

汇编语言程序设计

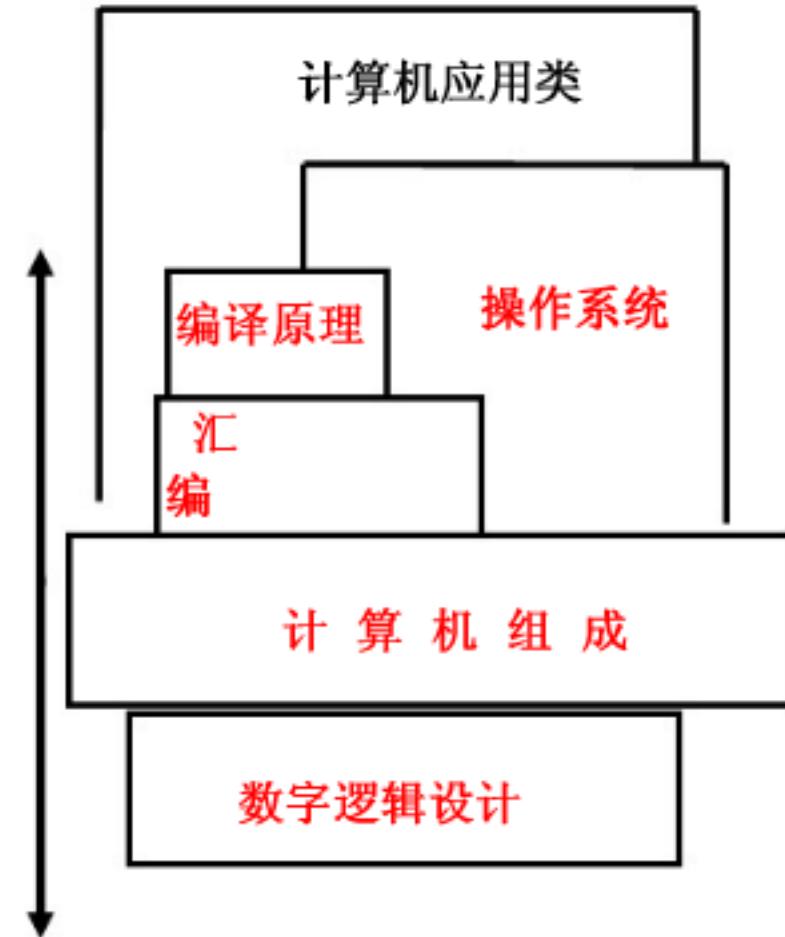
汇编语言与计算机系统结构

- ▶ 课程定位
- ▶ 主要授课内容
- ▶ 各类指令集简介

# 课程定位

与计算机组成原理、编译原理、  
操作系统、数字逻辑设计等组  
成计算机系统核心课程

- 汇编语言程序设计与计算机组成  
原理作为软硬件界面起到“承上  
启下”的作用



为后续课程打下指令集、汇编编程以及微体系结构入门的基础。

**机器语言**: 依赖于机器的低级语言，书写格式为二进制代码

优点: 执行速度快，效率高

缺点: 表达的意义不直观，编写、阅读、调试较困难

**汇编语言**: 是一种符号语言，与机器语言一一对应；它使用**助记符**表示相应的操作，并遵循一定的语法规则

优点: 面向机器编程，在“时间”和“空间”上效率

缺点: 涉及硬件细节，要求熟悉计算机系统的内部结构

**高级语言**: 面向人的语言，表达形式接近自然语言

优点: 便于阅读，易学易用，不涉及硬件，具有通用性

缺点: 目标代码冗长，占用内存多，从而执行时间长，效率不高，不能对某些硬件进行操作

- ▶ 编程语言是针对目标计算机系统结构的一种“抽象”
- ▶ 不同编程语言体现了对于系统结构的不同层面的“视角”

# 计算机系统结构

- ▶ **计算机系统结构**（中国计算机科学技术百科全书的定义）
  - 计算机系统的物理或者硬件结构、各部分组成的属性以及这些部分的相互联系。
  - 系统软件开发人员看到的计算机系统的功能行为和概念结构
  - 计算机系统的结构与实现（计算机组成）

计算机系统结构(狭义)  
Computer Architecture

程序员所看的计算机  
系统的属性

计算机组成  
Computer Organization

计算机系统的逻辑实现

计算机实现  
Computer Implementation

计算机系统的物理实现

- ▶ 计算机系统结构是研究计算机系统自身的学科。

- 其它定义：对计算机系统中各级之间界面的划分和定义，以及对各级界面上、下的功能进行分配。
  - 1964年，IBM/360系列机的总设计工程师G. M. Amdahl、G. A. Blaauw、F. P. Brooks等人提出。也称**体系结构**。
  - 是从**程序员**的角度所**看到的**系统的属性，是概念上的结构和功能上的行为
    - **程序员**：系统程序员（包括：汇编语言、机器语言、编译程序、操作系统）
    - **看到的**：编写出能在机器上正确运行的程序所必须了解到的
  - 它不同于数据流程和控制的组织，不同于逻辑设计以及物理实现方法。

一般认为这是一种狭义的定义。

# 范 围(广义)

- 指令系统 (Instruction set architecture, or ISA)
  - 机器语言, 还包括机器字长、内存地址模式、处理器寄存器等程序员可见的系统状态、数据格式等。
- 微体系结构 (Micro-architecture)
  - 如何实现指令集。
  - 处理器内部的实现, 包括流水线、处理部件、缓存等内容。
- 计算机系统
  - 计算机系统里的其它硬件, 包括总线、交换开关(Switch)、内存控制器、DMA控制器等。
- 其他内容
  - 虚拟化、计算机集群、NUMA。

- 指令系统（汇编语言可以看做是它的一种助记符）
  - 计算机处理器对外提供的主要接口与规格
  - 软硬件的分界
  - 系统程序员看到的计算机的主要属性

- 指令系统分类
  - CISC (复杂指令系统, Complex Set Instruction Computer)
    - 面向高级语言, 缩小机器指令系统与高级语言语义差距
    - 指令条数多, 寻址方式多变
    - 单条指令能够完成多个操作
    - 代表: X86
  - RISC (精简指令系统, Reduced instruction set computer)
    - 通常只支持常用的能在一个周期内完成的操作 (80: 20原则)
    - 简单而统一格式的指令格式与译码
    - 只有LOAD和STORE指令可以访问存储器, 简单的寻址方式
    - 较多的寄存器
    - 指令条数相对较少, 依赖于编译器产生高效的代码;
    - 处理器微体系结构相对简单, 运行频率高。
    - 代表: MIPS / ARM / PowerPC

## ▶ CISC与RISC走向融合

- X86处理器内部采用类似RISC的micro-op
  - 出于兼容性考虑，其指令集一直属于CISC
- 经典的RISC指令集也日益扩展、复杂化
  - PowerPC指令集(RISC)有超过230多条指令，较许多CISC指令集更为复杂

Load / store with (without) other operations?

- MIPS是一个经典的RISC指令集，兼具RISC设计的简洁优雅与不足
  - 代码密度较低
  - 应用于嵌入式领域，32bit指令有些“大材小用”
  - 因此注重扩展，包括提高代码密度（16位指令）以及多媒体、加密领域的指令扩展
- ARM指令集介于经典的RISC与CISC之间，相对复杂
  - 注重代码密度，降低功耗
  - 浮点较弱，逐步扩展强化

# 主要授课内容



## ▶ 基本知识

- 各类指令集初步
- 数制与整数表示
- 浮点数表示

## ▶ X86汇编

- 80x86计算机组织与保护模式
- X86指令系统与寻址方式
- C与X86汇编
- X86汇编语言程序格式与基本编程

## ▶ MIPS汇编

- MIPS计算机组织初步
- 指令系统介绍
- 汇编代码与异常处理

# 学习目标与要求：

- 了解汇编语言与计算机系统结构的**关系**及其起到的作用
- 了解以 X86系列微处理器为基础的PC机的**编程结构**，建立起“机器”与“程序”、“时间”与“空间”的概念
- 基本掌握 X86系列微处理器的**指令系统及寻址方式**，能够**编写程序**
- 掌握C语言基本代码段与汇编语言的**对应关系**
  
- 掌握MIPS指令集的**指令类型**，初步了解对应的**系统结构**
- 掌握撰写MIPS异常处理句柄的方法
- 初步了解MIPS指令集在操作系统中的作用\*

## 参考书：

1. 《深入理解计算机系统》( Computer Systems: A Programmer's Perspective ) , 第二版 , 2010

- 第2、3章。

- 课程ppt的部分素材也来源于该书作者的课程网站。

2. SEE MIPS RUN ( MIPS体系结构透视 ). 第二版 , 2008.

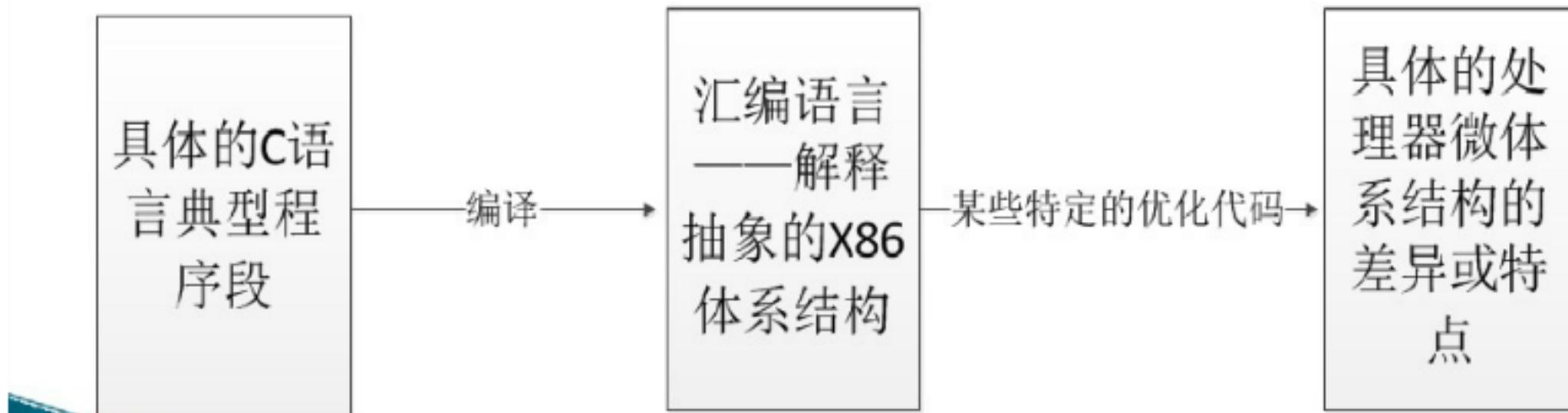
3. IBM-PC汇编语言程序设计 ( 第2版 )  
婵 编著 清华大学出版社

沈美明 温冬

## ▶ 授课角度

- 突出了在课程组中的“承上启下”作用，在内容编排上突出了与相关课程的衔接
  - 强化与高级语言的联系，从典型的C语言代码段入手，通过编译成汇编代码来详细解释程序员角度的微体系结构（包括X86与MIPS结构）运行模型。
  - 汇编语言是高级语言在机器层面的表示，掌握这两种语言的对应可以将程序的执行与计算机的工作过程紧密联系起来。
  - 通过对不同汇编代码的解释来给出微体系结构方面的差异，为后续课程，如编译原理、计算机组成原理等提供一些先导知识。
- 开展多角度内容组织，系统角度与应用角度并存
  - 从C语言角度、处理器结构角度分别“自上而下”、“自下而上”的以纵向角度讲解汇编语言的语义及其某些新的汇编指令出现的原因。
  - 突出汇编优化程序的特点以及“反汇编”应用。

