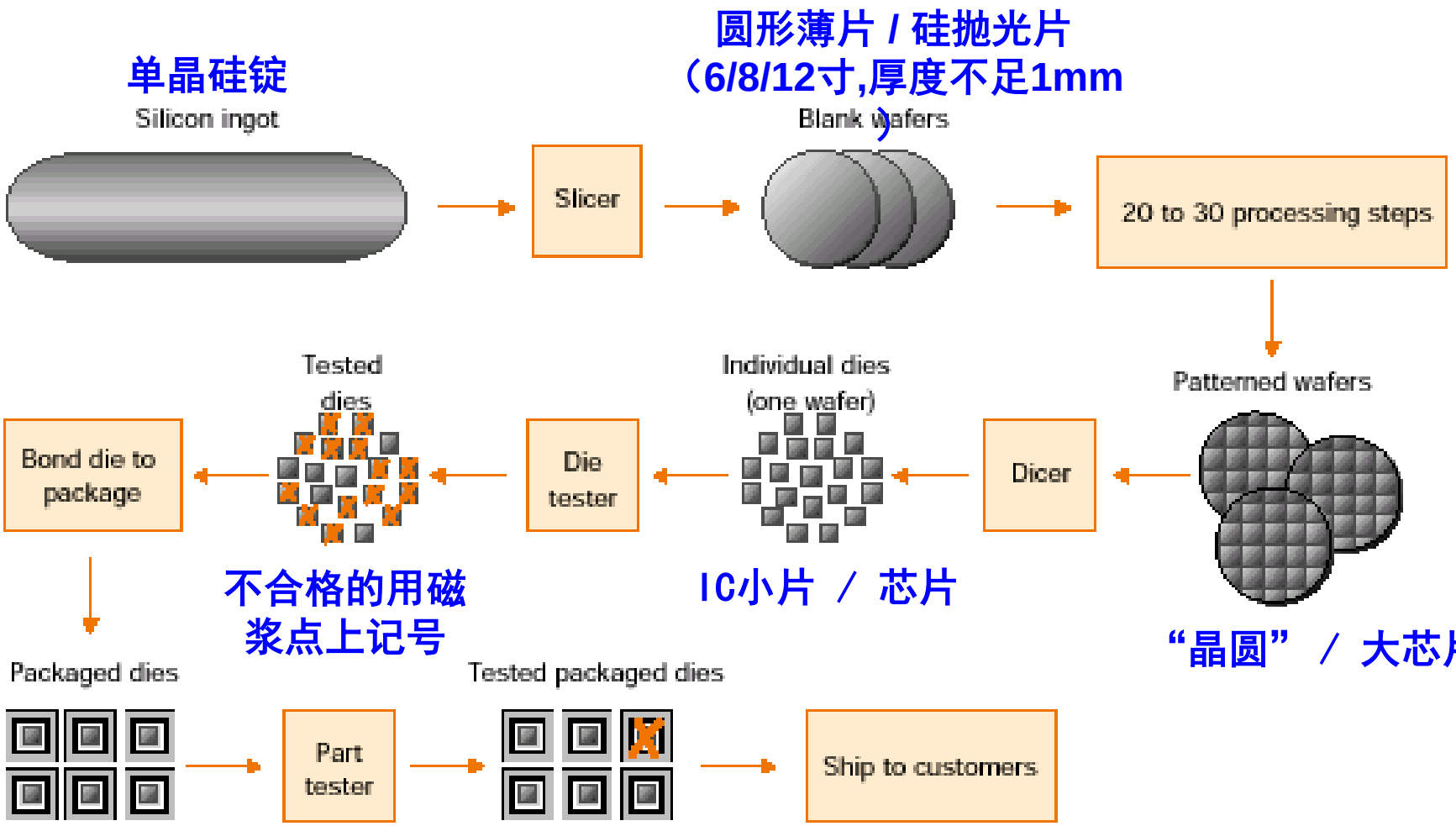
PC主板



解剖一台计算机（分而治之）



