Data Lab 实验报告

By Zeng GuanYang

`bitXor`

根据异或，我们有

$$x \oplus y = (x \wedge \neg y) \vee (\neg x \wedge y)$$

而

$$x \vee y = \neg(\neg(x \vee y)) = \neg(\neg x \wedge \neg y)$$

因此，

$$x \oplus y = \neg(\neg(x \wedge \neg y) \wedge \neg(\neg x \wedge y))$$

对应进行按位与和按位取反即可。

代码如下：

```
int bitXor(int x, int y)
{
  int p1 = x & ~y;
  int p2 = ~x & y;     // x ^ y = p1 | p2
  return ~(~p1 & ~p2); // actual p1 | p2
}
```

总操作数：8

---

tmin

返回最小二进制补码，也就是0x80000000。
依据补码表示法，知其为$1 << 31$。

代码如下：

```
int tmin(void)
{
  return 1 << 31; // 0x80000000
}
```

总操作数：1

---

isTmax

$\mathrm{Tmax} =$ 0x7fffffff。
依据有符号数32位整形的溢出规则，有$\mathrm{Tmin} = \mathrm{Tmax} + 1$，此外，Tmin还有一个特点即$-\mathrm{Tmin} = \mathrm{Tmin}$。不过除Tmin外还有这个特点的数为0，需要特判。

依据上述结论，可以判断数 x 是否为Tmax。

- 判断 x 是否为 -1
- 判断 $x + 1$ 取相反数（等价于按位取反加 1）是否和自身相等
- $x = y \Leftrightarrow x \oplus y = 0$

代码如下：

```
int isTmax(int x)
{
  int pls = x + 1;
  int nonzero = !!pls;        // if 1, x != -1, 0 for x=-1
  int neg = ~pls + 1;         // neg = -pls = -(x+1)
  int negeqal = !(neg ^ pls); // (x+1)==-(x+1)
  return nonzero & negeqal;   // nonzero && -pls==pls
}
```

总操作数：8

---

allOldBits

需要利用x & 0xAAAAAAAA 提取所有奇数位的 1, 再判断结果和 0xAAAAAAAA 是否相等。相等则说明奇数位全为 1。
现在需要构造0xAAAAAAAA。可以通过0xAA不断右移再加上原来的数构造。

代码如下：

```
int allOddBits(int x)
{
  int mask = 0xAA;
  int oddBits;
  mask = (mask << 8) | mask;  // 0xAAAA
  mask = (mask << 16) | mask; // 0xAAAAAAAA
  oddBits = x & mask;         // get all odd bits of x
  return !(oddBits ^ mask);   // check whether odd bits are all 1.
}
```

总操作数：7

---

negate

依据补码规则，$-x = \sim x + 1$。

代码如下：

```
int negate(int x)
{
  return ~x + 1;
}
```

总操作数：2

isAsciiDigit

要验证 0x30 $\leq x \leq$ 0x39，
可以先验证 x 的高四位是否为 3，再验证低四位是否在0x0 $\sim$ 0x9之间。
高四位$x_h = x >> 4$, 可以通过与 0x3 的异或验证。
对低四位 $x_l = x$ & 0xF 的验证等价于 x_l 符号位为 0（这个总满足）, 且 0x9 $- x_l =$ 0x9+ $\sim x_l + 1$ 的符号位为 0。
对于32位有符号整数 y, 可以通过 $y >> 31$ & 0x1 得知 y 的符号。

整合上述过程，得到代码如下：

```
int isAsciiDigit(int x)
{
  int x_l = x & 0xF;         // Lower 4 bits of x
  int x_h = x >> 4;          // Higher bits of x
  int hok = !(x_h ^ 0x3);    // Satisfy 0x3?
  int exp = 0x9 + ~x_l + 1; // 0x9 - x_l
  int lok = !(exp >> 31);    // 0x9 - x_l >= 0
  return hok & lok;
}
```

总操作数：10

conditional

对于 x 进行真/假（$!!x$）判断可以得到 $1/0$ 值。
将此值填满32位整数（左移31位再算数右移31位），得到掩码。
结果的每一位是根据掩码在 y 和 z 中二选一的结果，通过掩码和掩码的反码分别与 y 和 z 进行按位与操作可以得到不会干涉的结果，将这两个结果相加即可得到最后的返回值。

代码如下：

```
int conditional(int x, int y, int z)
{
  int val = !!x; // get condition result
  // fill the mask with condition value
  int mask = (val << 31) >> 31;
  return (mask & y) + (~mask & z);
}
```

