Shanghai Jiao Tong University

CS359 Computer Architecture

Project1: Manipulating Bits

Gao Chao

5142029014

October 25, 2016

1 Introduction

This project is aimed to deal with bit-level representations of integers and floating point numbers. There are series of programming puzzles, by doing this, I'm more familiar with the bits.

2 Environment

Virtual OS: Ubuntu 14.04.
Editor: Visual Studio Code.

3 Implementation

Here I will present the process to solve each puzzle including description, solution step by step and illustrations.

3.1 bitAnd

Requirement: bitAnd(int x, int y) x&y using only ∼ and |
This puzzle is easy. According to De Morgan's law, we know that ∼x|∼y = ∼(x&y), so we have ∼(∼x|∼y) = x&y.

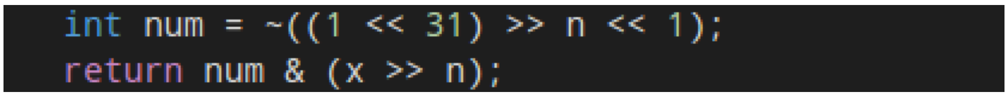
3.2 getByte

Requirement: getByte(int x, int n) extract byte n from word x
First shift the byte that we need to the lowest position, and then make & operation with 0xff, finally we can get the byte we want.

3.3 logicalShift

Requirement: logicalShift(int x, int n) shift x to the right by n, using a logical shift
LogicalShift simply move every bit a given number of bit positions, and the vacant bit-positions are filled with zeros. However, >> operation is the arithmetic shift, so we have to make sure that the front shoule be zeros. This can be accomplished

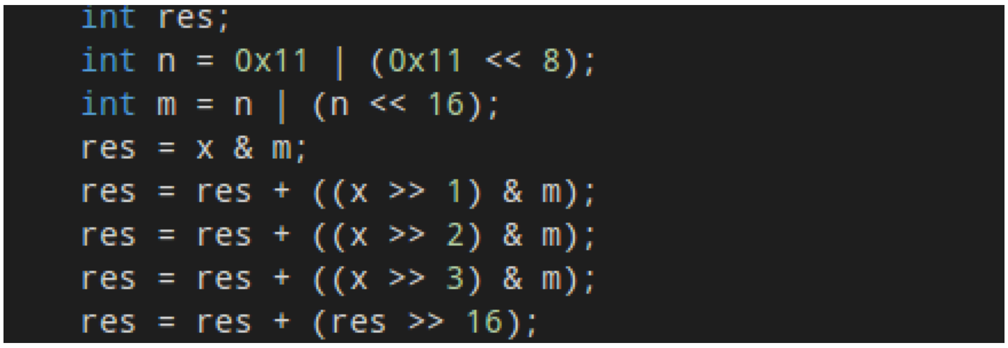
by & operation with a special number. When constructing this special number, I meet a bug, I use ∼((1<<31)>>(n - 1)), forgetting n = 0 case, so it has to be ∼((1<<31)>>n<<1).



```
int num = ~((1 << 31) >> n << 1);
return num & (x >> n);
```

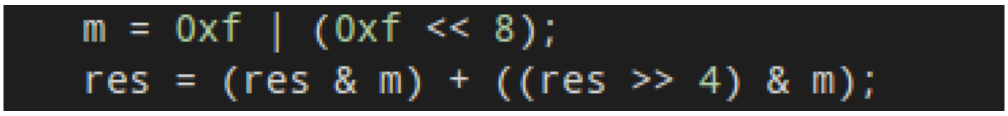
3.4 bitCount

Requirement: bitCount(int x) returns count of number 1's in word
This is really a hot potato. It takes 32 checks by brute force. Here I have a method that take four bits as a group, and calculate the count of 1's in each group. We know the largest count in each group is 4, so we can do the addition of font-16 bits and back-16 bits. Now we can get a 16-bit number that every 4 bits represents the count of 1's.