- 从典型的C语言代码段入手，通过编译成汇编代码来详细解释程序员角度的X86结构运行模型
  - 可以给编译给出一些先导知识
- 再通过对不同汇编代码的解释来稍进一步的给出微体系结构方面的差异
  - 为后续的计原等课程给出一些先导知识



- 举例：条件移动指令

```
int absdiff(
    int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```



\_absdiff:

|              |                   |
|--------------|-------------------|
| pushl        | %ebp              |
| movl         | %esp, %ebp        |
| movl         | 8(%ebp), %ecx     |
| pushl        | %ebx              |
| movl         | 12(%ebp), %edx    |
| movl         | %ecx, %eax        |
| movl         | %edx, %ebx        |
| subl         | %edx, %eax        |
| subl         | %ecx, %ebx        |
| cmpl         | %edx, %ecx        |
| <b>cmove</b> | <b>%ebx, %eax</b> |
| popl         | %ebx              |
| popl         | %ebp              |
| ret          |                   |

gcc ... -O2 –march=i686

- 对照

\_absdiff:

|       |                |
|-------|----------------|
| pushl | %ebp           |
| movl  | %esp, %ebp     |
| movl  | 8(%ebp), %edx  |
| movl  | 12(%ebp), %eax |
| cmpl  | %eax, %edx     |
| jle   | L27            |
| popl  | %ebp           |
| subl  | %eax, %edx     |
| movl  | %edx, %eax     |
| ret   |                |

.p2align 4,,7

L27:

|      |            |
|------|------------|
| popl | %ebp       |
| subl | %edx, %eax |
| ret  |            |

- 从处理器角度来解释

- 首先较新的X86处理器增加了条件执行（移动）指令

- 为何增加

- 从处理器的长流水线、多发射角度来解释条件跳转指令对性能可能带来的负面影响

- 尽量消除条件跳转指令

# △微体系结构背景

现代的通用处理器 支持深度流水线以及多发射结构，如  
Pentium 4 : >= 20 stages, up to 126 instructions on-fly

## Superscalar execution

| Clock cycle → | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | 10 | 11 |
|---------------|----|----|----|----|----|----|---|---|---|----|----|
| Instr. i      | FI | DI | CO | FO | EI | WO |   |   |   |    |    |
| Instr. i+1    | FI | DI | CO | FO | EI | WO |   |   |   |    |    |
| Instr. i+2    | FI | DI | CO | FO | EI | WO |   |   |   |    |    |
| Instr. i+3    | FI | DI | CO | FO | EI | WO |   |   |   |    |    |
| Instr. i+4    | FI | DI | CO | FO | EI | WO |   |   |   |    |    |
| Instr. i+5    | FI | DI | CO | FO | EI | WO |   |   |   |    |    |

条件指令往往会引起一定的性能损失，因此需要尽量消除。

# 各类指令集简介



# X86指令集



## ► X86指令集的基本特色

- 向下兼容
- 变长指令
  - 1-15 字节，多数为2-3字节长度
- 多种寻址方式（可访问不对齐内存地址）

$$\text{Offset} = \left( \begin{array}{l} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esp} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{array} \right) + \left( \begin{array}{l} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esp} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{array} \right) * \left( \begin{array}{l} 1 \\ 2 \\ 4 \\ 8 \end{array} \right) + \left( \begin{array}{l} \text{None} \\ \text{8-bit} \\ \text{16-bit} \\ \text{32-bit} \end{array} \right)$$

Base                  Index                  scale                  displacement

- 指令集的通用寄存器个数有限
  - X86-32系统下拥有8个通用寄存器（x86-64扩展到16个）
- 至多只有一个操作数在内存中，另一个操作数为立即数或者寄存器

# x86-32/64 General Purpose Registers

|      |      |
|------|------|
| %rax | %eax |
| %rdx | %edx |
| %rcx | %ecx |
| %rbx | %ebx |
| %rsi | %esi |
| %rdi | %edi |
| %rsp | %esp |
| %rbp | %ebp |

|      |       |
|------|-------|
| %r8  | %r8d  |
| %r9  | %r9d  |
| %r10 | %r10d |
| %r11 | %r11d |
| %r12 | %r12d |
| %r13 | %r13d |
| %r14 | %r14d |
| %r15 | %r15d |

## ▶ X86指令类型

- 数据传输指令

- 在存贮器和寄存器、寄存器和输入输出端口之间传送数据

- 通用数据传送指令

- MOV、PUSH、POP、XCHG

- 累加器专用传送指令

- IN、OUT、XLAT

- 地址传送指令

- LEA、LDS、LES

- 标志寄存器传送指令

- LAHF、SAHF、PUSHF、POPF

- 类型转换指令

- CBW、CWD

- 算术指令
  - 加法指令  
ADD、ADC、INC
  - 减法指令  
SUB、SBB、DEC、NEG、CMP
  - 乘法指令  
MUL、IMUL
  - 除法指令  
DIV、IDIV
  - 十进制调整指令  
DAA、DAS、AAA、AAS、AAM、AAD

## • 逻辑指令

- 逻辑运算指令

AND、OR、NOT、XOR、TEST

- 移位指令

SHL、SHR、SAL、SAR、ROL、ROR、RCL、RCR

## 控制转移指令：

- 无条件转移指令

**JMP**

- 条件转移指令

**JZ / JNZ、 JE / JNE、 JS / JNS、 JO / JNO、  
JP / JNP、 JB / JNB、 JL / JNL、 JBE / JNBE、  
JLE / JNLE、 JCXZ**

- 循环指令

**LOOP、LOOPZ / LOOPE、LOOPNZ / LOOPNE**

- 子程序调用和返回指令

**CALL、RET**

- 中断与中断返回指令

**INT、INTO、IRET**

## 处理机控制与杂项操作指令：

- 标志处理指令
  - CLC、 STC、 CMC、
  - CLD、 STD、
  - CLI、 STI
- 其他处理机控制与杂项操作指令
  - NOP、 HLT、 WAIT、 ESC、 LOCK

## ► X86浮点指令与SIMD

- 专用的浮点寄存器（最初为80-bit internal registers，后宽度逐步扩展）
  - IEEE Standard 754 (Established in 1985 as uniform standard for floating point arithmetic)
    - *Extended precision: 15 exp bits, 63 frac bits*



- 早先的x86中浮点寄存器是一个基于栈的结构（而RISC处理器中的浮点寄存器一般可使用寄存器号直接寻址，并且寄存器个数较多）
  - ST0 ~ ST7

## ► SIMD (Single Instruction Multi-DATA)

- 单指令、多数据指令，使得多个运算可以在同一条指令内并发进行（向量运算）
  - 通常复用浮点寄存器，如128-bit的寄存器可以同时存放2或4个浮点数 (64 or 32 bits wide respectively)；或者是2, 4, 8 或16个整数，同时进行相同运算
  - 充分利用应用的数据并发性
- Intel Sandy Bridge架构拥有了256-bit 的SIMD 指令 (including 256-bit memory load and store, AVX).
- SIMD位宽的加大的一个前提条件是访问内存指令的数据宽度的增大
  - 但是SIMD的应用效率取决于两个方面，一是处理器的IO带宽；二是应用自身的特性能否提供充分的数据并发性

# SIMD一再拓宽...

## ▶ 较早的MMX (1997, Intel)

- 主要应用于多媒体处理
- MMX “增加”了8个寄存器 (MM0~MM7)
  - 实际上，其复用了现有的浮点寄存器 (ST0~ST7)，但是寻址方式不同
    - 寄存器号直接寻址
- 任一个MMX寄存器的宽度是64bit，使用了*packed data types* (two 32-bit integers (**doubleword**), four 16-bit integers (**word**) or eight 8-bit integers)

- MMX的缺点
  - 只支持整数运算，浮点数运算仍然要使用传统浮点指令。
  - 与浮点寄存器相互重叠，这限制了MMX指令在需要大量浮点运算的程序，如三维几何变换、裁剪和投影中的应用
  - 栈式暂存器结构，使得硬件上将其流水线化和软件上合理调度指令都很困难，这成为提高x86架构浮点性能的一个瓶颈。
- ▶ 3DNow! (AMD)
  - K6-2处理器是第一个能执行浮点SIMD指令的x86处理器
  - SIMD多媒体指令集，支持单精度浮点数的向量运算

- ▶ SSE (Streaming SIMD Extensions)
  - SSE 彻底抛弃了传统的栈式浮点处理器结构，
  - 8个独立的128位寄存器 (XMM0 ~ XMM7)，64位结构下增至16个

SSE-1——同时处理4个32位单精度浮点数

SSE-2——增加了对于 $2 \times 64$ 位浮点或者整数、 $4 \times 32$ 位整数、 $8 \times 16$ 位整数、 $16 \times 8$ 位整数的支持

- Intel SandyBridge架构引入了Advanced Vector Extensions指令集扩展
  - SIMD 寄存器宽度增至256位，16个寄存器(YMM0-YMM15)
  - 向下兼容SSE
  - 每个周期可以进行两个256-bit 的AVX操作
  - 引入了三操作数的指令，
  - Sandy Bridge架构支持256位的内存访问接口

2015年计划推出最新的AVX-512（512位）

## ▶ 为何X86指令集长久不衰？

- 商业上的成功是其主要原因（生态环境）
- 技术上注重向下兼容
  - 一个反例是Itanium (IA-64, Explicitly Parallel Instruction Computing )，技术创新但是无法兼容，已基本“死亡”

## ▶ X86指令集的缺点？

- 向下兼容导致指令集越来越大、越复杂
- 类RISC内核，采用micro-op模式进行翻译，使得功耗相对增大，这导致其在注重低功耗的嵌入式领域不易占优势
- 对很多领域而言，资源利用率低
  - 在高性能计算领域，300余条X86指令中，大致只有80余条是被科学计算所需要的（美国劳伦斯伯克利国家实验室的研究，2009）

# MIPS指令集



- ▶ 经典的RISC指令集（主要应用于嵌入式领域）
  - MIPS I、MIPS II、MIPS III、MIPS IV到MIPS V，嵌入式指令体系MIPS16、MIPS32到MIPS64的发展已经十分成熟
  - 为充分利用处理器的流水线结构，其设计思想是使得各个指令的流水线分段较为均匀
    - 分段一致，每段的操作时延差不多，从而提高主频
    - 尽量使得能够每一周期完成一条指令，控制相对简单
  - 尽量利用软件办法避免流水线中的数据相关/控制相关问题
    - 一个实例：Delay Slot

- 以寄存器为中心（32个），只有Load/Store指令访问内存，所有的计算类型的指令均从寄存器堆中读取数据并把结果写入寄存器堆中。
  - MIPS32还定义了32个浮点寄存器
- MIPS32指令集的指令格式非常规整，所有的指令长度一定，而且指令操作码在固定的位置上。
- MIPS指令的寻址方式非常简单，每条指令的操作也较简单

## ► MIPS32™的指令格式只有3种

- R (register) 类型的指令从寄存器堆中读取两个源操作数，计算结果写回寄存器堆。
- I (immediate) 类型的指令使用一个16位的立即数作为源操作数。
- J (jump) 类型的指令使用一个26位立即数作为跳转的目标地址 (target address)。

|     | 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11        | 10 | 06   | 05 | 00 |
|-----|----|----|----|----|----|----|----|-----------|----|------|----|----|
| R类型 |    | op | rs | rt |    | rd |    | sa        |    | func |    |    |
|     |    | 6位 | 5位 | 5位 |    | 5位 |    | 5位        |    | 6位   |    |    |
| I类型 | 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11        | 10 | 06   | 05 | 00 |
|     |    | op | rs | rt |    |    |    | immediate |    |      |    |    |
|     |    | 6位 | 5位 | 5位 |    |    |    | 16位       |    |      |    |    |
| J类型 | 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11        | 10 | 06   | 05 | 00 |
|     |    | op |    |    |    |    |    | target    |    |      |    |    |
|     |    | 6位 |    |    |    |    |    | 26位       |    |      |    |    |

- ▶ Load和Store指令都为立即数（I-type）类型，用来在存储器和通用寄存器之间的储存和装载数据。MIPS指令集只有该类指令访问内存，而其他指令都在寄存器之间进行，所以指令的执行速度较高。
  - 该类指令只有基址寄存器的值加上扩展的16位有符号立即数一种寻址模式，数据的存取方式可以是字节（byte）、字（word）和双字（Double word）。

- MIPS扩展指令集
  - MIPS-3D, 浮点SIMD 用于三维几何处理 (MIPS64架构下)
  - Vertex transformation (matrix multiplication)
  - Clip-check (compare and branch)
  - Transform to screen coordinates (perspective division using reciprocal)
  - Lighting : infinite and local (normalization using reciprocal square root)
- 采用MIPS64浮点运算单元和双单精度数据类型。
  - PS (paired-single, 双单精度)操作可对64位寄存器中的两个32位浮点值进行运算, 从而提供2路SIMD (单指令多数据)能力

- MIPS 扩展指令集

- MDMX (MaDMAx, MIPS digital media extensions)
  - 定义了一个“新”的64位寄存器堆（32个）以及一个192位的乘法累加器
  - 重命名了现有的浮点寄存器
  - 新的数据类型： octo byte (8x8 Bit) and quad half (4x16 Bit)
  - 这些数据类型常见于音视频、图像处理应用

| Modified Instructions   |  |
|---|--|
| ADD, SUB, MUL, MIN, MAX, MSGN<br>AND, XOR, OR, NOR, SLL, SRL, SRA | Saturating arithmetic<br>Logicals and shifts |
| ALNI, ALNV  | Align vectors                                |
| C.EQ, C.LT, C.LE  | Compare bytes                                |
| New Instructions  |  |
| SHFL.op   | Shuffle bytes                                |
| PICKF, PICKT  | Combine vectors                              |
| ADDL, SUBL, MULL, MULSL   | Store result in ACC                          |
| ADDA, SUBA, MULA, MULS  | Operate on ACC                               |
| RZU, RNAU, RNEU, RZS, RNAS, RNES<br>RAC, WAC                      | Round ACC<br>Read/write ACC                  |

- MIPS扩展指令集
  - MIPS16e
  - 16位指令，指令集被压缩，代码存储容量要求减小，从而降低系统成本
  - 相比MIPS32，利用MIPS16e编译的应用程序平均减小30%
    - 32个通用寄存器中有8个可用于MIPS16e模式
    - 与MIPS32一起使用时，支持8位、16位和32位数据类型；与MIPS64一起使用时，支持8位、16位、32位和64位数据类型
  - MIPS16e 和 MIPS32/64之间的模式切换支持：通过一条特殊的跳转指令来实现模式切换的软件控制

## ▶ 小结

- MIPS是一个经典的RISC指令集，兼具RISC设计的简洁优雅与不足
  - 代码密度较低
  - 应用于嵌入式领域，32bit指令有些“大材小用”
- 因此注重扩展，包括提高代码密度（16位指令）以及多媒体、加密领域的指令扩展

# ARM指令集



## ▶ “有些不同”的RISC指令集（32位）

- ARM 指令提供简单的操作，使一个周期就可以执行一条指令。编译器或者程序员通过几条简单指令的组合来实现一个复杂的操作
- ARM 指令集大多数指令采用相同的字节长度，并且在字边界上对齐，字段位置固定，特别是操作码的位置。
- ARM 处理器使用Load/Store 的存储模式，其中只有 Load 和Store 指令才能从内存中读取数据到寄存器，所有其他指令只对寄存器中的操作数进行计算。
  - 16 个32位寄存器

## ▶ 与MIPS相比，寻址相对复杂

- 立即数寻址

每个立即数都是采用一个8位的常数循环右移偶数位间接得到。

- 寄存器寻址

ADD R3, R2, R1, LSR #2

寄存器R1的内容右移了两位，但是注意本指令执行完毕以后R1的内容并不改变。

- 前变址、自动变址和后变址

1、前变址：LDR R0, [R1, #4]

R1寄存器的内容先加4，然后执行内存操作，但操作完毕以后，R1的内容不变。

2、自动变址

R1寄存器的内容先加4，然后执行内存操作，R1变化

3、后变址：LDR R0, [R1], #4

先进行内存操作，然后 $R1 + 4 \rightarrow R1$ （即操作完毕后，变化）

- 栈寻址

- 多寄存器寻址

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。  
◦ 这种寻址方式可以用一条指令完成传送最多16个通用寄存器的值。

LDMIA R0!, {R1, R2, R3, R4}

; R1←[R0]  
; R2←[R0+4]  
; R3←[R0+8]  
; R4←[R0+12]

该指令的后缀IA表示在每次执行完加载/存储操作后，R0按字长度增加，因此，指令可将连续存储单元的值传送到R1～R4。

- 多数指令支持条件执行
  - A 4-bit condition code selector
  - 多数处理器只有跳转指令有条件码判断
  - 这一设计的优势在于对于简单的if-else语句，无需产生跳转指令；
  - 劣势在于减少了“立即数”域的位数

