计算机体系结构——位运算实验 说明文档

软件41 马子俊 2014013408

整数运算部分

1. int bitAnd(int x, int y)   // x&y using only ~ and

思路：

利用 `DeMorgan's Law` 实现 `~` 和 `|` 代替 `&` 即可， 即：`x&y = ~((~x) | (~y))`

---

2. int getByte(int x, int n)   // Extract byte n from word x

思路：

将整数 `x` 右移适当的位数(记为`shift_bits`)，使得待提取的字节移动至整数的末尾字节，利用掩码 `0xFF` 和 `&` 运算提取字节。

如何确定`shift_bits` ？

考虑

> n = 0   => shift_bits = 0
>
> n = 1   => shift_bits = 8
>
> n = 2   => shift_bits = 16
>
> n = 3   => shift_bits = 24

则有

> shift_bits = n * 8 = n << 3

问题得解

---

3. int logicalShift(int x, int n)   // shift x to the right by n, using a logical shift

思路：

`>>` 为`算数右移`, 右移后，高位上填充的符号位需置为 `0` ，即可实现`逻辑右移`。

利用掩码 `mask_number` 和 `&` 运算可以实现高位置 `0`。

所设计的 `mask_number`应当满足 —— 高 `n` 位为 `0` , 低 `32 - n` 位为 `1` (`0 <= n <= 31`)。

如何生成 `mask_number` ？

设

> int mask_number = ((1 << 31) >> n) << 1;

此时`mask_number` 恰为 —— 高 `n` 位为 `0` , 低 `32 - n` 位为 `1` (`0 <= n <= 31`)

> mask_number = ~mask_number;

此时`mask_number`符合要求。

问题得解

---

4. int bitCount(int x)   // returns count of number of 1's in word

思路:

本题统计`1`的个数的方法， 参考了`维基百科`之`海明权重`,

- Reference : https://en.wikipedia.org/wiki/Hamming_weight

这里以`8bits`常量`(10111010)`为例, 演示此方法的奇妙之处:

expression	binary	decimal	comment
a	10 11 10 10		the original number
b0 = (a >> 0) & 01 01 01 01	00 01 00 00	0,1,0,0	every other bit from a
b1 = (a >> 1) & 01 01 01 01	01 01 01 01	1,1,1,1	the remaining bits from a
c = b0 + b1	01 10 01 01	1,2,1,1	list giving the number of 1s in each 2-bit slice of a
d0 = (c >> 0) & 0011 0011	0010 0001	2, 1	every other count from c
d1 = (c >> 2) & 0011 0011	0001 0001	1, 1	the remaining counts from c
e = d0 + d1	0011 0010	3, 2	list giving the number of 1s in each 4-bit slice of a
f0 = (e >> 0) & 00001111	00000010	2	every other count from e
f1 = (e >> 4) & 00001111	00000011	3	the remaining counts from e
g = f0 + f1	00000101	5	the final answer for a

而对于`32bits`常量，除了应用上述方法外，我们还需要如下掩码:

```
1. mask_1 = 0x55555555
2. mask_2 = 0x33333333
3. mask_3 = 0x0F0F0F0F
4. mask_4 = 0x00FF00FF
5. mask_5 = 0x0000FFFF
```

这些掩码可利用`0xFF`通过`+`,`<<`,`^`产生，这更多是一种技巧而非逻辑，便不在此处继续叙述掩码产生过程了。

---

5. int bang(int x)　　// Compute !x without using !

思路:

`!`运算将`0`变为`1`, 将非`0`数变为`0`。问题的关键是如何用位运算符将`0`与非`0`数区别开来。

考虑`补码`运算(two's complement, 可视作取相反数运算)

Integer	Sign Bit	Integer after two's complement	Sign bit
0	0	0	0
-INT_MAX	1	-INT_MAX	1
other	0/1	~x + 1	1/0

将互为补码的两数进行`|`操作，则只有当原数为`0`时，结果的符号位才为`0`。

我们对符号位取反， 再通过移位操作来获得符号位，即可得到与`!`运算相同的结果。

问题得解

---

6. int tmin(void)　　// return minimum two's complement integer

思路:

`-INT_MAX = 0x80000000`, 可通过`(1 << 31)`实现。

---

7. int fitsBits(int x, int n)

// return 1 if x can be represented as an n-bit, two's complement integer.

