Data Lab: Manipulating Bits Documentation

Wu Jialong 2018013418

March 15, 2020

1 Comments for `bits.c`

1.1 Bit Manipulations

1.1.1 `bitAnd(x,y)`

```c
int bitAnd(int x, int y) {
  return ~((~x)|(~y));
}
```

Description: `x&y` using only `~` and `|`

Comments: See `x` and `y` as two sets X, Y, we have `x&y` to be the intersection of them $X \cap Y$ which is equivalent to $(X^c \cup Y^c)^c$.

1.1.2 `getByte(x,n)`

```c
int getByte(int x, int n) {
  return (x>>(n<<3))&0xff;
}
```

Description: Extract byte n from word `x`

Comments: The byte n from word `x` contains bit 8n to bit 8(n+1) of `x`, thus we right shift by 8n and get the last 8 bits with a mask `0xff`. We can easily get 8n by `n<<3`.

1.1.3 `logicalShift(x,n)`

```c
int logicalShift(int x, int n) {
  int TMin = 1<<31;
  int repeat_msb = ((TMin&x)>>n)<<1;
  return (x>>n)^repeat_msb;
}
```

Description: shift `x` to the right by n, using a logical shift

Comments: We know that right shift in C does arithmetic right shift, which will repeat the highest bit (namely, the Most Significant Bit). To reset these new bits shifted from left , we do XOR with `repeat_msb`, which only copies the Most Significant Bit of `x` in those new shifted bits from left and set others to 0. Note that our `repeat_msb` is constructed by trickly utilizing arithmetic right shift.

1.1.4 `bitCount(x)`

```c
int bitCount(int x) {
  int tmp2 = (0x55) | (0x55 << 8);
  int mask2 = tmp2 | (tmp2 << 16);
  int tmp4 = (0x33) | (0x33 << 8);
  int mask4 = tmp4 | (tmp4 << 16);
  int tmp8 = (0x0f) | (0x0f << 8);
  int mask8 = tmp8 | (tmp8 << 16);
  x = (x & mask2) + (x >> 1 & mask2);
  x = (x & mask4) + (x >> 2 & mask4);
  x = (x + (x >> 4)) & mask8;
  x = (x + (x >> 8));
  x = (x + (x >> 16)) & 0xff;
  return x;
}
```

Description: returns count of number of 1's in word

Comments: Let us first see the `mask2`: it has the shape `01 01 01 01 ... 01 01` in binary code. Thus after doing `x = (x & mask2) + (x >> 1 & mask2)`, we get every two bits of `x` count the number of 1 in these two bits. So do as `x = (x & mask4) + (x >> 2 & mask4);` with `mask4 = 0011 0011 ... 0011`, which makes every 4 bits of `x` count the number of 1's in these 4 bits, that is to say, $B2U(x_{k+3}, x_{k+2}, x_{k+1}, x_k)$ where $k \bmod 4 = 0$ is equal to the number of 1's on bit k+3, k+2, k+1 and k.

Notice that we MUST do the mask operation `&mask2`, `&mask4`, because it prevents those value polluting the count number when they are not masked. This kind of masking is not necessary after masking with `mask8` because the clean lowest 8 bits is enough to store the maximum of the answer 32 which only needs 6 bits.

Then we extract the lowest 8 bits by `&0xff` after all is done.

1.1.5 `bang(x)`

```c
int bang(int x) {
  x=(x>>1)|x;
  x=(x>>2)|x;
  x=(x>>4)|x;
  x=(x>>8)|x;
  x=(x>>16)|x;
  return ~x&1;
}
```

Description: Compute `!x` without using `!`

Comments: Compute `!x` is equivalent to find whether there is any 1 in `x`'s binary code. Just like what we do in `bitCount`, after `x=(x>>1)|x;`, the lowest bit of every two bits represents whether there is 1 in those two bits. And then we do the same for every 4, 8, 16, 32 bits. Finally, the lowest bit of `x` represents whether there is 1 in all 32 bits. We extract it and return its opposite.

