第 3 章

2 的幂边界

3.1 将数值上调/下调为 2 的已知次幂的倍数

如果想将无符号整数 x 下调为一个值最接近它同时还是 8 的倍数的数，那么只需执行 $x \,\&\, -8$ 就可以了。另外一种办法是 $(x \overset{u}{\gg} 3) \ll 3$。只要我们规定“下调”指的是向负无穷方向调整，那么上面两种算法就同样适用于带符号的整数了（例如 $(-37)\,\&\,(-8) = -40$）。

上调与下调一样容易。比方说，想将无符号整数 x 上调为值最接近它而又能被 8 整除的数，那么可以使用下列两式计算：

$$(x+7)\,\&\,-8 \text{ 或}$$
$$x+(-x\,\&\,7)$$

只要我们规定“上调”的意思是向正无穷方向取整，那么这两个表达式就同样适用于带符号的整数。第 2 个表达式的第 2 项很有用，如果你想知道最少给 x 加上几才能把它变成 8 的倍数，那么用 $-x\,\&\,7$ 就可以算出来 [Gold]。

如果想把一个带符号整数朝 0 所在的方向调整为值最接近它且能被 8 整除的数，那么可以把上面两个表达式组合一下，这样就能得到下面这个算法了：

$$t \leftarrow (x \overset{s}{\gg} 31)\,\&\,7;$$
$$(x+t)\,\&\,-8$$

上面这种算法的第 1 行也可以改成 $t \leftarrow (s \overset{s}{\gg} 2) \overset{u}{\gg} 29$。如果计算机没有和常量求与（and immediate）的指令，或是常量太大，没办法放在指令码的常量位段中，那么这个表达式就能派上用场了。

有些时候，舍入因子（rounding factor）不是对齐量，而是把对齐量[⊖]取以 2 为底的对数，然后用这个对数值来表示舍入因子（比方说，舍入因子值为 3，意思就是朝 8 的倍

⊖ alignment amount，也译作“调整量”、“调整边界”、“对齐边界”等，也就是舍入数值后的标准位置。舍入之后的数值必须是该值的整数倍。——译者注

数对齐[㊀])。在这种情况下，可以按照下列代码来舍入，其中 $k=\log_2$（对齐量）：

向下调整：　$x \,\&((-1)\ll k)$

$(x \overset{u}{\gg} k)\ll k$

向上调整：　$t\leftarrow(1\ll k)-1;(x+t)\,\&\neg t$

$t\leftarrow(-1)\ll k;(x-t-1)\,\&\,t$

3.2　调整到上一个/下一个 2 的幂

现在定义两个与“向下取整”和“向上取整”类似的函数，只是我们这次把舍入之后的结果规定为距离自变量最近的那个 2 的整数幂，而不是像原来那样调整到距其最近的整数。这两个函数的数学定义是：

$$\mathrm{flp2}(x)=\begin{cases}\text{未定义}, & x<0,\\ 0, & x=0,\\ 2^{\lfloor\log_2 x\rfloor}, & \text{其他};\end{cases}\qquad \mathrm{clp2}(x)=\begin{cases}\text{未定义}, & x<0,\\ 0, & x=0,\\ 2^{\lfloor\log_2 x\rfloor}, & \text{其他}.\end{cases}$$

函数名的头两个字母分别表示“floor”（向下取整）与“ceiling”（向上取整）。因此 $\mathrm{flp2}(x)$ 是小于等于 x 且最接近 x 的 2 的整数幂，而 $\mathrm{clp2}(x)$ 是大于等于 x 且最接近 x 的 2 的整数幂。即便 x 不是整数，也还是可以套用这两个函数的定义（例如，$\mathrm{flp2}(0.1)=0.0625$）[㊁]。这两个函数也满足一些与向上取整和向下取整函数类似的关系式。下面列出几个这样的式子，其中 n 为整数：

$\lfloor x\rfloor=\lceil x\rceil$　当且仅当 x 为整数　　$\mathrm{flp2}(x)=\mathrm{clp2}(x)$　当且仅当 x 是 2 的幂或 0

$\lfloor x+n\rfloor=\lceil x\rceil+n$　　$\mathrm{flp2}(2^n x)=2^n\mathrm{flp2}(x)$

$\lceil x\rceil=-\lfloor -x\rfloor$　　$\mathrm{clp2}(x)=1/\mathrm{flp2}(1/x),\ x\neq 0$

实际计算中我们只处理 x 为整数的情形，而且将其定为无符号数，这样的话，上面两个函数对所有的 x 值就都有定义了。此外，我们还规定，这些函数的值是将算数运算的正确结果与 2^{32} 取模后得到的（也就是说，如果 $\mathrm{clp2}(\boldsymbol{x})$ 中的 $\boldsymbol{x}$ 大于 $\boldsymbol{2^{31}}$，那么此函数的值就是 0）。下表列出了一些 $\boldsymbol{x}$ 的函数值。