思路：

```
1.
    当 x 是负数，它的符号位是 1 。如果说 32bits 的 x 可以用 nbits 进行表示 (1 <= n <= 32),

    那么 x 的高 (32 - n + 1) 位就不能出现 0 。否则 32bits 所表示的值和 nbits 表示的值不同。

2.
    当 x 是正数，它的符号位是 0 。如果说 32bits 的 x 可以用 nbits 进行表示 (1 <= n <= 32),

    那么 x 的高 (32 - n + 1) 位就不能出现 1。否则 32bits 所表示的值和 nbits 表示的值不同。

3.
    综合以上两点，如果说 32bits 的 x 可以用 nbits 进行表示 (1 <= n <= 32) ,

    那么 x 的高 (32 - n + 1) 位应当全为符号位。
```

所以，我们可以把`nbits`中的最高位复制到更高的`(32 - n)`位(我们把该结果记为`y`)，之后与`x`进行比较(用`^`运算)，如果 `x== y` ,那么`x`的高`(32 - n + 1)`位被证明为符号位， `x`可用`nbits`进行表示；如果`x != y`，则`x`的高`(32 - n + 1)`位则不全是符号位， `x`不可用`nbits`进行表示。

问题得解

---

8. int divpwr2(int x, int n)   // Compute x/(2^n), Round toward zero

思路:

```
1.   x >> n 为 x / (2 ^ n) 精确解向下取整的结果；
2.  当 x >= 0,  x >> n 相当于向 0 取整，即 x >> n 恰为答案；
3.  当 x < 0,  x >> n 相当于向 负无穷 取整，为得到正确结果，可计算（x + bias ) >> n， 这里bias可取2^n - 1
```

此处bias可通过如下位运算获得:

> int bias = ~((~0) << n)

问题得解

---

9. int negate(int x)   // return -x

思路:

前面提到过，补码运算本身就是取反运算，直接返回 `~x + 1` 即可。

---

10. int isPositive(int x)   // return 1 if x > 0, return 0 otherwise

思路:

1. 正整数满足符号位为`0`, 可用 !(x >> 31)加以判断
2. 正整数不包括`0`，可用 !!x 加以判断

故 `return !(x >> 31) & !!x` 即可

---

11. int isLessOrEqual(int x, int y)   // if x <= y then return 1, else return 0

思路:

```
1. 直接通过判断 x- y 的正负来得知 x ， y 的大小关系，是不可取的，因为存在  x - y 运算结果溢出的情况 ；
2. 当 x, y 符号位相同时，运算结果是不会溢出的，此时通过考察 x - y 的符号位来判断大小关系是合理的;
3. 当 x, y 符号位不同时，我们可以通过判断 x 和 y 的符号位来直接得出x, y的大小关系，避免触及 x - y 可能出现的结果溢出情况。
```

---

12. ilog2(int x)   // return floor(log base 2 of x), where x > 0

思路： 设

$$\lfloor \log_2 x \rfloor = ans$$

则

$$2^{ans} \leq x 且 2^{ans+1} > x$$

再设 x 的二进制表示中最高位的1所具有的的权重是2^n

那么

$$2^n \leq x$$

且

$$x \leq \sum_{i=0}^{n} 2^i = 2^{n+1} - 1 < 2^{n+1}$$

综上，n即为所求结果

接下来的问题在于，如何知道n的值？

将比最高位 `1` 权重要低的bit，全部置为 `1` , 之后利用前面分析的 `bitCount` 函数，统计 `1` 的个数即可。

我们通过如下操作把低位全部置 `1` :

> int most_2_one = x | (x >> 1); // 紧邻最高位“1”的1个低位被置为 1
>
> int most_4_one = most_2_one | (most_2_one >> 2); // 紧邻最高位“1”的3个低位被置为 1
>
> int most_8_one = most_4_one | (most_4_one >> 4); // 紧邻最高位“1”的7个低位被置为 1
>
> int most_16_one = most_8_one | (most_8_one >> 8); // 紧邻最高位“1”的15个低位被置为 1
>
> int most_32_one = most_16_one | (most_16_one >> 16); // 紧邻最高位“1”的31个低位被置为 1

对于正整数来说，比最高位 `1` 要低的bit数只可能 `<=30` , 所以经过以上 `5` 步操作，实际上可以保证 比最高位 `1` 要低的bit全部被置为 `1` 。

之后调用 `bitCount` 函数即可，问题得解。

单精度浮点数运算部分

说明：

1. 利用掩码 `0x7F800000` 和 `&` 运算可获取 `uf` 的指数(`exp`)部分,
2. 利用掩码 `0x007FFFFF` 和 `&` 运算可获取 `uf` 的小数(`frac`)部分,
3. 利用掩码 `0x80000000` 和 `&` 运算可获取 `uf` 的符号位(`sign`)部分,
4. 以下使用 `sign` , `exp` , `frac` 简化问题说明。

13. unsigned float_neg(unsigned uf)

//Return bit-level equivalent of expression -f for floating point argument f.

