# CS:APP阅读笔记

## Chapter 01 计算机系统漫游

- 后继处理器的设计具有**后向兼容性**(backward compatibility), 即新的处理器可以运行为老处理器编译的代码.

  > **backward**-compatibility ~ ~~后向兼容~~ ~ 兼容**过去**
  >
  > **forward**-compatibility ~ ~~前向兼容~~ ~ 兼容**未来**
  >
  > ~~后向/前向~~ 的翻译可能造成迷惑, 理解为 过去/未来 可能有帮助.

- `hello.c` ->[^预处理] `hello.i` ->[^编译] `hello.s` ->[^汇编] `hello.o` + `printf.o` ->[^链接] `hello`

```c
printf("0x%04x", pa); // 4表示输出的宽度 至少为4个字符; 0表示不足4个字符的部分用0来补充
printf("0x%04lx", pa); // %lx 表示类型为 unsigned long, %x 表示 unsigned int
```

- (p110)精通**细节**是理解是理解更深和更基本概念的先决条件. “理解一般规则、但不愿意去劳神细节”是在自欺欺人

## Chapter 02 信息的表示和处理

- 十进制个位数 $x$ 的ASCII码为 `0x3x` , 比如 $3$ 的 ASCII 码为 `0x33` .
- 估算一个十进制数的二进制长度:

  由于 $2^{10} \approx_{(>)} 10^3$ , 十进制的3位 ~ 二进制的10位.

> 例如1234567, 7位, 所以二进制最多是7*10/3=23位(而且这个最多是可以保证的, 因为上述估计中, 还有2^10 > 10^3)

- **MSByte**: most significant byte 高字节

  **LSByte**: least significant byte 低字节
- “小端法(little endian)”最为常用, 但“大端法”的写法更符合书写时的直觉

  > 其实小端法很符合编程思维, 比如用数组实现"HugeInt类"时, 就是a[0],...,a[n]分别代表从低位到高位.

- 位运算 $XOR$ 满足**交换律**和**结合律**, 这是因为 $XOR$ 其实就是整数的 `mod2加法` .
- > 补码的英文是 Two's Complement, 来源于事实: $-x = 2^w - x$ , 因为补码的"0"其实是靠"溢出"得到的(?)
  >
  > 反码的英文是 Ones' Complement, 来源于事实: -x = **[11...1]** - x, 因为反码的其中一个"0"就是[11...1]

- `<limit.h>` 中定义了 `INT_MAX`, `INT_MIN`, `UINT_MAX` 常量
  `<stdint.h>` 中定义了 `INTN_MIN`, `INTN_MAX`, `UINTN_MAX` 常量, 以及 `intN_t`, `uintN_t` 类型.
- `UMAX` 和 `-1` 有相同的位模式.
- C语言中, 两个数进行运算, 如果其中一个是**unsigned**, 就把两个数的 bit 都解析成**unsigned**.
  这在算术运算时无所谓, 但在进行关系运算时, 会产生非直观结果. 比如**sizeof()**返回的是**unsigned**, 从而 `i-sizeof(t) >= 0` 恒成立.
- 当把 **short** 转换成**unsigned**时, C标准规定, 要**先**改变大小(符号拓展到int), **再**转为 unsigned (更换bit的解析方式).

  > 这种规定是有道理的. 否则试想: 假设 x 是负数, 我们**改为**对它先转unsigned, 那么改变大小时则不是符号拓展(而是0拓展),
  > 则即使后来我们试图重新用有符号的方式来理解x的位向量, 比如: int y = int(unsigned(x)) , y 也不再等于 x 了.

- 检测**无符号加法**溢出:
  $c \triangleq a + b\, \text{overflow} \iff \text{c} < \text{a} \iff \text{c} < \text{b}$
  检测**补码加法**溢出:
  $c \triangleq a + b\, \text{overflow} \iff \text{a} < 0, \text{b} < 0, \text{c} \geq 0 \iff \text{a} > 0, \text{b} > 0, \text{c} \leq 0$
  即: **溢出** `当且仅当` **产生数学错误**

  ```
  -T_MIN == T_MIN;                      // 构造反例时多留意 T_MIN
  |T_MIN| - |T_MAX| == 1;       // <负数>和<非负数>一样多
  -x == ~x + 1;                             // 人脑计算 -x 的技巧
  ```