总操作数：8

---

isLessOrEqual

数学上 $x \leq y \Leftrightarrow y - x \geq 0$，然而计算机需要考虑溢出的情况。
不妨令$r = y - x = y + \sim x + 1$。下面讨论 x 和 y 的符号。

- 如果 $y \geq 0$， $x < 0$，那结果为返回1，尽管此时r可能会溢出。
- 如果 $y \geq 0$， $x \geq 0$，此时不会溢出，需要看 r 的符号，$r \geq 0$则返回 1，$r < 0$则返回 0。
- 如果 $y < 0$， $x \geq 0$，那结果为返回0，尽管此时r可能会溢出。
- 如果 $y < 0$， $x < 0$，此时不会溢出，需要看 r 的符号。

综合上述条件，得到代码如下：

```
 */
int isLessOrEqual(int x, int y)
{
  int sx = (x >> 31) & 0x1; // Get the sign of x
  int sy = (y >> 31) & 0x1; // Get the sign of y
  int r = y + ~x + 1;
  int sr = (r >> 31) & 0x1;      // Get the sign of result
  int validr = !(sx ^ sy) & !sr; // when signs are the same, check r>=0
  return (sx & !sy) | validr;    // first indicates x<0 && y>=0
}
```

总操作数：16

logicalNeg

仅当 $x = 0$ 时返回 1。
由于 $-x = \sim x + 1$, 若 x 不为 0，设 x 最高的 1 在第 k 位， 那么，如果 $k = 31$，那么 x 最高位为 1, $x|-x$ 最高位为 1；若 $k < 31$，那么，$\sim x$的最高位为 1，且第 k 位为 0。可知 $\sim x + 1$ 最高位的 1，因为进位在第 k 位之后停止。因此，若 x 不为 0，$x|-x$最高位总为 1 。当 $x = 0$ 时，$x|-x = 0$。

据此可得到代码如下：

```
int logicalNeg(int x)
{
  int neg = ~x + 1;         // neg = -x
  int y = x | neg;          // y = x|-x
  return ~(y >> 31) & 0x1; // ~sign(y) & 0x1, negate the sign.
}
```

总操作数：6

---

howManyBits

- 对于正数来说，在补码表示下，最少所需要的位数为 x 最高位 1 所在的位数 $+1$（用于表示符号）。
- 对于负数来说，在补码表示下，最少所需要的位数为 x 最高位 0 所在的位数 $+1$，等价于求其反码最高位 1 所在位数 $+1$，这样可以统一以便后续计算。

不妨设 s 为 x 的符号位构成的32位掩码，后续计算可利用统一的 $v = (s \ \& \ (\sim x))|((\sim s) \ \& \ x)$
随后需要求最高位的 1 所在的位置。
可以通过不断将 v 右移的方式求最高位的 1。
首先判断 v 的高16位有无 1，若有，将位数计算记上16，再将 v 右移16位判断高16位中 1 的位置；若无，说明最高位的 1 在低16位中。我们可以利用相似的方式继续判断这16位中最高位的 1 所在的位置，这个时候需要右移 8 位进行判断（实际上是二分法）。直到最后要右移的位数为0停止。

代码如下：

```
int howManyBits(int x)
{
  int s = x >> 31; // sign maskx
  int v = (s & ~x) | (~s & x);
  int _16bit, _8bit, _4bit, _2bit, _1bit;
  // 32 bit split
  _16bit = !!(v >> 16) << 4; // high 16bits. check if there's 1. turn the result to count.
  v = v >> _16bit;           // _16bit = 0 => highest 1 in lower 16bit;
                             // _16bit = 16 => highest 1 in higher 16bit, needs shift.
  // 16 bit split
  _8bit = !!(v >> 8) << 3; // check high 8 bits. store the count.
  v = v >> _8bit;          // similar operation
  // 8 bit split
  _4bit = !!(v >> 4) << 2;
  v = v >> _4bit;
  // 4 bit split
  _2bit = !!(v >> 2) << 1;
  v = v >> _2bit;
  // 2 bit split
  _1bit = !!(v >> 1);
  v = v >> _1bit; // needs to know v is in high bit or low bit.
  // adds up. v it self represent the 1bit result.
  return _16bit + _8bit + _4bit + _2bit + _1bit + v + 1;
}
```

总操作数：36

---

floatScale2

分类讨论：

- 对于非规格化数，$Exp = 0$，此时对尾数部分右移即可。如果尾数溢出，多余的 1 会自然溢出到阶码位上变成规格化数。因此在这里只需要取出符号位和位数做对应处理即可。
- 对于规格化数，直接将阶码加 1 即可。如果阶码加 1 后全 1，说明应当表示为无穷，此时尾数当清零。