思路:

1. 依据 `exp` 和 `frac` 判断 `uf` 是否为 `NaN`，如果是，直接返回 `uf`

2. 如果不是 `NaN`,则更改 `uf` 的符号位，这可以通过掩码 `0x80000000` 和 `^` 运算实现

问题得解。

---

14. unsigned float_i2f(int x)   // Return bit-level equivalent of expression (float) x

思路：

1. 在所有的整数中，只有 `0` 是 `denormalized value`，且 `0` 的整数和浮点数表示一致。故如果 `x == 0`，返回 `0` 即可；

2. 对于非 `0` 整数，我们可以判断正负，从而知道 `sign` 的值

3. 我们假设一个非零整数 `x` 的二进制表示如下：

'0' bits	first '1' bit	remaining bits
00...00	1	XXX..XXX

注意到此时我们是从normalized value的角度去构造x的浮点数表示，可以想见：

$$exp = len(remaining\ bits) + bias = len(remaining\ bits) + 127$$

$$frac = remaining\ bits$$

但注意到 `frac` 的长度如果大于 `23bits`，则要舍弃末尾的bits, 近似到 `23bits`。近似的时候基本遵循 `四舍五入` 的原则，但是 `half-way` 的时候要考虑到让近似后最末位为 `0`。

最后将求得的 `sign`，`exp` 和 `remain` 拼接在一起即可，问题得解

---

15. unsigned float_twice(unsigned uf)

// Return bit-level equivalent of expression 2 * f for floating point argument f.

思路:

1. 如果 `uf` 是 `NaN` 或者 `Infinity`，直接返回 `uf`；
2. 如果 `uf` 是 `Normalized Value`，则直接将 `exp + 1`，这可以通过 `+ 0x00800000` 实现
3. 如果 `uf` 是 `Denormalized Value`，显然不可以通过 `exp + 1` 实现，因为IEEE标准对 `Normalized Value` 和 `Denormalized Value` 的解读方式是不同的。

那么如何实现 `Denormalized Value * 2` 呢?

我们接下来考察这样一个操作 `(uf << 1) (uf is denormalized value)`，我们把 `denormalized value` 分为以下两类:

Type 1:

sign bit	8 exp bits	23 frac bits
0/1	0000000	0XXXXX...XXX

Type 2:

sign bit	8 exp bits	23 frac bits
0/1	0000000	1XXXXX...XXX

对于Type 1, 设移位操作前浮点数的值是

$$V_1 = (-1)^s * M_1 * 2^{-126}$$

左移1位后，浮点数变为

Type 1' :

sign bit	8 exp bits	23 frac bits
0	0000000	XXXXX...XXX0

设移位操作后浮点数的值是

$$V_1^{'} = (-1)^0 * M_1^{'} * 2^{-126}$$

则

$$\frac{V_1^{'}}{V_1} = (-1)^s * \frac{M_1^{'}}{M_1} = (-1)^s * 2$$

对于Type2, 设移位操作前浮点数的值是

$$V_2 = (-1)^s * M_2 * 2^{-126}$$

左移1位后，浮点数变为

Type 2' :

sign bit	8 exp bits	23 frac bits
0	0000001	XXXXX...XXX0

注意到此时移位操作后的浮点数为Normalized Value, 所以解读方式应当发生改变，其值应为

$$V_2^{'} = (-1)^s * M_2^{'} * 2^{1-127}$$

则

$$\frac{V_2^{'}}{V_2} = (-1)^s * \frac{M_2^{'}}{M_2} = (-1)^s * 2$$

注：虽然 M_2 和 $M_2^{'}$ 对应的浮点数解读方式不一样，但是 $\frac{M_2^{'}}{M_2} = 2$ 的结论仍成立。

可见在不考虑符号位的情况下，当 `uf` 是 `denormalized value` 时， `(uf << 1)` 是可以被视为 `uf * 2` 的。由于 `(uf << 1)` 的结果符号位总为0，可以将其再置为`uf`的符号位。实现如下：

> int result = (uf << 1) | sign

综上，问题得解