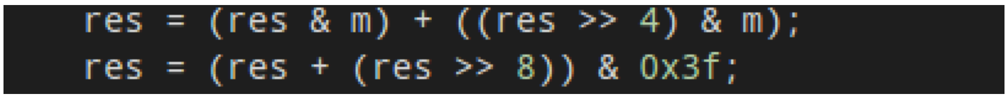
```
int res;
int n = 0x11 | (0x11 << 8);
int m = n | (n << 16);
res = x & m;
res = res + ((x >> 1) & m);
res = res + ((x >> 2) & m);
res = res + ((x >> 3) & m);
res = res + (res >> 16);
```

Now we cannot do the previous addition, because the largest count in each group is 8, after addition we may get 16 which cannot be represented by 4 bits. Therefore, we take 8 bits as a group to store the count of bits.



```
m = 0xf | (0xf << 8);
res = (res & m) + ((res >> 4) & m);
```

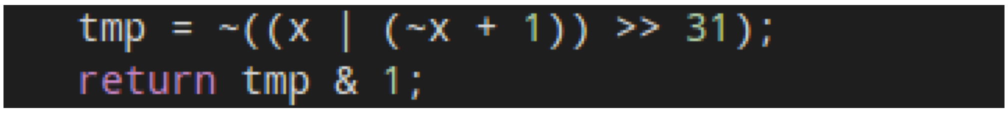
Now we can do the addition of front-8 bits and back-8 bits and get the count of 1's.



```
res = (res & m) + ((res >> 4) & m);
res = (res + (res >> 8)) & 0x3f;
```

3.5 bang

Requirement: bang(int x) compute !x without !
The result is 1 when and only when x = 0, so we aim to judge if x = 0. There is a trick that 0 is the only number whose sign bit is 0 both in number and (-number). So we have



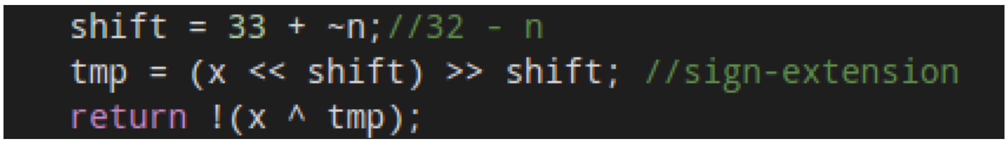
```
    tmp = ~((x | (~x + 1)) >> 31);
    return tmp & 1;
```

3.6 tmin

Requirement: tmin(void) return minimum two's complement integer
This is easy. Return 0x10000000.

3.7 fitsBits

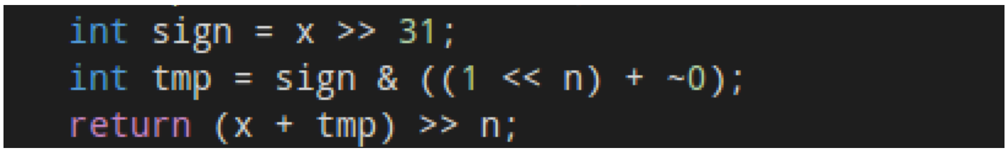
fitsBits(int x, int n) return 1 if x can be represented as an n-bit, two's complement integer
For this problem, I have a reverse thinking, when we want to extend a number not changing its value, we can take sign-extension. So we can get the lowest n bits of number and do sign-extension to see if it equals to the original number.



```
    shift = 33 + ~n;//32 - n
    tmp = (x << shift) >> shift; //sign-extension
    return !(x ^ tmp);
```

3.8 divpwr2

Requirement: divpwr2(int x, int n) compute x/(2ˆn), for 0 <= n <= 30
First I saw this puzzle, I think it is very easy, just right shift the number. However, there is a problem when the number is less than 0. For example, if the number is -6, n = 2, what I get is -2 but the correct answer should be -1. Therfore, if the number is less than 0, we can add 2ˆn - 1 to it and then shift, else we directly shift the number.



```
  int sign = x >> 31;
  int tmp = sign & ((1 << n) + ~0);
  return (x + tmp) >> n;
```