- 对于$Exp = 255$的情况，数为NaN或无穷，直接返回原数即可。

代码如下：

```
unsigned floatScale2(unsigned uf)
{
  unsigned sgn = uf & 0x80000000u;   // sign part
  unsigned exp = (uf >> 23) & 0xFFu; // exp part
  unsigned frac = uf & 0x7fffffu;    // frac part
  if (exp == 0)                      // denorm
  {
    frac = frac << 1;
    return sgn | frac;
  }
  if (exp == 0xffu) // NaN, infty
  {
    return uf;
  }
  exp = exp + 1;   // norm
  if (exp == 0xffu) // overflow
  {
    return sgn | (exp << 23); // no need for frac part.
  }
  return sgn | (exp << 23) | frac;
}
```

总操作数：15

---

floatFloat2Int

首先利用位运算得到符号位，阶码和尾数。

- 对于非规格化数，显然由于过小只能舍入为 0。
- 对于正无穷和NaN，由于超出范围因此返回 $Tmin$。
- 对于规格化数，$Bias = 2^7 - 1 = 127, E = Exp - Bias$。
  - 由于整数范围是$-2^{31} \sim 2^{31} - 1$因此，对于 $E \geq 31$，此时超出范围，应当返回 $Tmin$（刚好 $Tmin$ 自身也在其中）；而对于$E < 0$，可知浮点数的绝对值小于1，

此时舍入为 0。
- 对于 $0 \leq E \leq 30$ 的情况，可以基于尾数进行处理。
  已知尾数有23位，那么可以先假设其能够在整数中完整表达（记为 $v = (1 << 23) | frac$，加上隐藏位的 1 ）。通过左右移来控制实际的位数实现截断。
  如果 $E \geq 23$，那么 v 应当还要左移 $E - 23$ 位以匹配数值，
  如果 $E < 23$，那么 v 应当还要右移 $23 - E$ 位以匹配数值。

最后依据浮点数符号判断给出最后的正负结果。

综合上述判断得到代码如下：

```c
int floatFloat2Int(unsigned uf)
{
  unsigned sgn = uf & 0x80000000u;
  unsigned exp = (uf >> 23) & 0xFFu;
  unsigned frac = uf & 0x7fffffu;
  unsigned v;
  int E;
  if (exp == 255)
  {
    return 0x80000000u;
  }
  E = exp - 127; // actually, the same logic for norm & denorm
  if (E < 0)
  {
    return 0;
  }
  else if (E > 30) // out of range for norm
  {
    return 0x80000000u;
  }
  v = (1u << 23) | frac; // 1 for hidden bit
  if (E >= 23)
  {
    v = v << (E - 23);
  }
  else // 0<=E<23
  {
    v = v >> (23 - E);
  }
  if (sgn) // sign check
  {
    return -v;
  }
  else
  {
    return v;
  }
}
```

总操作数：16

`floatPower2`

由于 $Bias = 127$，目标计算 $Bias + x$ 的结果并构造浮点数返回。
分类讨论：

- $Exp = Bias + x \geq 255$ 超出边界，返回正无穷，此时 $x \geq 128$。
- $1 \leq Exp = Bias + x < 255$ 位于规格化范围内，因此可以构造对应的浮点满足 $Exp = Bias + x, frac = 0$返回，此时 $-126 \leq x < 128$。
- x 过小但位于非规格化数范围，此时 $E = 1 - Bias = -126$，利用尾数表示 $x - E$ 的部分。由于隐含位为 0，尾数最多23位，因此，最多可满足 $-23 \leq x - E \leq -1$，此时 $-149 \leq x \leq -127$。为了表示对应的尾数部分，需要 $1 << 23$ 右移 $|(x - E)| = E - x$位，等价于 $1 << (23 - (E - x)) = 1 << (x + 149)$
- $x < -149$ 过小而无法表示，返回 0。

代码如下：

```
unsigned floatPower2(int x)
{
  if (x >= 128) // too large
  {
    return 0x7f800000; // +INF
  }
  else if (x >= -126) // norm , frac = 0
  {
    unsigned exp = 127 + x;
    return exp << 23;
  }
  else if (x >= -149) // denorm, use frac to represent
  {
    return 1u << (x + 149);
  }
  else // too small to represent, return 0
  {
    return 0;
  }
}
```

总操作数：9

感想

时隔1年再一次做了CSAPP的Lab，~~哈，感觉都忘光了~~
重新回顾了整数和浮点数的知识还挺有用的XD。