1.2 Two's Complement Arithmetic

1.2.1 `tmin()`

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

Description: return minimum two's complement integer

Comments: It is clear by the knowledge of two's complement that `TMin` has 1 on the highest bit and 0 on others bits.

1.2.2 `fitsBits(x,n)`

```
int fitsBits(int x, int n) {
  int TMin = 1<<31;
  int neg_mask = (x&TMin)>>31;
  int abs = (neg_mask&(~x)) + ((~
      neg_mask)&x);
  return !(abs>>(n+(~0)));
}
```

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

Comments: That `x` can be represented as an n-bit, two's complement integer is equivalent to

$$-2^{n-1} \leq x < 2^{n-1}$$

First we need to determine `x` is negative or not. We get `neg_mask==0xffffffff` when `x` is negative and `neg_mask==0x00000000` when `x` is non-negative. Then we get `abs` which equals to `x` when it is non-negative and `-x-1` (but not `-x`) when negative. The problem now transforms to determine whether `abs` is less than 2^{n-1}. We just need to see the (n-1)th bit (Note that we use `~0` to denote -1 since we are not allowed to use operator `-`).

1.2.3 `divpwr2(x,n)`

```
int divpwr2(int x, int n) {
  int TMin = 1<<31;
  int neg_mask = (x&TMin)>>31;
  int bias = neg_mask & ((1<<n)+(~0));
  return (x+bias)>>n;
}
```

Description: Compute x/(2^n), for $0 \leq n \leq 30$. Round toward zero

Comments: We know that division in C always rounds toward $-\infty$. For non-negative result, that is what we expect to. But for negative result, we need a round-to-ceil division, by add a bias which is only less by 1 than the divisor, that is $2^n - 1$ in this case (we use `~0` to denote -1 once again).

1.2.4 `negate(x)`

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

Description: return `-x`

Comments: It is clear by the knowledge of two's complement that `-x` has the binary code `~x+1` .

1.2.5 `isPositive(x)`

```
int isPositive(int x) {
  int not_neg = !(x&(1<<31));
  int not_zero = !!x;
  return not_neg & not_zero;
}
```

Description: return 1 if x > 0, return 0 otherwise

Comments: We determine whether `x` is non-nagative by see the highest bit of it and determine whether it is 0 by use `!` twice.

1.2.6 `isLessOrEqual(x,y)`

```
int isLessOrEqual(int x, int y) {
  int TMin = 1<<31;
  int x_neg_y_pos = (x&TMin)&(~y&
       TMin);
  int x_pos_y_neg = (~x&TMin)&(y&
       TMin);
  int sub = y+(~x+1);
  int sub_pos = ~sub&TMin;
  return !!((x_neg_y_pos | sub_pos) & ~
       x_pos_y_neg);
}
```

Description: if x <= y then return 1, else return 0

Comments: Considering the influence of int overflow, we first determine whether `x` is negative and `y` is non-negative and denote it with `x_neg_y_pos`, and also determine whether `x` is non-negative and `y` is negative and denote it with `x_pos_y_neg`. Then for the remaining cases the difference between `x` and `y` is always in the range of `int` . We see the highest bit of the difference `sub` to determine its sign (note that we use `˜x+1` to denote `-x` since operator `-` is illegal).

1.2.7 `ilog2(x)`

```
int ilog2(int x) {
  int res = 0;
  res = (!!(x>>16))<<4;
  res = res + ((!!(x>>(8+res)))<<3);
  res = res + ((!!(x>>(4+res)))<<2);
  res = res + ((!!(x>>(2+res)))<<1);
  res = res + (!!(x>>(1+res)));
  return res;
}
```

Description: return floor(log base 2 of x), where x > 0

Comments: This method is kind of similar with binary search algorithm. We first determine whether the answer is greater or equal to 16 by seeing the highest 16 bits. If the higher 16 bits are non-zero, than the answer is added by 16 and we pay our attention to the higher 16 bits; otherwise we pay our attention to the lower 16 bits. The same things are down for those 8, 4, 2, 1 bit(s) of interest.