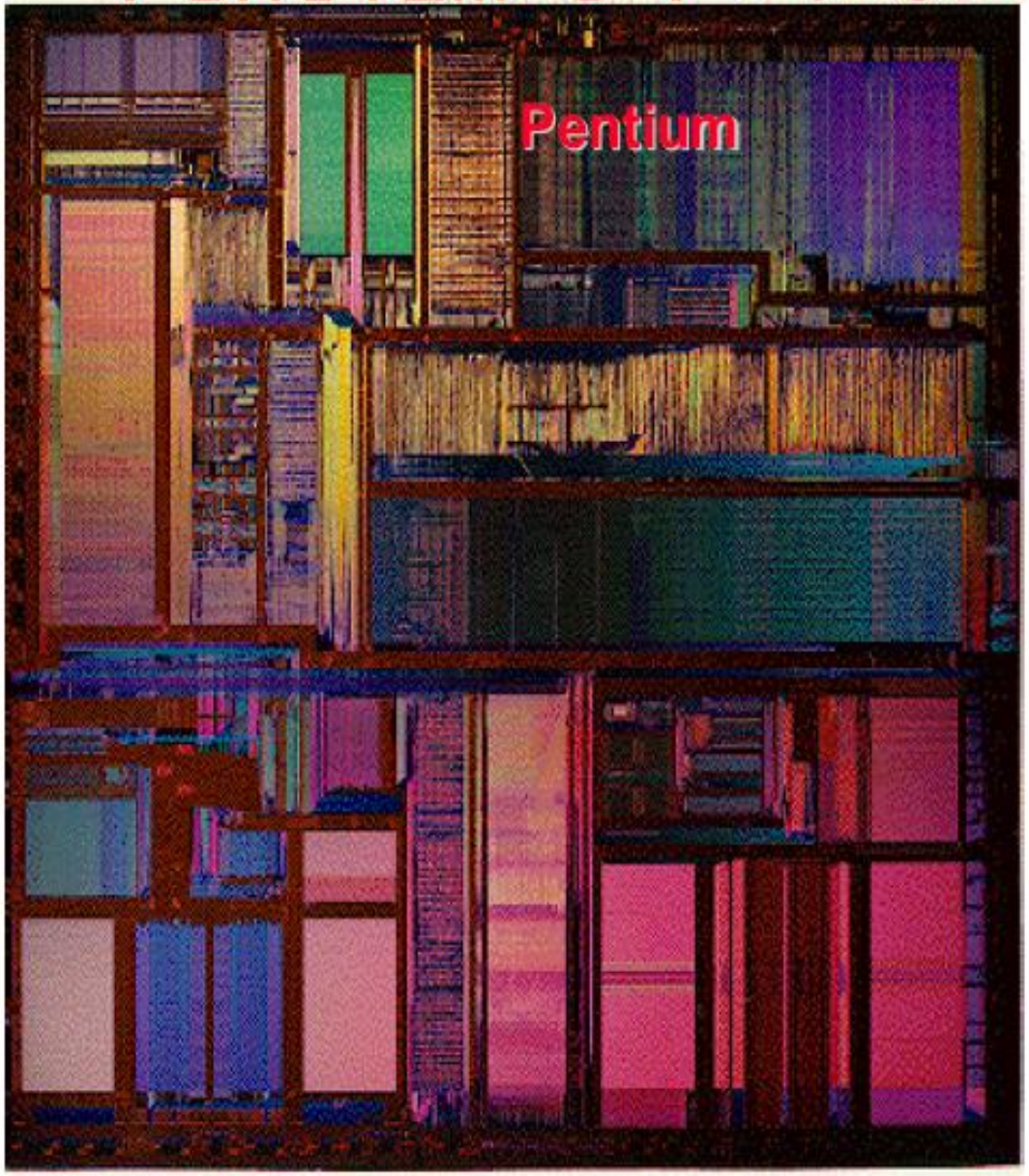
Integrated Circuits manufacturing process