$\boldsymbol{x}$	$\mathrm{flp2}(\boldsymbol{x})$	$\mathrm{clp2}(\boldsymbol{x})$
0	0	0
1	1	1
2	2	2
3	2	4
4	4	4

㊀　因为 $\log_2 8=\log_2 2^3=3$。——译者注

㊁　因为 $2^{-3}=0.125$，$2^{-4}=0.0625$，0.1 的值位于两者之间，所以对齐向下调整，就是 0.0625。——译者注

5	4	8
…	…	…
$2^{31}-1$	2^{30}	2^{31}
2^{31}	2^{31}	2^{31}
$2^{31}+1$	2^{31}	0
…	…	…
$2^{32}-1$	2^{31}	0

下面列出了flp2与clp2函数之间的关系式。在给定的条件下，可以利用这些关系式，根据其中一个函数的值，推算出另一个函数。

$$\begin{aligned}\mathrm{clp2}(\boldsymbol{x}) &= 2\,\mathrm{flp2}(\boldsymbol{x}-\mathbf{1}), && \boldsymbol{x}\neq 1,\\ &\quad \mathrm{flp2}(2\boldsymbol{x}-\mathbf{1}), && \mathbf{1}\leqslant \boldsymbol{x}\leqslant \mathbf{2^{31}},\end{aligned}$$

$$\begin{aligned}\mathrm{flp2}(\boldsymbol{x}) &= \mathrm{clp2}(\boldsymbol{x} \overset{u}{\div} 2+\mathbf{1}), && \boldsymbol{x}\neq \mathbf{0},\\ &= \mathrm{clp2}(\boldsymbol{x}+\mathbf{1}) \overset{u}{\div} 2, && \boldsymbol{x}< \mathbf{2^{31}}.\end{aligned}$$

使用前导0计数指令（number of leading zeros instruction）很容易就能根据下列算式求出上调函数和下调函数的值。然而，如果要在 $\boldsymbol{x}=\mathbf{0}$ 及 $\boldsymbol{x}>\mathbf{2^{31}}$ 时使用这些关系式，那么就必须保证计算机的左移指令在移位量为−1，32及63时产生的结果是**0**。很多计算机（例如Power PC）都采用“模64”移位，这种移位方式符合上述要求。在移位量为−1的情况下，计算机朝相反方向移位也可以（也就是说，将左移−1位解读为右移1位也行）。

$$\begin{aligned}\mathrm{flp2}(\boldsymbol{x}) &= \mathbf{1}\ll(\mathbf{31}-\mathrm{nlz}(\boldsymbol{x}))\\ &= \mathbf{1}\ll(\mathrm{nlz}(\boldsymbol{x})\oplus\mathbf{31})\\ &= \mathbf{0x8000\ 0000}\overset{u}{\gg}\mathrm{nlz}(\boldsymbol{x})\\ \mathrm{clp2}(\boldsymbol{x}) &= \mathbf{1}\ll(\mathbf{32}-\mathrm{nlz}(\boldsymbol{x}-\mathbf{1}))\\ &= \mathbf{0x8000\ 0000}\overset{u}{\gg}(\mathrm{nlz}(\boldsymbol{x}-\mathbf{1})-\mathbf{1})\end{aligned}$$

3.2.1 向下舍入

图3.1演示了如何在没有前导0计数指令时实现无分支的向下舍入算法。该算法的核心思路就是把最左方的“1”向右传播，执行起来共需12条指令。

图3.2分别采用两个简单的循环来计算同一个函数，其中所有变量都是无符号整数。在右侧的那个循环中，如果 $\boldsymbol{x}$ 不等于**0**，我们就反复“关闭” $\boldsymbol{x}$ 最右侧的“1”；如果 $\boldsymbol{x}$ 等于**0**，我们就把执行本轮循环前的 $\boldsymbol{x}$ 作为函数值返回。

左侧循环需要 $4\mathrm{nlz}(\boldsymbol{x})+3$ 条指令实现，而如果忽略与0比较的指令，那么在 $\boldsymbol{x}\neq 0$ 时，右侧循环所需的指令数为 $4\mathrm{pop}(\boldsymbol{x})$[⊖]。

⊖ $\mathrm{pop}(\boldsymbol{x})$ 就是 $\boldsymbol{x}$ 中值为1的位元个数。

```
unsigned flp2(unsigned x) {
   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >> 16);
   return x - (x >> 1);
}
```

图 3.1　用无分支代码将 x 下调为不大于 x 且值与之最近的 2 的幂

```
y = 0x80000000;
while (y > x)
   y = y >> 1;
return y;
```

```
do {
   y = x;
   x = x & (x - 1);
} while(x != 0);
return y;
```