```

while(i != j) {
    if (i > j)
        i -= j;
    else
        j -= i;
}
  
```

→

loop CMP Ri, Rj ; set condition "NE" if (i != j),  
; "GT" if (i > j),  
; or "LT" if (i < j)  
SUBGT Ri, Ri, Rj ; if "GT" (greater than), i = i-j;  
SUBLT Rj, Rj, Ri ; if "LT" (less than), j = j-i;  
BNE loop ; if "NE" (not equal), then loop

- 其他特性（一般RISC指令集所不具备的）
  - PC-relative addressing
  - 融合数据处理功能（数据移动、算术计算）与移位（shifts 与 rotates）功能

- ARM的Thumb指令集

- Thumb指令可以看做是ARM指令压缩形式的子集，为提高代码密度而引入的
- 必须与ARM指令集混用
- 主要区别
  - 除了跳转指令 有条件执行功能外,其它指令均为无条件执行
  - 没有乘加指令及 64 位乘法指令等，且指令的第二操作数受到限制

The shorter opcodes give improved code density overall, even though some operations require extra instructions. In situations where the memory port or bus width is constrained to less than 32 bits, the shorter Thumb opcodes allow increased performance compared with 32-bit ARM code, as less program code may need to be loaded into the processor over the constrained memory bandwidth.

- 浮点运算扩展
  - ARM的浮点运算能力不是强项
    - 某些ARM内核只支持软件浮点指令模拟
  - VFP
    - single-precision and double-precision floating-point computation
    - 但是对于SIMD的支持不好
  - NEON
    - 64/128位的 SIMD浮点指令集（相当于Intel的SSE）
    - 采用独立的寄存器堆与硬件流水线
    - 支持8-, 16-, 32- and 位整数与单精度(32-bit) 浮点数据与计算

## ▶ 小结

- ARM指令集介于经典的RISC与CISC之间，相对复杂
- 注重代码密度，降低功耗
- 浮点较弱，逐步扩展强化

## ▶ 小结

- CISC与RISC指令集互为借鉴，走向融合
- 兼容性考虑是指令集发展的关键性因素
- 为提高数据并行度，SIMD扩展是指令集发展的一个共性
  - 但是取决于处理器访问存储的宽度以及应用的特性
  - 充分利用SIMD是一个非常困难的事！

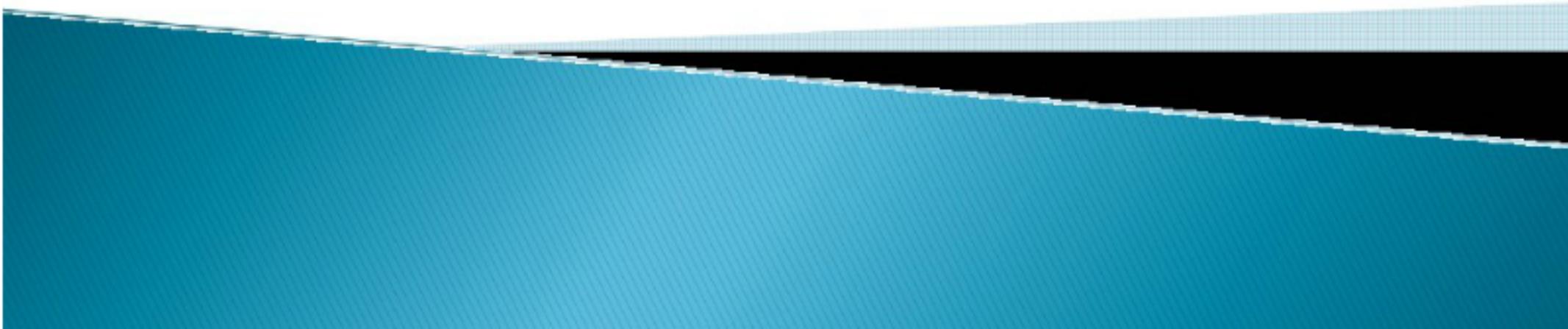
# 补充\*



- ▶ Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures
  - In the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA 2013)
  - Question
    - The question of whether ISA plays an intrinsic role in performance or energy efficiency is becoming important. We seek to answer this question through a detailed measurement based study on real hardware running real applications.
  - Answer
    - We find that ARM and x86 processors are simply engineering design points optimized for different levels of performance, and there is nothing fundamentally more energy efficient in one ISA class or the other. The ISA being RISC or CISC seems irrelevant.

汇编语言程序设计

# 整 数



- ▶ 数制
- ▶ 数制之间的转换
- ▶ 逻辑运算
- ▶ 数的机器表示(初步)
- ▶ 整数表示

## 预备知识：

**1K =  $2^{10}$  = 1024 (Kilo)**

**1M = 1024K =  $2^{20}$  (Mega)**

**1G = 1024M =  $2^{30}$  (Giga)**

**1T = 1024G =  $2^{40}$  (Tera)**

**1P = 1024T =  $2^{50}$  (Peta)**

1个二进制位：**bit** (比特)

8个二进制位：**Byte** (字节) **1Byte = 8bit**

2个字节：**Word** (字)

**1Word = 2Byte = 16bit**

# 1. 数制

| 数 制              | 基 数 | 数 码   |
|------------------|-----|---|
| 二进制 Binary       | 2   | 0, 1  |
| 八进制 Octal        | 8   | 0, 1, 2, 3, 4, 5, 6, 7                            |
| 十进制 Decimal      | 10  | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9                      |
| 十六进制 Hexadecimal | 16  | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,<br>A, B, C, D, E, F |

## 2. 数制之间的转换

- 二进制  $\xleftrightarrow{\hspace{1cm}}$  十六进制
  - 十进制  $\xleftrightarrow{\hspace{1cm}}$  二进制
  - 十进制  $\xleftrightarrow{\hspace{1cm}}$  十六进制
- $\longrightarrow$  降幂法 除法

十六进制数：逢十六进一 借一当十六

$$\begin{array}{r} 0\ 5\ C\ 3\ H \\ + 3_1 D\ 2\ 5\ H \\ \hline 4\ 2\ E\ 8\ H \end{array}$$
$$\begin{array}{r} 3\ D^{-1}\ 2\ 5\ H \\ - 0\ 5\ C\ 3\ H \\ \hline 3\ 7\ 6\ 2\ H \end{array}$$

### 3. 逻辑运算（按位操作）

“与”运算 (AND)

