CHAPTER 3

POWER-OF-2 BOUNDARIES

3–1 Rounding Up/Down to a Multiple of a Known Power of 2

Rounding an unsigned integer $\boldsymbol{x}$ down to, for example, the next smaller multiple of 8, is trivial: $\boldsymbol{x} \mathbin{\&} \boldsymbol{-8}$ does it. An alternative is $(\boldsymbol{x} \overset{u}{\gg} 3) \ll 3.$ These work for signed integers as well, provided "round down" means to round in the negative direction (e.g., $(\boldsymbol{-37}) \mathbin{\&} (\boldsymbol{-8}) = \boldsymbol{-40}$).

Rounding up is almost as easy. For example, an unsigned integer $\boldsymbol{x}$ can be rounded up to the next greater multiple of 8 with either of

$$(\boldsymbol{x} + \boldsymbol{7}) \mathbin{\&} \boldsymbol{-8}, \quad \text{or}$$
$$\boldsymbol{x} + (-\boldsymbol{x} \mathbin{\&} \boldsymbol{7}).$$

These expressions are correct for signed integers as well, provided "round up" means to round in the positive direction. The second term of the second expression is useful if you want to know how much you must add to $\boldsymbol{x}$ to make it a multiple of 8 [Gold].

To round a signed integer to the nearest multiple of 8 toward 0, you can combine the two expressions above in an obvious way:

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

An alternative for the first line is $\boldsymbol{t} \leftarrow (\boldsymbol{x} \overset{s}{\gg} 2) \overset{u}{\gg} 29,$ which is useful if the machine lacks *and immediate*, or if the constant is too large for its immediate field.

Sometimes the rounding factor is given as the $\log_2$ of the alignment amount (e.g., a value of 3 means to round to a multiple of 8). In this case, code such as the following may be used, where $\boldsymbol{k} = \log_2(\text{alignment amount})$:

$$\begin{array}{rl}
\text{round down:} & \boldsymbol{x} \mathbin{\&} ((\boldsymbol{-1}) \ll \boldsymbol{k}) \\
 & (\boldsymbol{x} \overset{u}{\gg} \boldsymbol{k}) \ll \boldsymbol{k} \\
\text{round up:} & \boldsymbol{t} \leftarrow (\boldsymbol{1} \ll \boldsymbol{k}) - \boldsymbol{1};\quad (\boldsymbol{x} + \boldsymbol{t}) \mathbin{\&} \neg \boldsymbol{t} \\
 & \boldsymbol{t} \leftarrow (\boldsymbol{-1}) \ll \boldsymbol{k};\quad (\boldsymbol{x} - \boldsymbol{t} - \boldsymbol{1}) \mathbin{\&} \boldsymbol{t}
\end{array}$$

3–2 Rounding Up/Down to the Next Power of 2

We define two functions that are similar to floor and ceiling, but which are directed roundings to the closest integral power of 2, rather than to the closest integer. Mathematically, they are defined by

$$\operatorname{flp2}(x) = \begin{cases} \text{undefined}, & x < 0, \\ 0, & x = 0, \\ 2^{\lfloor \log_2 x \rfloor}, & \text{otherwise;} \end{cases} \qquad \operatorname{clp2}(x) = \begin{cases} \text{undefined}, & x < 0, \\ 0, & x = 0, \\ 2^{\lceil \log_2 x \rceil}, & \text{otherwise.} \end{cases}$$

The initial letters of the function names are intended to suggest “floor” and “ceiling.” Thus, $\operatorname{flp2}(x)$ is the greatest power of 2 that is $\le x$, and $\operatorname{clp2}(x)$ is the least power of 2 that is $\ge x$. These definitions make sense even when x is not an integer (e.g., $\operatorname{flp2}(0.1) = 0.0625$). The functions satisfy several relations analogous to those involving floor and ceiling, such as those shown below, where n is an integer.