1.3 Floating-Point Operations

1.3.1 `float_neg(uf)`

```c
unsigned float_neg(unsigned uf) {
  unsigned exp = uf&0x7f800000;
  unsigned fra = uf&0x007fffff;
  if (!(exp^0x7f800000) && fra) // is nan
      return uf;
  return uf^(1<<31);
}
```

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

Comments: We need to process NaN specially (By extract `exp` and `frac` using mask, we can judge whether `uf` is not a number). For other cases, it is easy to inverse the highest bit which denotes the sign of `uf` and then we get the opposite.

1.3.2 `float_i2f(x)`

```c
1   unsigned float_i2f(int x) {
2     int tmp = x;
3     unsigned sgn = x&(1<<31);
4     unsigned exp = 0;
5     unsigned fra = 0;
6     if (x==-2147483648) return 0xcf000000;
7     if (x==0) return 0;
8     if (sgn){ x = -x; tmp = x; }
9     while (tmp>1){
10      exp = exp + 1; tmp = tmp>>1;
11    }
12    fra = x-(1<<exp);
13    if (exp<=23) fra = fra<<(23-exp);
14    else{
15      int len = exp-23, mid = 1<<(len-1);
16      tmp = fra; fra = fra>>len;
17      tmp = tmp - (fra<<len);
18      if ((tmp == mid && (fra&1)) || tmp >
            mid)
19        fra = fra+1;
20    }
21    return sgn + ((exp+127)<<23) + fra;
22  }
```

Description: Return bit-level equivalent of expression (float) x

Comments: (Lines 6-7) We process `-2147483648` and `0` specially: for the former, its opposite can not be represented by `int`, and for the latter, it is a denormalized number (other integers are always normalized number).

(Lines 9-12) Then we try to extract `exp` and `frac` use a while loop which divides absolute value of `x` until it turns to 0. And by ignoring the highest non-zero bit, we get `frac`.

(Lines 13-20) The last thing to do is rounding `frac`. If `exp` is not greater than 23, we do a left shift to align `frac`. Otherwise, we prune the last `exp-23` bits, and if the tail is greater than a half or equal to a half but the remaining `frac` is odd, then the remaining `frac` is added by 1, which behaves exactly as a rounding toward nearest even.

1.3.3 `float_twice(uf)`

```
unsigned float_twice(unsigned uf) {
  unsigned sgn = uf&0x80000000;
  unsigned exp = uf&0x7f800000;
  unsigned fra = uf&0x007fffff;
  if (!(exp^0x7f800000) && fra) // is nan
      return uf;
  if (!(exp^0x7f800000) && !fra) // is inf
      return uf;
  if (exp){ // normalized
      exp = exp + 0x00800000;
      if (!(exp^0x7f800000))
          fra = 0;
      return sgn + exp + fra;
  }
  // denormalized
  fra = fra<<1;
  return sgn + exp + fra;
}
```

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

Comments: We first extract `sgn`, `exp`, `frac` with masks. And we do the categorized discussion:

1. nan and inf is same with the double of themselves;
2. for normalized number, we need to add `exp` by 1, and if the result is larger than inf, we round it to inf;
3. for denormalized number, we just do a left shift to `frac` (it is fortunately to see that when the highest bit of `frac` is moved to the region of `exp` then it naturally turns to normalized number).

2 Impressions of Data Lab

By solving Data Lab, I get more familiar with bit-level representations of integers and floating point numbers and can skillfully manipulate bits to give some interesting operations.

Some of these puzzles are pretty easy but some are really challenge which makes me enjoy this process of puzzling out.