| A | B | $A \wedge B$ (&) |
|---|---|------------------|
| 0 | 0 | 0                |
| 0 | 1 | 0                |
| 1 | 0 | 0                |
| 1 | 1 | 1                |

“或”运算 (OR)

| A | B | $A \vee B$ ( ) |
|---|---|----------------|
| 0 | 0 | 0              |
| 0 | 1 | 1              |
| 1 | 0 | 1              |
| 1 | 1 | 1              |

“非”运算 (NOT)

| A | $\neg A$ (~) |
|---|--------------|
| 0 | 1            |
| 1 | 0            |

“异或”运算 (XOR)

| A | B | $A \Delta B$ (^) |
|---|---|------------------|
| 0 | 0 | 0                |
| 0 | 1 | 1                |
| 1 | 0 | 1                |
| 1 | 1 | 0                |

例：X = 00FFH Y = 5555H, Z = X $\forall$ Y = ?

$$\begin{array}{r} X = 0000 \ 0000 \ 1111 \ 1111 \text{ B} \\ \forall \ Y = 0101 \ 0101 \ 0101 \ 0101 \text{ B} \\ \hline Z = 0101 \ 0101 \ 1010 \ 1010 \text{ B} \end{array}$$

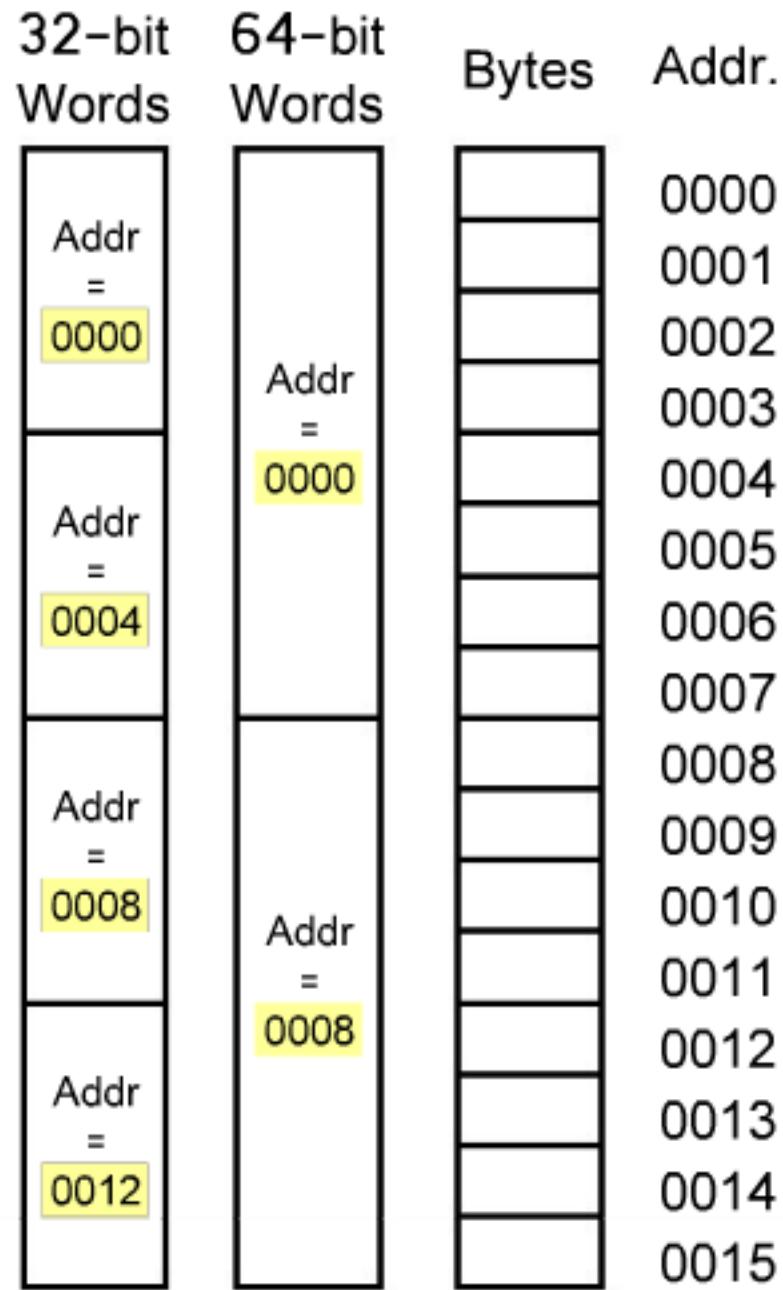
$\therefore Z = 55AAH$

## 4. 数的机器表示

- 机器字( machine word)长
  - 一般指计算机进行一次整数运算所能处理的二进制数据的位数
    - 通常也包括数据地址长度
  - 32位字长
    - 地址的表示空间是4GB
    - 对很多内存需求量大的应用而言，非常有限
  - 64位字长
    - 地址的表示空间约是 $1.8 \times 10^{19}$  bytes
    - 目前的x86-64 机型实际支持 48位宽的地址: 256 TB

## 机器字在内存中的组织

- 地址按照字节(byte)来定位
  - 机器字中第一个字节的地址
  - 相邻机器字的地址相差4(32-bit)或者8(64-bit)



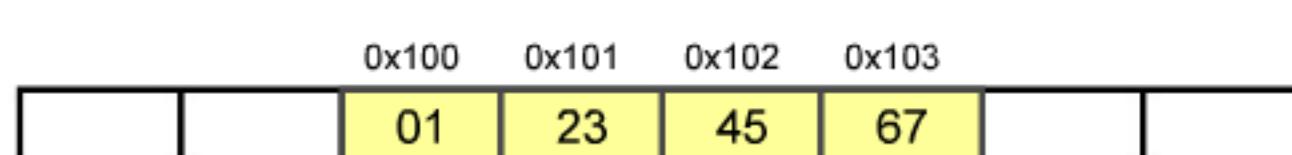
# 字节序 (Byte Ordering)

► 一个机器字内的各个字节如何排列？

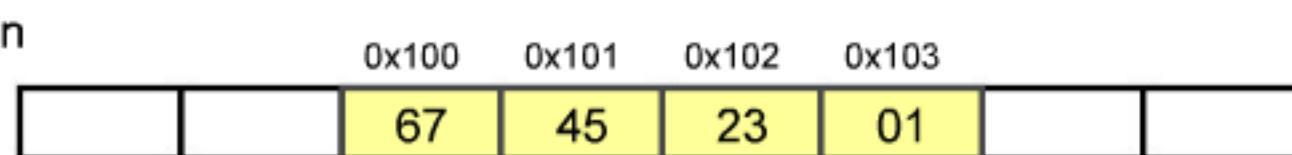
- Big Endian: Sun, PowerPC, Internet
  - 低位字节(Least significant byte, LSB) 占据高地址
- Little Endian: x86
  - 与LSB相反

数值是0x01234567，地址是0x100

Big Endian

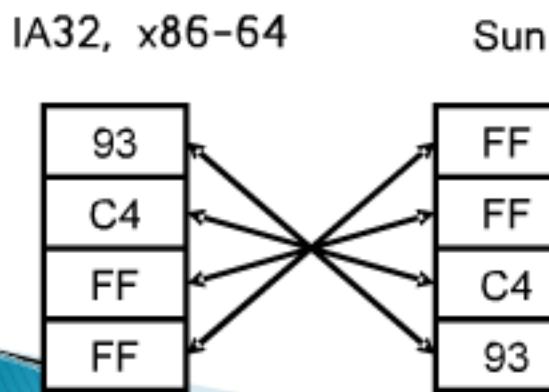
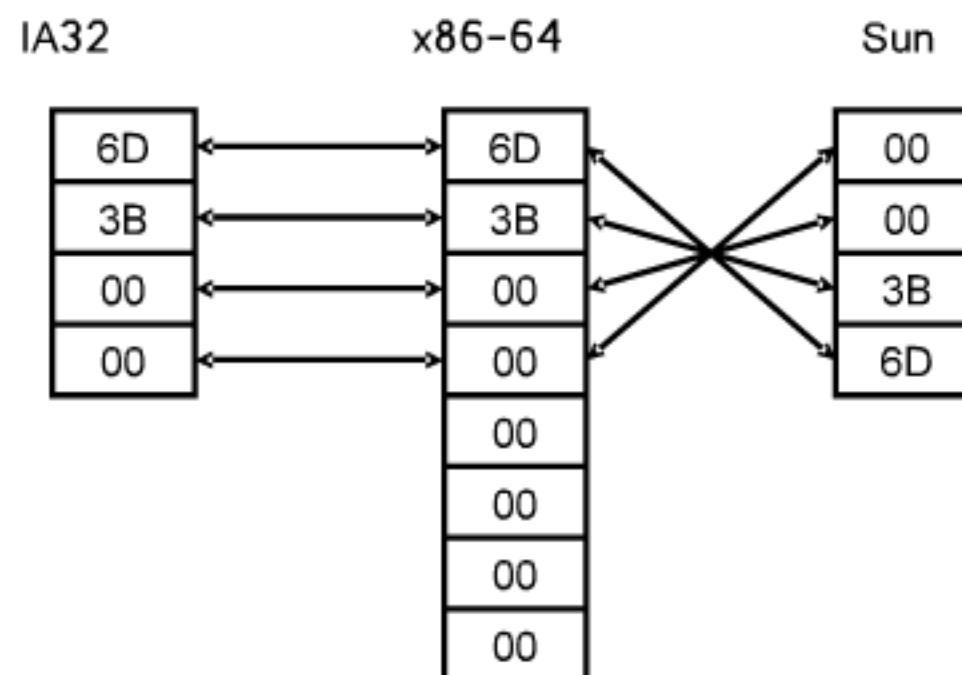
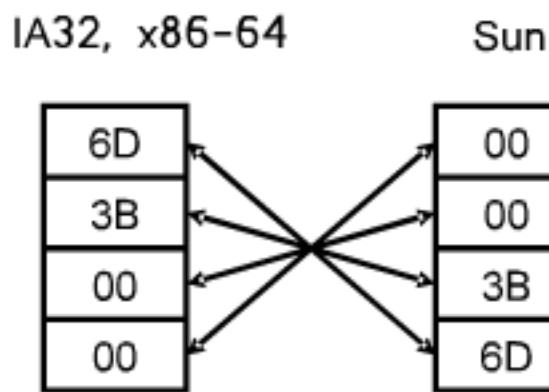


Little Endian



- ▶ int A = 15213;
- ▶ int B = -15213;
- ▶ long int C = 15213;

|          |                     |
|----------|---------------------|
| Decimal: | 15213               |
| Binary:  | 0011 1011 0110 1101 |
| Hex:     | 3 B 6 D             |



补码表示

## 5. 整数表示

### ▶ C语言中基本数据类型的大小 (in Bytes)

| C Data Type            | Typical 32-bit | x86-32 | x86-64 |
|------------------------|----------------|--------|--------|
| • char                 | 1              | 1      | 1      |
| • short                | 2              | 2      | 2      |
| • int                  | 4              | 4      | 4      |
| • long                 | 4              | 4      | 8      |
| • long long            | 8              | 8      | 8      |
| • float                | 4              | 4      | 4      |
| • double               | 8              | 8      | 8      |
| • long double          | 8              | 10/12  | 10/16  |
| • char *               | 4              | 4      | 8      |
| • Or any other pointer |                |        |        |

# Integer C Puzzles

- 判断以下的推断或者等式是否成立(不成立则给出范例)

- x, y 为32位带符号整数;

初始化

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- $x < 0 \Rightarrow ((x^2) < 0)$
- $ux \geq 0$
- $x \& 7 == 7 \Rightarrow (x << 30) < 0$
- $ux > -1$
- $x > y \Rightarrow -x < -y$
- $x * x \geq 0$
- $x > 0 \&& y > 0 \Rightarrow x + y > 0$
- $x \geq 0 \Rightarrow -x \leq 0$
- $x \leq 0 \Rightarrow -x \geq 0$
- $(x|-x) >> 31 == -1$
- $ux >> 3 == ux/8$
- $x >> 3 == x/8$
- $x \& (x-1) != 0$

# 计算机中整数的二进制编码方式 (w表示字长)

无符号数

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

带符号数 (补码, Two's Complement)  
)

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