$$\begin{array}{ll} \lfloor x \rfloor = \lceil x \rceil \text{ iff } x \text{ is an integer} & \operatorname{flp2}(x) = \operatorname{clp2}(x) \text{ iff } x \text{ is a power of 2 or is 0} \\ \lfloor x + n \rfloor = \lfloor x \rfloor + n & \operatorname{flp2}(2^n x) = 2^n \operatorname{flp2}(x) \\ \lceil x \rceil = -\lfloor -x \rfloor & \operatorname{clp2}(x) = 1/\operatorname{flp2}(1/x),\ x \ne 0 \end{array}$$

Computationally, we deal only with the case in which x is an integer, and we take it to be unsigned, so the functions are well defined for all $\boldsymbol{x}$. We require the value computed to be the arithmetically correct value modulo 2^{32} (that is, we take $\operatorname{clp2}(\boldsymbol{x})$ to be $\mathbf{0}$ for $\boldsymbol{x} > \mathbf{2^{31}}$). The functions are tabulated below for a few values of $\boldsymbol{x}$.

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

Functions flp2 and clp2 are connected by the relations shown below. These can be used to compute one from the other, subject to the indicated restrictions.

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

The round-up and round-down functions can be computed quite easily with the *number of leading zeros* instruction, as shown below. However, for these relations to hold for $\boldsymbol{x} = \mathbf{0}$ and $\boldsymbol{x} > \mathbf{2^{31}}$, the computer must have its shift instructions defined to produce $\mathbf{0}$ for shift amounts of –1, 32, and 63. Many machines (e.g., PowerPC) have “mod 64” shifts, which do this. In the case of –1, it is adequate if the machine shifts in the opposite direction (that is, a shift left of –1 becomes a shift right of 1).

$$\begin{aligned}
\text{flp2}(\boldsymbol{x}) &= \mathbf{1} \ll (\mathbf{31} - \text{nlz}(\boldsymbol{x})) \\
&= \mathbf{1} \ll (\text{nlz}(\boldsymbol{x}) \oplus \mathbf{31}) \\
&= \mathbf{0x80000000} \overset{u}{\gg} \text{nlz}(\boldsymbol{x}) \\
\text{clp2}(\boldsymbol{x}) &= \mathbf{1} \ll (\mathbf{32} - \text{nlz}(\boldsymbol{x}-\mathbf{1})) \\
&= \mathbf{0x80000000} \overset{u}{\gg} (\text{nlz}(\boldsymbol{x}-\mathbf{1}) - \mathbf{1})
\end{aligned}$$

Rounding Down

Figure 3–1 illustrates a branch-free algorithm that might be useful if *number of leading zeros* is not available. This algorithm is based on right-propagating the leftmost 1-bit, and executes in 12 instructions.

Figure 3–2 shows two simple loops that compute the same function. All variables are unsigned integers. The loop on the right keeps turning off the rightmost 1-bit of $\boldsymbol{x}$ until $\boldsymbol{x} = \mathbf{0}$, and then returns the previous value of $\boldsymbol{x}$.

```
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);
}
```

FIGURE 3–1. Greatest power of 2 less than or equal to $\boldsymbol{x}$, branch-free.

```
   y = 0x80000000;            do {
   while (y > x)                 y = x;
      y = y >> 1;                x = x & (x - 1);
   return                     } while(x != 0);
                              return y;
```

FIGURE 3–2. Greatest power of 2 less than or equal to $\boldsymbol{x}$, simple loops.

The loop on the left executes in $4\,\mathrm{nlz}(\boldsymbol{x}) + 3$ instructions. The loop on the right, for $x \neq 0$, executes in $4\,\mathrm{pop}(\boldsymbol{x})$ instructions,[1] if the comparison to 0 is zero-cost.

Rounding Up

The right-propagation trick yields a good algorithm for rounding up to the next power of 2. This algorithm, shown in Figure 3–3, is branch-free and runs in 12 instructions.