3.9 negate

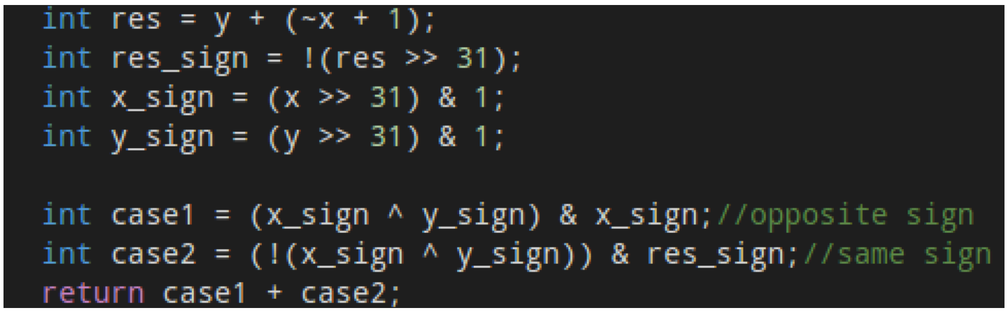
Requirement: negate(int x) return -x
This is easy. Just take bit inversion and plus one.

3.10 isPositive

Requirement: return 1 if x > 0, return 0 otherwise
At the beginning I write !(x >> 31), but I forget the x = 0 case. So the final answer should be !(x >> 31) & !!x.

3.11 isLessOrEqual

Requirement: isLessOrEqual(int x, int y) if x <= y then return 1, else return 0
First, we do the calculation y - x, and then judge the sign. However, if the two numbers have opposite signs, there exists the overflow problem. So I consider two cases seperately, if the tw numbers has opposite signs, we directly know the result which is the same as the sign of subtrahend. If the two numbers have the same signs, we can consider the sign of y - x.

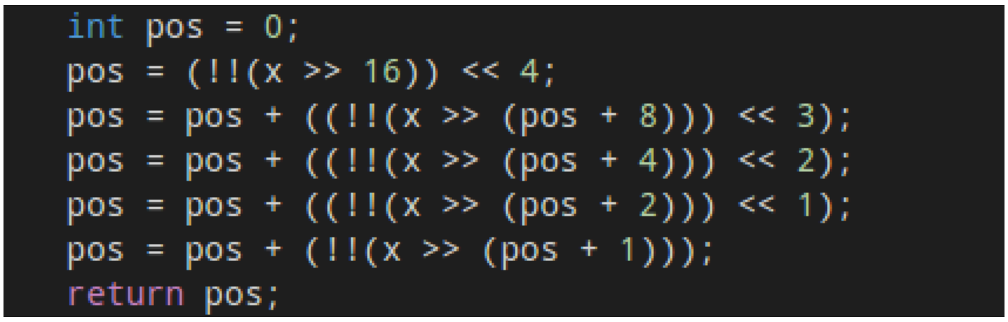


```
  int res = y + (~x + 1);
  int res_sign = !(res >> 31);
  int x_sign = (x >> 31) & 1;
  int y_sign = (y >> 31) & 1;

  int case1 = (x_sign ^ y_sign) & x_sign;//opposite sign
  int case2 = (!(x_sign ^ y_sign)) & res_sign;//same sign
  return case1 + case2;
```

3.12 ilog2

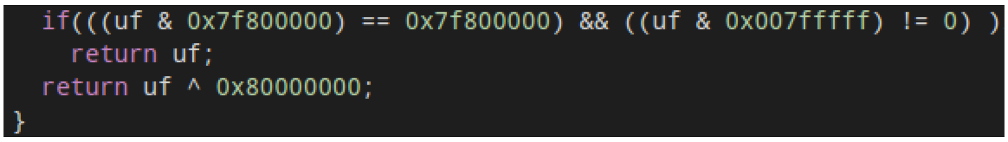
Requirement: ilog2(int x) return floor(log base 2 of x), where x > 0
For this puzzle, we aims to find the highest 1 in the number. This puzzle is very difficult for me. And I get some information from the Internet, the trick is the binary search process. First find out if the highest is in front-16 bits or back-16 bits and go on until we get the final position.



```
int pos = 0;
pos = (!!(x >> 16)) << 4;
pos = pos + ((!!(x >> (pos + 8))) << 3);
pos = pos + ((!!(x >> (pos + 4))) << 2);
pos = pos + ((!!(x >> (pos + 2))) << 1);
pos = pos + (!!(x >> (pos + 1)));
return pos;
```