```
short int x = 15213;  
short int y = -15213;
```

符号位

|   | Decimal | Hex   | Binary            |
|---|---------|-------|-------------------|
| x | 15213   | 3B 6D | 00111011 01101101 |
| y | -15213  | C4 93 | 11000100 10010011 |

## ▶ 符号位 (sign bit)

- 对于补码表示, MSB (Most Significant Bit) 表示整数的符号
  - 0 for nonnegative
  - 1 for negative

# 取值范围

## ▶ 无符号数

- $U_{Min} = 0$   
000...0
- $U_{Max} = 2^w - 1$   
111...1

## ▶ 带符号数（补码）

- $T_{Min} = -2^{w-1}$   
100...0
- $T_{Max} = 2^{w-1} - 1$   
011...1

## ▶ Other Values

- 负1 = 111...1

假设字长为16( $w=16$ )

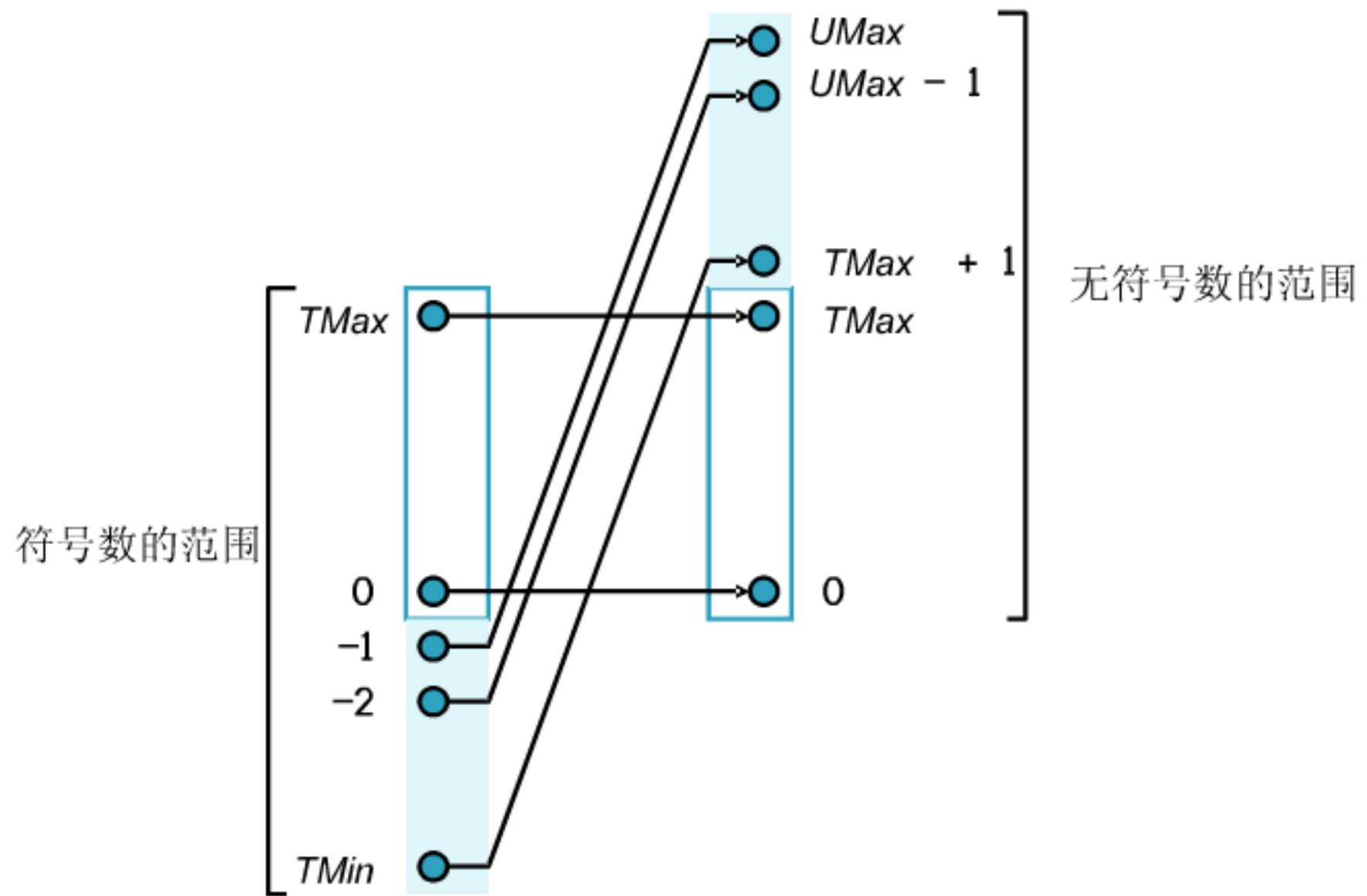
|      | Decimal | Hex   | Binary            |
|------|---------|-------|-------------------|
| UMax | 65535   | FF FF | 11111111 11111111 |
| TMax | 32767   | 7F FF | 01111111 11111111 |
| TMin | -32768  | 80 00 | 10000000 00000000 |
| -1   | -1      | FF FF | 11111111 11111111 |
| 0    | 0       | 00 00 | 00000000 00000000 |

# 无符号数与带符号数

| X    | B2U(X) | B2T(X) |
|------|--------|--------|
| 0000 | 0      | 0      |
| 0001 | 1      | 1      |
| 0010 | 2      | 2      |
| 0011 | 3      | 3      |
| 0100 | 4      | 4      |
| 0101 | 5      | 5      |
| 0110 | 6      | 6      |
| 0111 | 7      | 7      |
| 1000 | 8      | -8     |
| 1001 | 9      | -7     |
| 1010 | 10     | -6     |
| 1011 | 11     | -5     |
| 1100 | 12     | -4     |
| 1101 | 13     | -3     |
| 1110 | 14     | -2     |
| 1111 | 15     | -1     |

无符号数与带符号数之间的转换：  
二进制串的表示是不变的。

$$ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$$



# C语言中的无符号数与带符号数

- ▶ 常数 (Constants)
  - 默认是带符号数
    - 如果有“U”作为后缀则是无符号数，如 0U, 4294967259U
- ▶ 如果无符号数与带符号数混合使用，则带符号数默认转换为无符号数
  - 包括比较操作符
    - 实例 (w=32)

| Constant <sub>1</sub> | Constant <sub>2</sub> | Relation |
|-----------------------|-----------------------|----------|
| 0                     | 0U                    |          |
| -1                    | 0                     |          |
| -1                    | 0U                    |          |
| 2147483647            | -2147483647-1         |          |
| 2147483647U           | -2147483647-1         |          |
| -1                    | -2                    |          |
| (unsigned) -1         | -2                    |          |
| 2147483647            | 2147483648U           |          |
| 2147483647            | (int) 2147483648U     |          |

| <b>Constant<sub>1</sub></b> | <b>Constant<sub>2</sub></b> | <b>Relation</b> | <b>Evaluation</b> |
|-----------------------------|-----------------------------|-----------------|-------------------|
| 0                           | 0U                          | ==              | unsigned          |
| -1                          | 0                           | <               | signed            |
| -1                          | 0U                          | >               | unsigned          |
| 2147483647                  | -2147483647-1               | >               | signed            |
| 2147483647U                 | -2147483647-1               | <               | unsigned          |
| -1                          | -2                          | >               | signed            |
| (unsigned) -1               | -2                          | >               | unsigned          |
| 2147483647                  | 2147483648U                 | <               | unsigned          |
| 2147483647                  | (int) 2147483648U           | >               | signed            |

# 何时采用无符号数

- ▶ 模运算
- ▶ 按位运算
- ▶ 建议：不能仅仅因为取值范围是非负而使用

示例一

```
unsigned i;  
for (i = cnt-2; i >= 0; i--)  
    a[i] += a[i+1];
```

示例二

```
#define DELTA sizeof(int)  
int i;  
for (i = CNT; i-DELTA >= 0; i-= DELTA)
```

...

# 无符号数加法

Operands:  $w$  bits



True Sum:  $w+1$  bits

Discard Carry:  $w$  bits

$\text{UAdd}_w(u, v)$

$$s = \text{UAdd}_w(u, v) = (u + v) \bmod 2^w$$

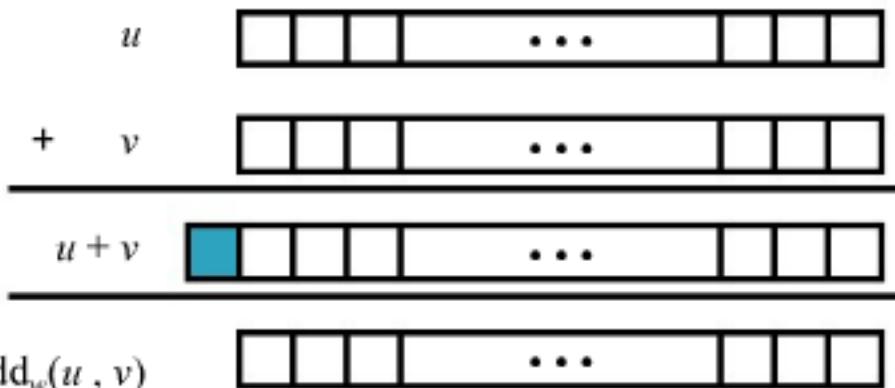
$$\text{UAdd}_w(u, v) = \begin{cases} u + v & u + v < 2^w \\ u + v - 2^w & u + v \geq 2^w \end{cases}$$

# 补码加法

Operands:  $w$  bits

True Sum:  $w+1$  bits

Discard Carry:  $w$  bits

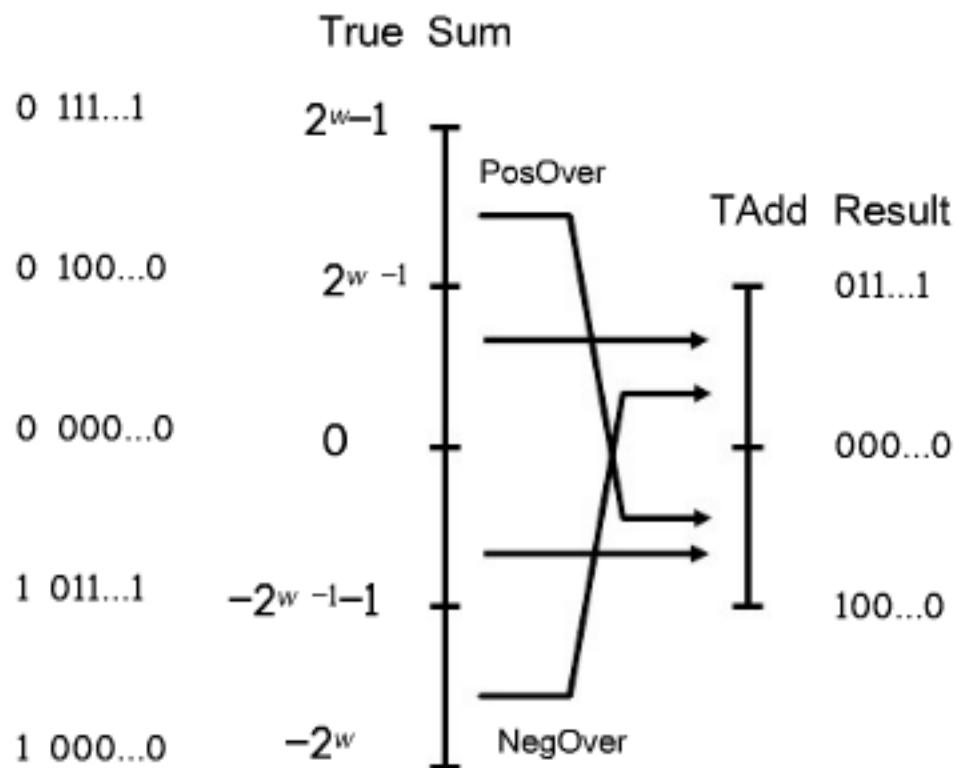


- ▶ 与无符号数的一致
  - Signed vs. unsigned addition in C:

```
int s, t, u, v;  
s = (int) ((unsigned) u + (unsigned) v);  
t = u + v
```

- $s == t$

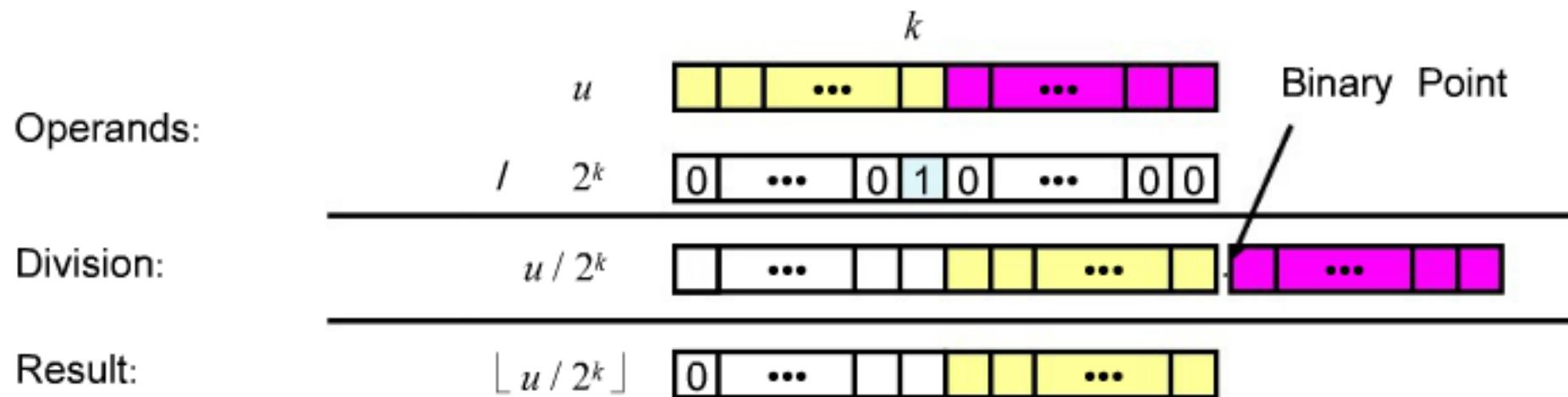
## 补码加法的溢出



$$TAdd_w(u, v) = \begin{cases} u + v + 2^w & u + v < TMin_w \text{ (NegOver)} \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^w & TMax_w < u + v \text{ (PosOver)} \end{cases}$$

# 无符号整数除以2的k次幂

- $u \gg k$  gives  $\lfloor u / 2^k \rfloor$
- 采用逻辑右移



|           | Division   | Computed | Hex   | Binary                   |
|-----------|------------|----------|-------|--------------------------|
| x         | 15213      | 15213    | 3B 6D | 00111011 01101101        |
| $x \gg 1$ | 7606.5     | 7606     | 1D B6 | <b>00011101</b> 10110110 |
| $x \gg 4$ | 950.8125   | 950      | 03 B6 | <b>00000011</b> 10110110 |
| $x \gg 8$ | 59.4257813 | 59       | 00 3B | <b>00000000</b> 00111011 |

## C Function

```
unsigned udiv8(unsigned x)
{
    return x/8;
}
```

## Compiled Arithmetic Operations

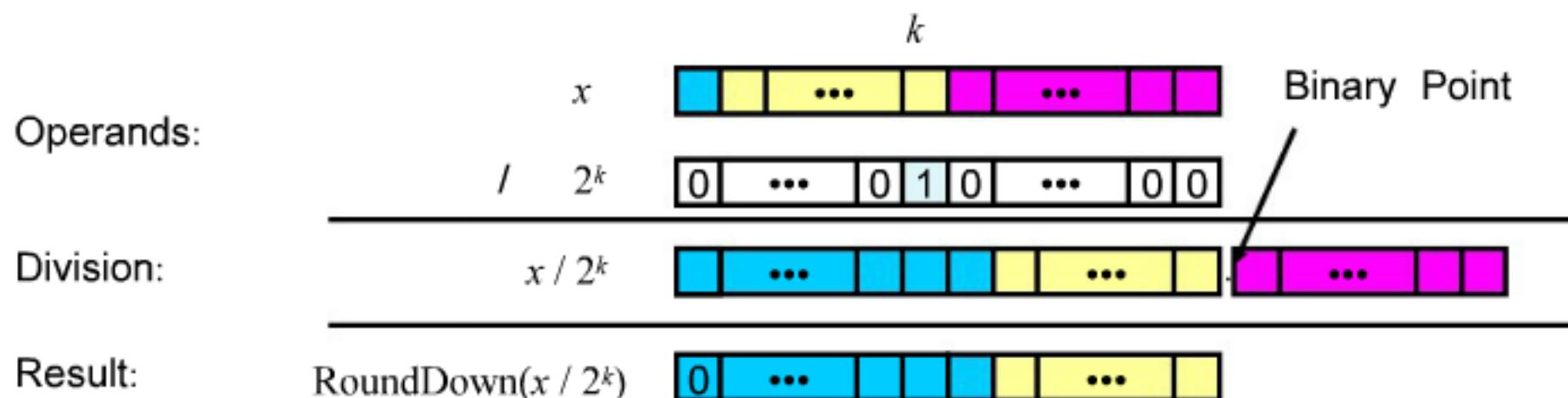
```
shrl $3, %eax
```

## Explanation

```
# Logical shift
return x >> 3;
```

# 带符号整数除以2的幂

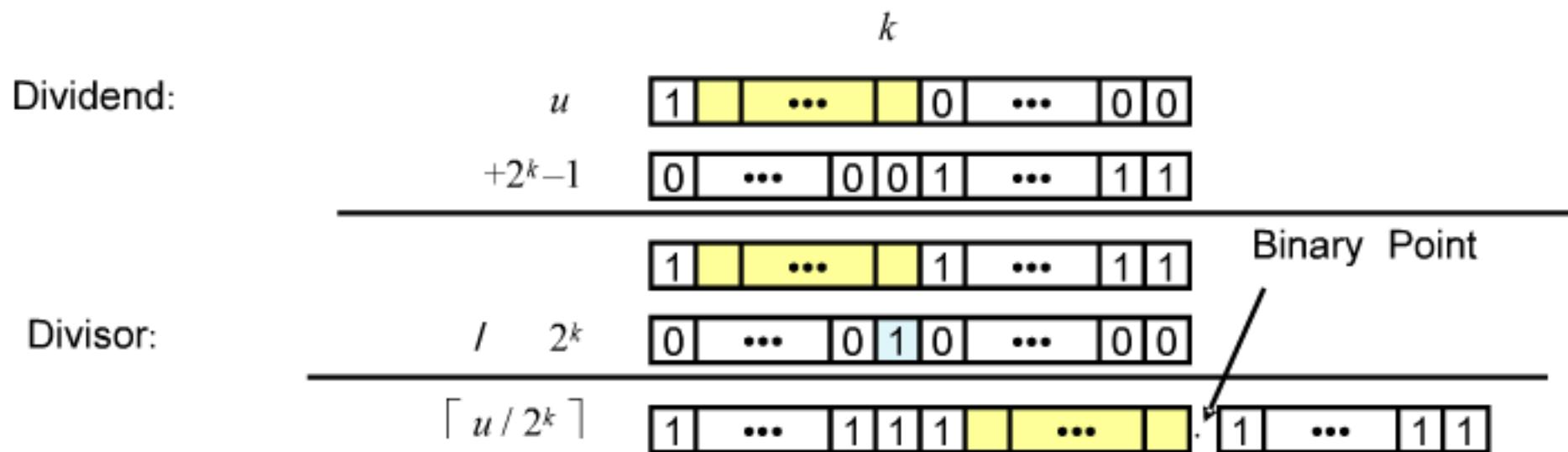
- $x \gg k$  gives  $\lfloor x / 2^k \rfloor$
- 采用算术右移
  - 但是  $x < 0$  时，舍入错误



|        | Division    | Computed | Hex   | Binary            |
|--------|-------------|----------|-------|-------------------|
| y      | -15213      | -15213   | C4 93 | 11000100 10010011 |
| y >> 1 | -7606.5     | -7607    | E2 49 | 11100010 01001001 |
| y >> 4 | -950.8125   | -951     | FC 49 | 11111100 01001001 |
| y >> 9 | -59.4257813 | -60      | FF C4 | 11111111 11000100 |

- Want  $\lceil x / 2^k \rceil$  (需要向0舍入，而不是向下舍入)
- Compute as  $\lfloor (x+2^{k-1}) / 2^k \rfloor$ 
  - In C: `(x + (1<<k)-1) >> k`
  - Biases dividend toward 0

## Case 1: No rounding



*Biasing has no effect*

## Case 2: Rounding

Dividend:  $x$

$+2^k - 1$

---

$1 \quad \dots \quad \dots \quad \dots \quad \dots$

$0 \quad \dots \quad 0 \ 0 \ 1 \quad \dots \quad 1 \ 1$

---

$1 \quad \dots \quad \dots \quad \dots \quad \dots$

Incremented by 1

Binary Point

Divisor:  $y / 2^k$

---

$0 \quad \dots \quad 0 \ 1 \ 0 \quad \dots \quad 0 \ 0$

$\lceil x / 2^k \rceil$

$1 \quad \dots \quad 1 \ 1 \ 1 \quad \dots \quad \dots$

Biasing adds 1 to final result

Incremented by 1

## C Function

```
int idiv8(int x)
{
    return x/8;
}
```

## Compiled Arithmetic Operations

```
testl %eax, %eax
js    L4
L3:
    sarl $3, %eax
    ret
L4:
    addl $7, %eax
    jmp  L3
```

## Explanation

```
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```

# Integer C Puzzles

- 判断以下的推断或者等式是否成立(不成立则给出示例)

- x, y 为32位带符号整数;

初始化

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- $x < 0 \Rightarrow ((x^2) < 0)$
- $ux \geq 0$
- $x \& 7 == 7 \Rightarrow (x << 30) < 0$
- $ux > -1$
- $x > y \Rightarrow -x < -y$
- $x * x \geq 0$
- $x > 0 \&& y > 0 \Rightarrow x + y > 0$
- $x \geq 0 \Rightarrow -x \leq 0$
- $x \leq 0 \Rightarrow -x \geq 0$
- $(x|-x) \gg 31 == -1$
- $ux \gg 3 == ux/8$
- $x \gg 3 == x/8$
- $x \& (x-1) != 0$