An attempt to compute this with the obvious loop does not work out very well:

```
   y = 1;

   while (y < x)      // Unsigned comparison.
      y = 2*y;
   return y;
```

This code returns 1 for $x = 0$, which is probably not what you want, loops forever for $x \geq 2^{31}$, and executes in $4n + 3$ instructions, where n is the power of 2 of the returned integer. Thus, it is slower than the branch-free code, in terms of instructions executed, for $n \geq 3$ $(x \geq 8)$.

```
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;
}
```

FIGURE 3–3. Least power of 2 greater than or equal to $\boldsymbol{x}$.

1. pop($\boldsymbol{x}$) is the number of 1-bits in $\boldsymbol{x}$.

3–3 Detecting a Power-of-2 Boundary Crossing

Assume memory is divided into blocks that are a power of 2 in size, starting at address 0. The blocks may be words, doublewords, pages, and so on. Then, given a starting address a and a length l, we wish to determine whether or not the address range from a to $a + l - 1,\ l \ge 2,$ crosses a block boundary. The quantities a and l are unsigned and any values that fit in a register are possible.

If $l = 0$ or 1, a boundary crossing does not occur, regardless of a. If l exceeds the block size, a boundary crossing does occur, regardless of a. For very large values of l (wraparound is possible), a boundary crossing can occur even if the first and last bytes of the address range are in the same block.

There is a surprisingly concise way to detect boundary crossings on the IBM System/370 [CJS]. This method is illustrated below for a block size of 4096 bytes (a common page size).

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

The first instruction forms the logical *or* of RA (which contains the starting address a) and the number 0xFFFFF000. The second instruction adds in the length, and sets the machine's 2-bit condition code. For the *add logical* instruction, the first bit of the condition code is set to 1 if a carry occurred, and the second bit is set to 1 if the 32-bit register result is nonzero. The last instruction branches if both bits are set. At the branch target, RA will contain the length that extends beyond the first page (this is an extra feature that was not asked for).

If, for example, $a = 0$ and $l = 4096,$ a carry occurs but the register result is 0, so the program properly does *not* branch to label CROSSES.

Let us see how this method can be adapted to RISC machines, which generally do not have *branch on carry and register result nonzero*. Using a block size of 8 for notational simplicity, the method of [CJS] branches to CROSSES if a carry occurred ($(a \mid -8) + l \ge 2^{32}$) and the register result is nonzero ($(a \mid -8) + l \ne 2^{32}$). Thus, it is equivalent to the predicate

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

This in turn is equivalent to getting a carry in the final addition in evaluating $((a \mid -8) - 1) + l.$ If the machine has *branch on carry*, this can be used directly, giving a solution in about five instructions counting a load of the constant −8.

If the machine does not have *branch on carry*, we can use the fact that carry occurs in $x + y$ iff $\neg \boldsymbol{x} \overset{u}{<} \boldsymbol{y}$ (see "Unsigned Add/Subtract" on page 29) to obtain the expression

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

Using various identities such as $\neg(\boldsymbol{x} - \mathbf{1}) = -\boldsymbol{x}$ gives the following equivalent expressions for the “boundary crossed” predicate:

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

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

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

These can be evaluated in five or six instructions on most RISC computers.

Using another tack, clearly an 8-byte boundary is crossed iff

$$(a \,\&\, 7) + l - 1 \geq 8.$$

This cannot be directly evaluated because of the possibility of overflow (which occurs if l is very large), but it is easily rearranged to $8 - (a \,\&\, 7) < l,$ which can be directly evaluated on the computer (no part of it overflows). This gives the expression

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

which can be evaluated in five instructions on most RISCs (four if it has *subtract from immediate*). If a boundary crossing occurs, the length that extends beyond the first block is given by $\boldsymbol{l} - (\mathbf{8} - (\boldsymbol{a} \,\&\, \mathbf{7})),$ which can be calculated with one additional instruction (*subtract*).