图 3.2　用简单的循环代码找出值不大于 x 且与之最接近的 2 的幂

3.2.2　向上舍入

利用刚才说的向右传播技巧，可以编写出一个巧妙的算法，把某个值向上调整为下一个 2 的幂。图 3.3 列出了此算法，代码不含分支，共需 12 条指令。

用刚才那种循环的办法实现向上舍入，效果并不好：

```
y = 1;

while (y < x)          //比较两个无符号数
   y = 2*y;
return y;
```

上述代码在 $x=0$ 时的结果为 1，这恐怕与大家的预期不符[㊀]。而且一旦 $x \geqslant 2^{31}$，就会陷入死循环。在正常情况下，它执行时要花费 $4n+3$ 条指令，其中 n 是返回整数 y 所对应的 2 的幂指数[㊁]。因此，从执行所需的指令数来判断，在 $n \geqslant 3$（也就是 $x \geqslant 8$）时，该算法比无分支算法慢。

```
unsigned clp2(unsigned x) {
   x = x - 1;
   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >> 16);
   return x + 1;
}
```

图 3.3　寻找值不小于 x 且与之最接近的 2 的幂

3.3　判断取值范围是否跨越了 2 的幂边界

假定内存被分隔成若干块（block），每一块的大小均为 2 的幂，且内存地址从 0 开始计数。这里说的“块”可以指字组、双字、页（page）等。现在给定起始地址 a 与长度 $l(l \geqslant 2)$，判断从 a 到 $a+l-1$ 之间的这一段内容是否跨越了两个块的边界。a 与 l 均为无符号数，其值不限，只要能容纳于寄存器中即可。

假如 $l=0$ 或 1，那么不论 a 为何值，都不会跨越块边界。而一旦 l 本身比块的大小还大，那么不论 a 为何值，都肯定会跨越块边界。假如 l 的值特别大（有可能大到 $a+l-1$ 的值已经突破了计算机容许的最大值，从而又折回到 0），那么就算内存范围的第一个字节与最后一个字节位于同一块内，也还是有可能跨越边界。

㊀　按照 2.3 节中 clp2 函数的定义，当 x 为 0 时，函数值也应为 0，而不应为 1。——译者注

㊁　例如 y 为 8 时，n 为 3，因为 $8=2^3$，也可以将 n 理解成 $\log_2 y$。——译者注

令人惊讶的是，在IBM System/370计算机[⊖]中，有一种非常精准的办法［CJS］能检测边界跨越。我们假设块大小为4096字节（也就是常见的内存页面大小），该方法代码如下：

```
O    RA,=A(-4096)
ALR  RA,RL
BO   CROSSES
```

第1条指令的意思是将RA（其中含有起始地址a）与数字$0x$FFFF FF00取逻辑或。第2条指令则是将RA与长度l相加，并设置条件码。执行完这条逻辑加（add logical）指令后，如果发生进位，那么特征码的第1个位元置1，如果32位寄存器RA的值非0，则特征码的第2个位元置1。若两个特征码位同时为1，则最后一条指令会令程序转入分支。转入分支时，RA的值就是该取值范围超过首个页面的那部分长度（这是一个未出现在需求中的附加功能）。

例如，若$a=0$且$l=4096$，那么会发生进位，然而结果寄存器中的值是0。于是两个特征码不可能都是1，所以程序也就不会转入CROSSES标签所指的分支了。

现在研究如何将上述算法移植到具备RISC指令集的计算机上，此类计算机通常没有那种“当发生进位且寄存器中运算结果非0时转入分支”的指令。为了描述起来方便，我们假定块的大小为8字节，这样的话，只有当发生进位（$((a\mid -8)+l\geqslant 2^{32})$且寄存器结果非0（$(a\mid -8)+l\neq 2^{32}$）时，［CJS］算法才会转入CROSSES分支。于是，上述算法也就等同于这个谓词的值：

$$(a\mid -8)+l>2^{32}$$

而这个谓词的值，又与在计算$((a\mid =8)-1)+l$的最后一个加法时的进位情况相同，所以在支持“进位时转入分支”（branch on carry）指令的计算机上，可以直接用此式来判断，这样的话，把载入常数−8的那条指令算上，一共需要5条。

如果计算机中没有“进位时转入分支”的指令，那么根据2.13.3节讲的“当且仅当$\neg x\overset{u}{<}y$时，$x+y$才会发生进位”这一事实，可以得到下式：

$$\neg((a\mid -8)-1)\overset{u}{<}l$$

使用$\neg(x-1)=-x$等式子，可以推出下述几种等效的“边界跨越”谓词表达式：

$$-(a\mid -8)\overset{u}{<}l$$

$$\neg(a\mid -8)+1\overset{u}{<}l$$

$$(\neg a\ \&7)+1\overset{u}{<}l$$

如果算上最后的分支指令，那么在大多数RISC架构的计算机中，上述算式需要5至6条指令。

⊖ IBM公司1970年推出的大型电脑系列，是IBM System/360的后继产品。详情参见：https://en.wikipedia.org/wiki/IBM_System/370。——译者注