汇编语言程序设计

浮 点 数

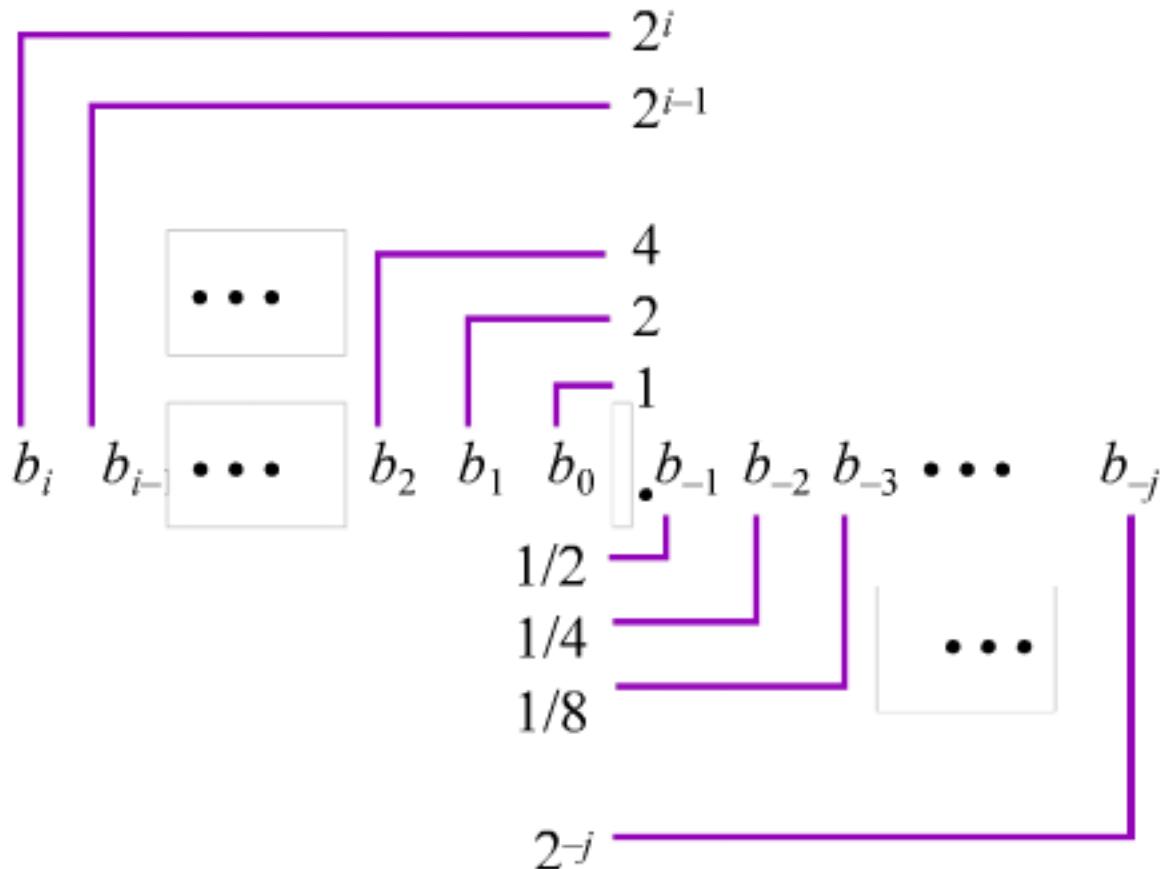
- ▶ IEEE的浮点数标准
- ▶ Rounding (舍入)
- ▶ C语言中的浮点数

# IEEE浮点数标准

- ▶ IEEE的754标准
  - 1985年建立

- ▶ 二进制表示方式

$$\sum_{k=-j}^i b_k \cdot 2^k$$



## 浮点数示例

### ▶ 值                  二进制表示

$5 - 3/4$                    $101.11_2$

$2 - 7/8$                    $10.111_2$

$63/64$                    $0.111111_2$

### ▶ 局限性

- 只能精确地表示 $x/2^k$ 这类形式的数据（ $k$ 为整数）

### ▶ 值                  二进制表示

$1/3$                    $0.0101010101[01] \cdots_2$

$1/5$                    $0.001100110011[0011] \cdots_2$

$1/10$                    $0.0001100110011[0011] \cdots_2$

# 计算机中的浮点数二进制表示

## ▶ 数字形式

- $(-1)^s \ M \ 2^E$ 
  - 符号:  $s$
  - 尾数:  $M$ , 是一个位于区间  $[1.0, 2.0)$  内的小数
  - 阶码:  $E$

## ▶ 编码



- exp域:  $E$
- frac域:  $M$



单精度浮点数： exp域宽度为8 bits, frac域宽度为23  
bits  
总共32 bits。

双精度浮点数： exp域宽度为11 bits, frac域宽度为  
52bits  
总共64 bits。

扩展精度浮点数： exp域宽度为15 bits, frac域宽  
度为63bits  
总共80 bits。 (*1 bit wasted*)

## ▶ 浮点数的类型

- 规格化浮点数
- 非规格化浮点数
- 一些特殊值

## 规格化浮点数 ( Normalized )

- ▶ 满足条件

- $\text{exp} \neq 000\cdots 0$  且  $\text{exp} \neq 111\cdots 1$

- ▶ 真实的阶码值需要减去一个偏置 (biased) 量

$$E = \text{Exp} - \text{Bias}$$

- Exp : exp域所表示的无符号数值
  - Bias的取值
    - 单精度数: 127 (Exp: 1…254, E: -126…127)
    - 双精度数: 1023 (Exp: 1…2046, E: -1022…1023)
      - $\text{Bias} = 2^{e-1} - 1$ , e = exp域的位数

- ▶ frac域的第一位隐含为1

$$M = 1.\text{xxxx}\dots x_2$$

- 因此, 第一位的“1”可以省去, xxxx…x: bits of frac
  - Minimum when 000…0 ( $M = 1.0$ )
  - Maximum when 111…1 ( $M = 2.0 - \varepsilon$ )

# 规格化浮点数示例

Float F = 15213.0

◦  $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$

尾数

$M = 1.\underline{1101101101101}_2$

$\text{frac} = \underline{1101101101101}0000000000_2$

阶码

$E = 13$

$Bias = 127$

$Exp = 140 = 10001100_2$

Hex: 4 6 6 D B 4 0 0

Binary: 0100 0110 0110 1101 1011 0100 0000 0000

140: 100 0110 0

15213: **1**110 1101 1011 01

# 非规格化浮点数 ( Denormalized )

- ▶ 满足条件
  - $\text{exp} = 000\cdots0$
- ▶ 其它域的取值
  - $E = -\text{Bias} + 1$
  - $M = 0.\text{xxx}\cdots\text{x}_2$ 
    - $\text{xxx}\cdots\text{x}$ : bits of `frac`
- ▶ 具体示例
  - $\text{exp} = 000\cdots0, \text{frac} = 000\cdots0$ 
    - 表示0
    - 注意有 +0 与 -0
  - $\text{exp} = 000\cdots0, \text{frac} \neq 000\cdots0$ 
    - 表示“非常接近”于0的浮点数
    - 会逐步丧失精度
      - 称为“Gradual underflow”

# 一些特殊值

## ▶ 满足条件

- $\exp = 111\cdots 1$

## ▶ 具体示例

- $\exp = 111\cdots 1, \frac{1}{0} = 000\cdots 0$ 
  - 表示无穷 ◎
    - 可用于表示数值的溢出
    - 有“正无穷”与“负无穷”
    - E. g.,  $1.0/0.0 = +\infty, -1.0/0.0 = -\infty$
  - $\exp = 111\cdots 1, \frac{1}{0} \neq 000\cdots 0$ 
    - Not-a-Number (NaN)
    - E. g.,  $\sqrt{-1}, \infty - \infty, \infty * 0$

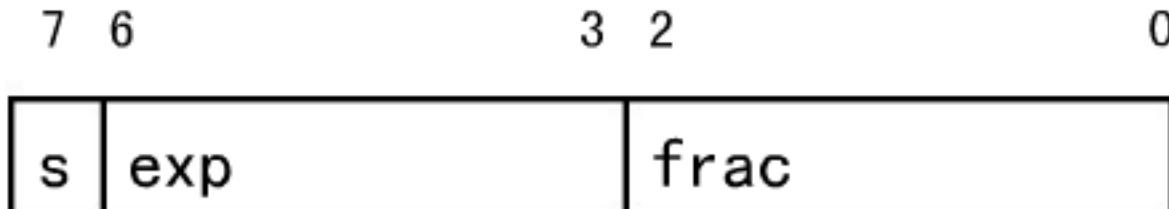
# 一种“小”浮点数实例

- ▶ 8位浮点数表示

- exp域宽度为4 bits, frac域宽度为3bits

- 其他规则符合IEEE 754规范

- 规格化 / 非规格化
  - 表示0, NaN与无穷



| Exp | <b>exp</b> | E   | $2^E$ |           |
|-----|------------|-----|-------|-----------|
| 0   | 0000       | -6  | 1/64  | (非规格化数)   |
| 1   | 0001       | -6  | 1/64  |           |
| 2   | 0010       | -5  | 1/32  |           |
| 3   | 0011       | -4  | 1/16  |           |
| 4   | 0100       | -3  | 1/8   |           |
| 5   | 0101       | -2  | 1/4   |           |
| 6   | 0110       | -1  | 1/2   |           |
| 7   | 0111       | 0   | 1     |           |
| 8   | 1000       | +1  | 2     |           |
| 9   | 1001       | +2  | 4     |           |
| 10  | 1010       | +3  | 8     |           |
| 11  | 1011       | +4  | 16    |           |
| 12  | 1100       | +5  | 32    |           |
| 13  | 1101       | +6  | 64    |           |
| 14  | 1110       | +7  | 128   |           |
| 15  | 1111       | n/a |       | (无穷, NaN) |

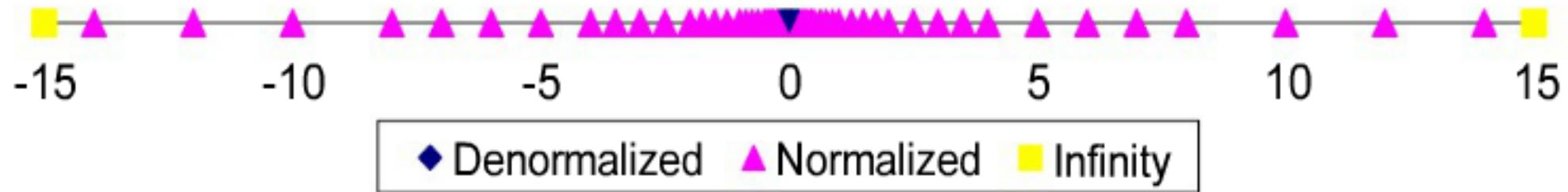
# 取值范围

非规格化数

| s     | exp  | frac | E   | Value                |             |
|-------|------|------|-----|----------------------|-------------|
| 0     | 0000 | 000  | -6  | 0                    |             |
| 0     | 0000 | 001  | -6  | $1/8 * 1/64 = 1/512$ | ← 接近于0      |
| 0     | 0000 | 010  | -6  | $2/8 * 1/64 = 2/512$ |             |
| ...   |      |      |     |                      |             |
| 0     | 0000 | 110  | -6  | $6/8 * 1/64 = 6/512$ |             |
| 0     | 0000 | 111  | -6  | $7/8 * 1/64 = 7/512$ | ← 最大的非规格化数  |
| ..... |      |      |     |                      |             |
| 0     | 0001 | 000  | -6  | $8/8 * 1/64 = 8/512$ | ← 最小的规格化数   |
| 0     | 0001 | 001  | -6  | $9/8 * 1/64 = 9/512$ |             |
| ...   |      |      |     |                      |             |
| 0     | 0110 | 110  | -1  | $14/8 * 1/2 = 14/16$ |             |
| 0     | 0110 | 111  | -1  | $15/8 * 1/2 = 15/16$ | ← 接近于1 (<1) |
| 0     | 0111 | 000  | 0   | $8/8 * 1 = 1$        |             |
| 0     | 0111 | 001  | 0   | $9/8 * 1 = 9/8$      | ← 接近于1 (>1) |
| 0     | 0111 | 010  | 0   | $10/8 * 1 = 10/8$    |             |
| ...   |      |      |     |                      |             |
| 0     | 1110 | 110  | 7   | $14/8 * 128 = 224$   |             |
| 0     | 1110 | 111  | 7   | $15/8 * 128 = 240$   | ← 最大的规格化数   |
| 0     | 1111 | 000  | n/a | inf                  |             |