- $\lceil \frac{x}{y} \rceil = \lfloor \frac{x}{y} + \frac{y-1}{y} \rfloor$ , 其中 $\frac{y-1}{y}$ 称为偏置项 $Bias$ .
  在C语言中, 除法被规定为向0舍入, 但右移运算由于吃掉了低位bit, 所以总是向下舍入的. 这使得对于负数, 无脑直接右移不是对除以2的幂的正确优化. 我们需要利用上述公式, 对于负数的情形, 加入偏置项即可. 比如 `-123 / 8` 就可以被优化为 `(-123 + 7) >> 3`

  ```
  int result = (x < 0 ? x + ((1<<k)-1) : x) >> k; // 利用上述原理, 通过右移来计算 x / (2^k)
  ```

- `数学概念`: `符号S`, `阶码E`, `尾数M(Mantissa)`, 有公式: $\text{value} = (-1)^S \cdot 2^E \cdot M$
  `编码概念`: `符号位(s)`, `阶码字段(exp),长度记为k`, `尾数字段(frac),长度记为n`
- IEEE浮点数的位向量形式: `s|exp|frac`
  **float**中, 有8bit的exp, 23bit的frac
  **double**中, 有11bit的exp, 52bit的frac

> 可见 **double** 相比 **float** 主要是提升精度(frac), 而阶码字段(exp)仅仅增加了3个bit.

- 当阶码字段的bit全为0时, 称为 `非规格化数`
  当阶码字段的bit全为1时, 称为 `特殊值`
  否则, 称为 `规格化值`
- 定义 $\text{Bias} = 2^{k-1} - 1$ (也就是阶码字段为011...1时对应的无符号数的值)

  > 特别地, 对于 **float** 是 $2^7 - 1 = 127$, 对于 **double** 是 $2^{10} - 1 = 1023$
  >
  > ~~(其实Bias定义为别的, 比如说是 2^(k-1) , 似乎也没什么不对的, 但这里是IEEE的规定...)~~

- 从 **`编码概念`** 到 **`数学概念`** 的转化:
  - `规格化` : $E = B2T(exp) - \text{Bias}$ ; $M = \overline{1.frac}_2$ **"隐含的1.开头"**
  - `非规格化` : $E \equiv 1 - \text{Bias}\ (= -2^{k-1} + 2)$ ; $M = \overline{0.frac}_2$ **"隐含的0.开头"**
  - `特殊值` :
    - 无穷: `s|1...1|0...0` 表示 $(-1)^S \cdot \infty$ , 包含 $\pm\infty$
    - NaN: `s|1...1|这里有1` 表示 $NaN$ , **包含±NaN**
- 关于"非规格化到规格化的**平滑过渡**"的设计思路:
  **规格化数**的 `阶码字段exp` 最小为 `00...01` , 对应于阶码 $E = 1 - \text{Bias}$, 而 `尾数M` 则**最小**为$\overline{1.00...0}$ ;
  而**非规格化数**的 `尾数M` **最大**为$\overline{0.11...1}$, 阶码则是**一个待规定**的常数,
  易见, 如果将阶码**规定为1 - Bias**, 就刚刚好和**规格化数**的**最小数**只相差一丢丢 $(\overline{1.00...0} - \overline{0.11...1})$
- **int** 转 **double** 不会丢失精度, 因为 **double** 中有至少53bit的精度, 53 > 32, 所以 **(int)(double)x == x**
- 浮点数运算结果将使用"**向偶数舍入**"的规则, 即一般而言"四舍五入", 但如果恰巧为"50...0", 则舍入到能使得舍入结果的末bit为偶数(0)的情况.

  > [举例] 保留1位小数的向偶数舍入:
  >
  > 11.001 --> 11.0 没啥纠结的, 小于一半, 舍去.
  >
  > 10.011 --> 10.1 没啥纠结的, 大于一半, 进位.
  >
  > 10.010 --> 10.0 正好在中间, 若进位则得到10.1, 若舍弃则得到10.0, 按照规则选择舍弃.
  >
  > 10.110 --> 11.0 正好在中间, 若进位则得到11.0, 若舍弃则得到10.1, 按照规则选择进位.

- 但 **double / float** 转 **int** 会**向0舍入** , (和 int 除法的舍入规则相同)
  如果转换时发生溢出, C语言规定结果为 $[100...00](-2147483648)$ .
- (p82)

| 一些重要的**非负数** | exp | frac | 备注 |
|---|---|---|---|
| 0 | 00...00 | 00...00 | |
| 最小非规格化数 | 00...00 | 00...0**1** | |
| 最大正非规格化数 | 00...00 | **11...11** | |
| 最小正规格化数 | 00...0**1** | 00...00 | |
| | | | |

| | | | |
|---|---|---|---|
| 1 | 0**1...11** | 00...00 | 因为Bias的位模式为[011...11] |
| 最大规格化数 | 11...1**0** | 11...11 | exp不是全1! 全1用于编码±无穷或NaN! |

---

End Of Note, 148 Lines.