此问题还有另外一种解决思路。因为某个取值范围是否跨越 8 字节边界的充分必要条件是：

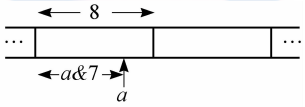
$$(a\ \&7)+l-1\geqslant 8$$

因为（当 l 很大时）可能会溢出，所以不能直接求此式的值。然而只要将其重排为 $\mathbf{8}-(\boldsymbol{a}\ \&\mathbf{7})<\boldsymbol{l}$，就能直接计算了（因为该式各部分均不会溢出）。于是就得到了这个表达式：

$$\mathbf{8}-(\boldsymbol{a}\ \&\mathbf{7})\overset{u}{<}\boldsymbol{l}$$

在绝大多数 RISC 架构的计算机中，这个式子都只需 5 条指令（如果有“立即数减法”指令，那就是 4 条）。若发生了跨越边界现象，那么可以用 $\boldsymbol{l}-(\mathbf{8}-(\boldsymbol{a}\ \&\mathbf{7}))$ 求得超出第 1 块的长度。只需多用一条减法指令即可算出此值。

下边这张图［Kumar］直观地演示了此公式。其中 $\boldsymbol{a}\ \&7$ 就是 $\boldsymbol{a}$ 在块中的偏移量，所以 $\mathbf{8}-(\boldsymbol{a}\ \&7)$ 就是当前块中剩余的空间。



3.4　习题

1. 将无符号整数向 8 的倍数舍入。分别实现下列 3 种舍入标准：㊀
   （a）三舍四入。㊁
   （b）四舍五入。㊂
   （c）三舍五入四成双㊃。在这种舍入方式下，一个除以 8 余 4 的数，应该舍入为距离其最近的 16 的倍数（这也叫“无偏差”舍入，“unbiased”rounding）。
2. 将无符号整数向 10 的倍数舍入。分别实现下列 3 种舍入标准。
   （a）四舍五入。
   （b）五舍六入。
   （c）四舍六入五成双。在这种舍入方式下，一个除以 10 余 5 的数，应该舍入为距离其

㊀ 原文为 halfway case，也就是当待舍入的数与左右两个 8 的倍数之间的距离均为 4 时，是应该舍还是应该入。不同标准下的舍入结果有可能不同。比如待舍入的数是 20，它左边那个 8 的倍数是 16，右边那个 8 的倍数是 24，20 与两者的距离均为 4，那么 20 究竟舍为 16 还是入为 24，则取决于所选的标准。——译者注

㊁ 若该数除以 8 的余数大于等于 4，则向上调整，否则向下调整。——译者注

㊂ 若该数除以 8 的余数小于等于 4，则向下调整，否则向上调整。——译者注

㊃ 此处借用十进制下的俗语。若该整数除以 8 的余数小于 4，则向下调整，大于 4 则向上调整。若恰好为 4，且商为偶数，则向下调整；若商为奇数，则向上调整，调整后的值必定是 8 的双数倍。比如 28 这个数，与 24 和 32 的距离都是 4，然而 28 与 8 的商为 3，是奇数，所以应该上调为 32，而不能下调为 24，因为 24 不是 8 的偶数倍。——译者注

最近的10的偶数倍。[⊖]

此题可任意使用除法、求余、乘法指令，而且不必考虑那些与最大的无符号整数非常接近的值。

3. 用C语言编写一个实现“不对齐加载”（unaligned load）功能的函数。该函数接收一个起始地址a，然后把从a到$a+3$的4个字节视为一个整数，载入32位寄存器中。起始地址参数a所指的内容，是待加载整数中权重最低的那个字节（也就是说，假定计算机是以“小端序”[⊜]的方式存放多字节整数的）。函数代码不要有分支语句，而且最多只能执行两次加载指令。如果a本身恰好位于字组边界处，则函数不应读取$a+4$这个地址中的内容，因为此字节可能处于一个“读保护块”（read-protected block）中。

⊖ 即按照“四舍六入五成双”标准舍入之后的数必定是20的倍数。其中“成双”一词的意思是，如果发现待舍入的值除以10的余数为5，那么一定要把它调整成10的双数倍（也就是20的倍数）。例如35与30和40的距离均为5，而35与10的商为3，是奇数，所以应该上调为40，而不能下调为30，因为30不是10的双数倍。详情参见：http://zh.wikipedia.org/wiki/数值简化规则。——译者注

⊜ little-endian，又称“小尾序”，它会把权重最低的字节存放在内存地址最小处。详情参见：http://zh.wikipedia.org/wiki/字节序。——译者注