规格化数

# 数轴上的分布



## 一些特例

| Description                             | exp     | frac    | Numeric Value                               |
|---|---------|---------|---|
| Zero                                    | 00...00 | 00...00 | 0.0   |
| Smallest Pos. Denorm.                   | 00...00 | 00...01 | $2^{-\{23,52\}} \times 2^{-\{126,1022\}}$   |
| ◦ Single $\approx 1.4 \times 10^{-45}$  |         |         |   |
| ◦ Double $\approx 4.9 \times 10^{-324}$ |         |         |   |
| Largest Denormalized                    | 00...00 | 11...11 | $(1.0 - \epsilon) \times 2^{-\{126,1022\}}$ |
| ◦ Single $\approx 1.18 \times 10^{-38}$ |         |         |   |
| ◦ Double $\approx 2.2 \times 10^{-308}$ |         |         |   |
| Smallest Pos. Normalized                | 00...01 | 00...00 | $1.0 \times 2^{-\{126,1022\}}$              |
| ◦ Just larger than largest denormalized |         |         |   |
| One                                     | 01...11 | 00...00 | 1.0   |
| Largest Normalized                      | 11...10 | 11...11 | $(2.0 - \epsilon) \times 2^{\{127,1023\}}$  |
| ◦ Single $\approx 3.4 \times 10^{38}$   |         |         |   |
| ◦ Double $\approx 1.8 \times 10^{308}$  |         |         |   |

# 浮点数的一些编码特性

- ▶ (几乎) 可以直接使用无符号整数的比较方式
  - 反例：
    - 必须先比较符号位
    - 考虑+0、-0的特例
    - 还有NaN的问题…
      - (不考虑符号位的话), NaN比其他值都大
      - 实际的比较结果如何?
  - 其他情况都可以直接使用无符号整数的比较方式
    - 规格化 vs. 非规格化
    - 规格化 vs. 无穷

## 非规格化数

| s     | exp  | frac | E   | Value                |             |
|-------|------|------|-----|----------------------|-------------|
| 0     | 0000 | 000  | -6  | 0                    |             |
| 0     | 0000 | 001  | -6  | $1/8 * 1/64 = 1/512$ | ← 接近于0      |
| 0     | 0000 | 010  | -6  | $2/8 * 1/64 = 2/512$ |             |
| ...   |      |      |     |                      |             |
| 0     | 0000 | 110  | -6  | $6/8 * 1/64 = 6/512$ |             |
| 0     | 0000 | 111  | -6  | $7/8 * 1/64 = 7/512$ | ← 最大的非规格化数  |
| ..... |      |      |     |                      |             |
| 0     | 0001 | 000  | -6  | $8/8 * 1/64 = 8/512$ | ← 最小的规格化数   |
| 0     | 0001 | 001  | -6  | $9/8 * 1/64 = 9/512$ |             |
| ...   |      |      |     |                      |             |
| 0     | 0110 | 110  | -1  | $14/8 * 1/2 = 14/16$ |             |
| 0     | 0110 | 111  | -1  | $15/8 * 1/2 = 15/16$ | ← 接近于1 (<1) |
| 0     | 0111 | 000  | 0   | $8/8 * 1 = 1$        |             |
| 0     | 0111 | 001  | 0   | $9/8 * 1 = 9/8$      | ← 接近于1 (>1) |
| 0     | 0111 | 010  | 0   | $10/8 * 1 = 10/8$    |             |
| ...   |      |      |     |                      |             |
| 0     | 1110 | 110  | 7   | $14/8 * 128 = 224$   |             |
| 0     | 1110 | 111  | 7   | $15/8 * 128 = 240$   | ← 最大的规格化数   |
| 0     | 1111 | 000  | n/a | inf                  |             |

## 规格化数

给定一个实数，如何给出其浮点数表示？

▶ 基本流程

- 首先计算出精确值
- 然后将其转换为所需的精度
  - 可能会溢出（如果指数绝对值很大）
  - 可能需要完成舍入(rounding)操作

▶ 各种舍入模式

|                          |         |         |         |         |           |
|--------------------------|---------|---------|---------|---------|-----------|
|                          | \$1. 40 | \$1. 60 | \$1. 50 | \$2. 50 | - \$1. 50 |
| ◦ Zero                   | \$1     | \$1     | \$1     | \$2     | - \$1     |
| ◦ Round down             | \$1     | \$1     | \$1     | \$2     | - \$2     |
| ◦ Round up               | \$2     | \$2     | \$2     | \$3     | - \$1     |
| ◦ Nearest Even (default) | \$1     | \$2     | \$2     | \$2     | - \$2     |

给定一个实数，如何给出其浮点数表示？

▶ 基本流程

- 首先计算出精确值
- 然后将其转换为所需的精度
  - 可能会溢出（如果指数绝对值很大）
  - 可能需要完成舍入(rounding)操作

▶ 各种舍入模式

|                          |         |         |         |         |           |
|--------------------------|---------|---------|---------|---------|-----------|
|                          | \$1. 40 | \$1. 60 | \$1. 50 | \$2. 50 | - \$1. 50 |
| ◦ Zero                   | \$1     | \$1     | \$1     | \$2     | - \$1     |
| ◦ Round down             | \$1     | \$1     | \$1     | \$2     | - \$2     |
| ◦ Round up               | \$2     | \$2     | \$2     | \$3     | - \$1     |
| ◦ Nearest Even (default) | \$1     | \$2     | \$2     | \$2     | - \$2     |

## 向偶数舍入 (Round-To-Even)

- ▶ 这是计算机内默认的舍入方式，也称为“向最接近值的舍入”
  - ▶ 其它方式会产生系统误差
- ▶ 关键的设计决策是确定两个可能结果的中间数值的舍入
  - 确保舍入后的最低有效数字是偶数
  - 比如：向百分位舍入

|           |      |                         |
|-----------|------|-------------------------|
| 1.2349999 | 1.23 | (Less than half way)    |
| 1.2350001 | 1.24 | (Greater than half way) |
| 1.2350000 | 1.24 | (Half way—round up)     |
| 1.2450000 | 1.24 | (Half way—round down)   |

## 向偶数舍入 (Round-To-Even)

- ▶ 对于二进制数而言
  - “Even” 意味着最低有效数字需为0
  - 而最低有效数字右侧的位串为100…。

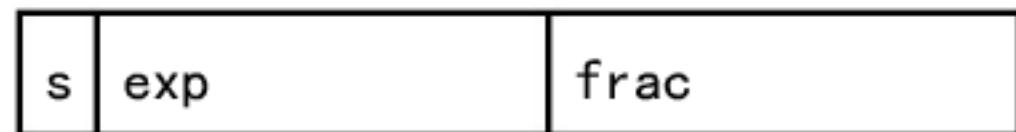
- ▶ 实例
  - 舍入到小数点后2位

|        | Value                 | Binary             | Rounded     | Action | Rounded Value |
|--------|-----------------------|--------------------|-------------|--------|---------------|
| 2 3/32 | 10.00011 <sub>2</sub> | 10.00 <sub>2</sub> | (<1/2—down) | 2      |               |
| 2 3/16 | 10.00110 <sub>2</sub> | 10.01 <sub>2</sub> | (>1/2—up)   | 2 1/4  |               |
| 2 7/8  | 10.11100 <sub>2</sub> | 11.00 <sub>2</sub> | (1/2—up)    | 3      |               |
| 2 5/8  | 10.10100 <sub>2</sub> | 10.10 <sub>2</sub> | (1/2—down)  | 2 1/2  |               |

7 6

3 2

0



## ▶ 具体步骤

- 将数值规格化（小数点前为1）
- 舍入（round to even）以便符合尾数的位数需求
- 后调整

## ▶ 实例

- 将8位无符号数转换为8位浮点数（exp域宽度为4 bits, frac域宽度为3bits）

128      10000000

15        00001101

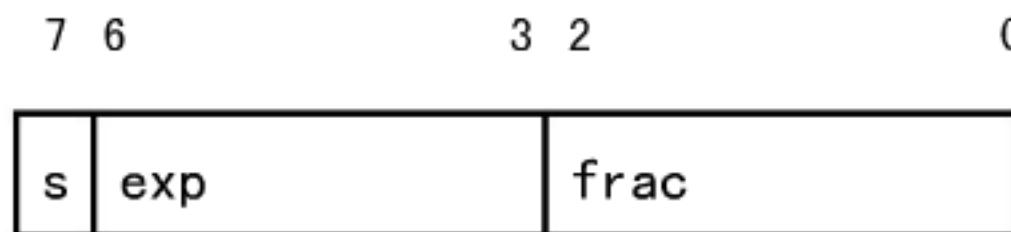
33        00010001

35        00010011

138      10001010

63        00111111

# 规格化



| Value | Binary   | Fraction  | Exponent |
|-------|----------|-----------|----------|
| 128   | 10000000 | 1.0000000 | 7        |
| 15    | 00001101 | 1.1010000 | 3        |
| 17    | 00010001 | 1.0001000 | 4        |
| 19    | 00010011 | 1.0011000 | 4        |
| 138   | 10001010 | 1.0001010 | 7        |
| 63    | 00111111 | 1.1111100 | 5        |

# 舍入

| Value | Fraction           | Incr? | Rounded |
|-------|--------------------|-------|---------|
| 128   | 1. 000 <b>0000</b> | N     | 1. 000  |
| 15    | 1. 101 <b>0000</b> | N     | 1. 101  |
| 17    | 1. 000 <b>1000</b> | N     | 1. 000  |
| 19    | 1. 001 <b>1000</b> | Y     | 1. 010  |
| 138   | 1. 000 <b>1010</b> | Y     | 1. 001  |
| 63    | 1. 111 <b>1100</b> | Y     | 10. 000 |

# 调整 (Postnormalize)

- 舍入操作可能引起溢出

| Value | Rounded | E | Adjusted | Result |
|-------|---------|---|----------|--------|
| 128   | 1.000   | 7 |          | 128    |
| 15    | 1.101   | 3 |          | 15     |
| 17    | 1.000   | 4 |          | 16     |
| 19    | 1.010   | 4 |          | 20     |
| 138   | 1.001   | 7 |          | 134    |
| 63    | 10.000  | 5 | 1.000/6  | 64     |

# C语言中的浮点数

|        |        |
|--------|--------|
| float  | 单精度浮点数 |
| double | 双精度浮点数 |

## ▶ 类型转换

- 当int（32位宽），float，与double等类型间进行转换时，基本的原则如下：
- double或float 转换为int
  - 尾数部分截断
  - 如果溢出或者浮点数是NaN，则转换结果没有定义
  - 通常置为 Tmin or Tmax
- int转换为double
  - 能够精确转换
- int转换为float
  - 不会溢出，但是可能被舍入（即精度受损）

◦ 以下判断是否成立，如不成立请给出反例。

```
int x = ...;
float f = ....;
double d = ...;
```

假设d 与 f 都不是 NaN

- $x == (int)(float) x$
- $x == (int)(double) x$
- $f == (float)(double) f$
- $d == (float) d$
- $f == -(-f);$
- $2/3 == 2/3.0$
- $d < 0.0 \quad \text{?} \quad ((d*2) < 0.0)$
- $d > f \quad \text{?} \quad -f > -d$
- $d * d >= 0.0$
- $(d+f)-d == f$

给定一个浮点格式，有  $k$  位指数和  $n$  位小数，对于下列数，写出阶码  $E$ 、尾数  $M$ 、小数  $f$  和值  $V$  的公式。另外，请描述其位表示。

- A. 数 5.0。
- B. 能够被准确描述的最大奇整数。
- C. 最小的正规格化数。