封装：将芯片固定在塑胶或陶瓷基座上，把芯片上蚀刻出来的引线与基座底部伸出的引脚连接，盖上盖板并封焊成芯片

约需400多道工序！

Pentium芯片内的主要功能块



Die Area: 91 mm^2

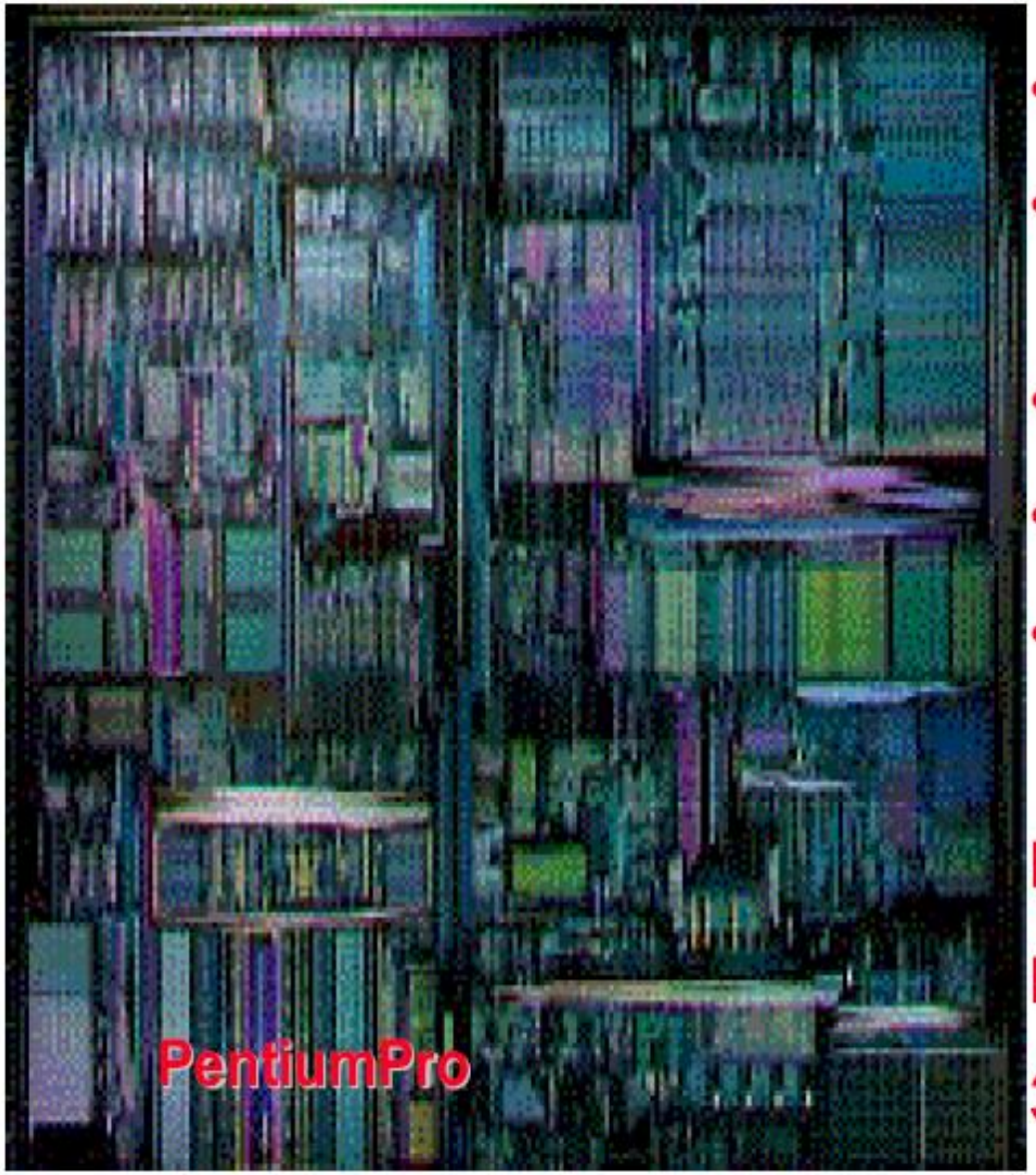
直径8 inch(200mm)的
Wafer最多可做 196个Die

≈ 3,300,000 Transistors

Cache: ≈1M Transistors

296 Pins

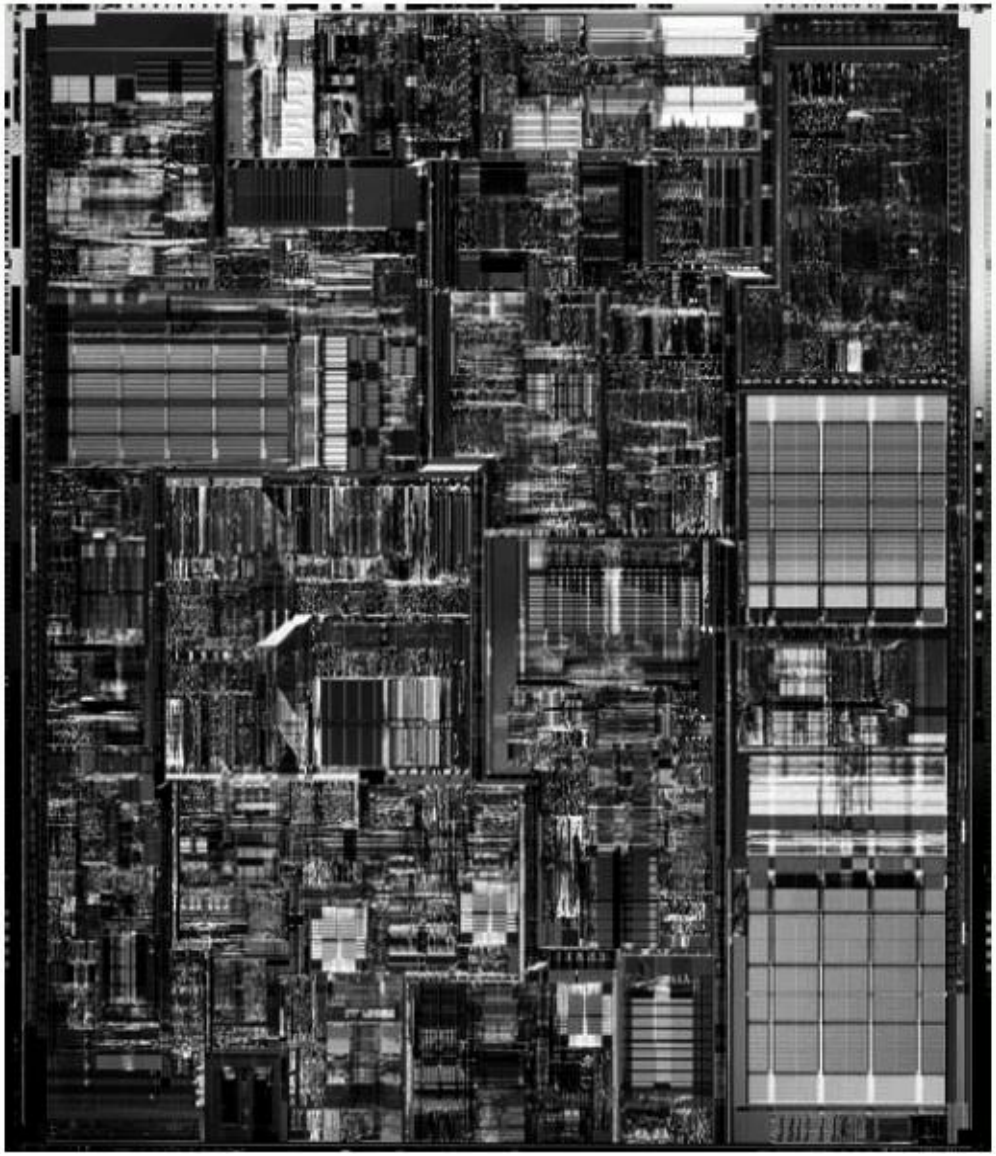
PentiumPro芯片内的主要功能块

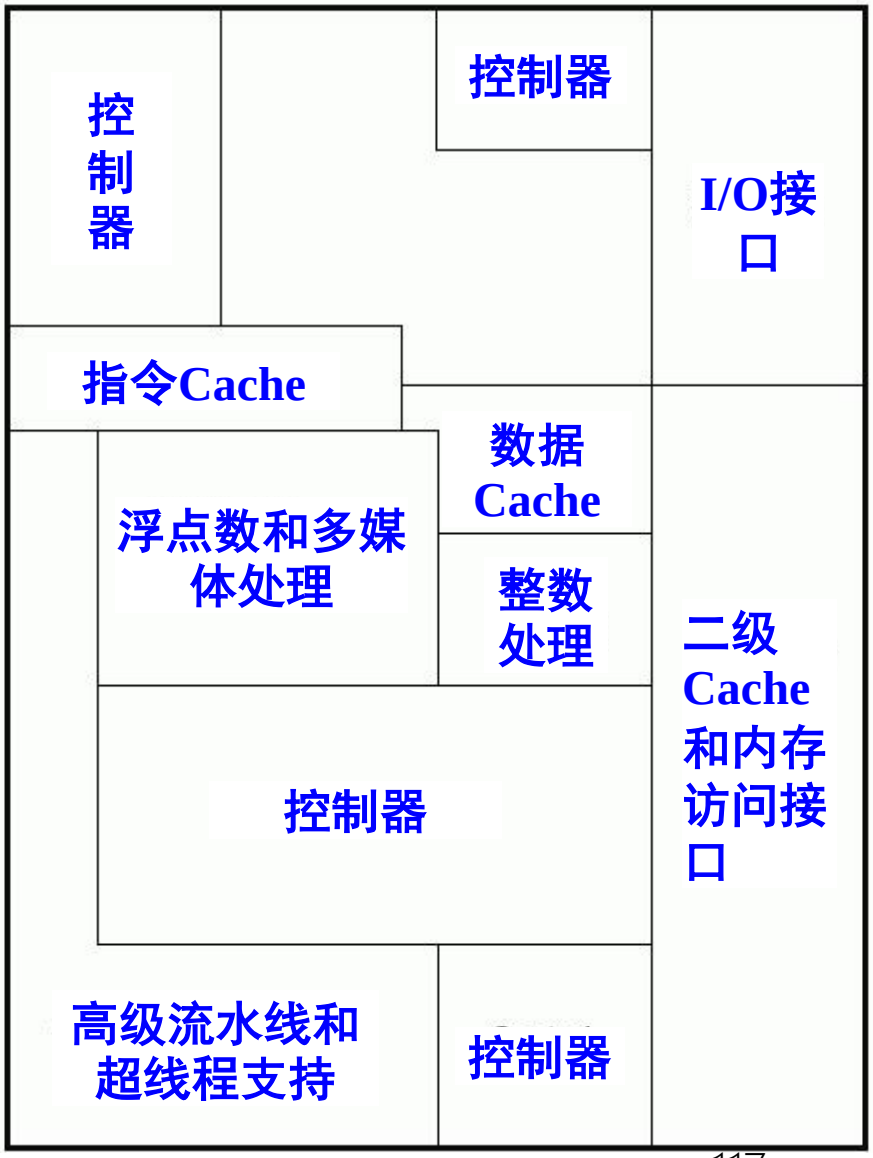


- Die Area: 306 mm^2
- 直径8 inch(200mm)的Wafer最多可做 78个Die
- ≈ 5,500,000 Transistors
- Cache: ≈1M Transistors
- External Cache: 31M Transistors

PentiumPro Package = PentiumPro+ExternalCache 387 Pins

Pentium4处理器内部布局





早期计算机系统的层次

- 最早的计算机用机器语言编程

  机器语言称为第一代程序设计语言（First generation programming language，1GL）

- 后来用汇编语言编程

  汇编语言称为第二代程序设计语言（Second generation programming language，2GL）

现代（传统）计算机系统的层次

NJUAI

- 现代计算机用高级语言编程

  第三代程序设计语言（3GL）为过程式语言，编码时需要描述实现过程，即“如何做”。

  第四代程序设计语言（4GL） 为非过程化语言，编码时只需说明“做什么”，不需要描述具体的算法实现细节。

可以看出：语言的发展是一个不断“抽象”的过程，因而，相应的计算机系统也不断有新的层次出现

语言处理系统包括：各种语言处理程序（如编译、汇编、链接）、运行时系统（如库函数、调试、优化等功能）

操作系统包括人机交互界面、提供服务功能的内核例程