3.13 float_neg

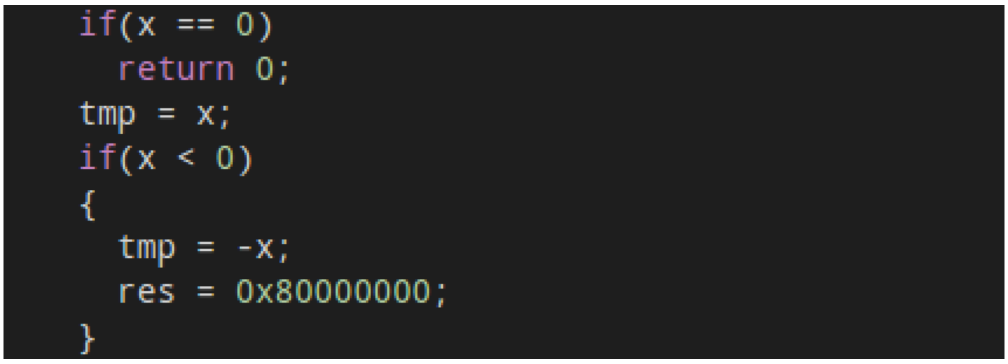
Requirement: float_neg(unsigned uf) return bit-level equivalent of expression -f for floating point argument f
The critical point is about NaN judgement. NaN's exponential field needs to be all 1, and mantissa should not be all 0. Therefore, we fist judge if it is NaN, if it is NaN, return it directly, else return -f.



```
  if(((uf & 0x7f800000) == 0x7f800000) && ((uf & 0x007fffff) != 0) )
    return uf;
  return uf ^ 0x80000000;
}
```

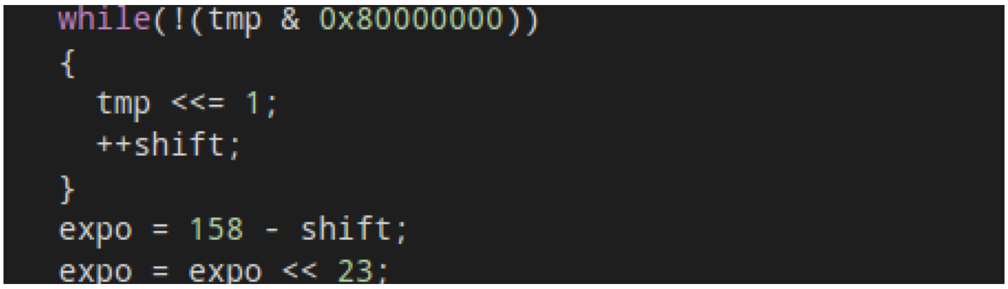
3.14 float_i2f

Requirement: float_i2f(int x) return bit-level equivalent of expression (float) x
This puzzle is very complex. First, we judge the sign, if the number is 0, return 0 directly, if the number is less than 0, change it to the positive number, and make the sign-bit be 1.



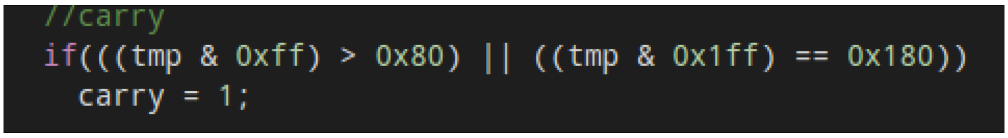
```
if(x == 0)
  return 0;
tmp = x;
if(x < 0)
{
  tmp = -x;
  res = 0x80000000;
}
```

Next, we only need to focus on the positive number. By shifting, we easily know the expotential field.



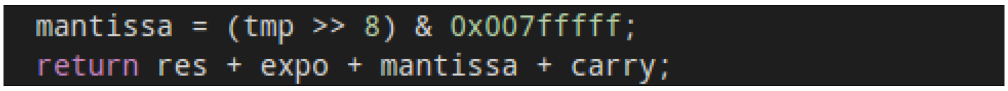
```
while(!(tmp & 0x80000000))
{
  tmp <<= 1;
  ++shift;
}
expo = 158 - shift;
expo = expo << 23;
```

However, it is a little bit complex for mantissa, we need to consider that the number may be beyond the range of single precision floating point, at this time, we should compute the carry.



```
//carry
if(((tmp & 0xff) > 0x80) || ((tmp & 0x1ff) == 0x180))
  carry = 1;
```

Finally, add the elements together, we can get the final answer.

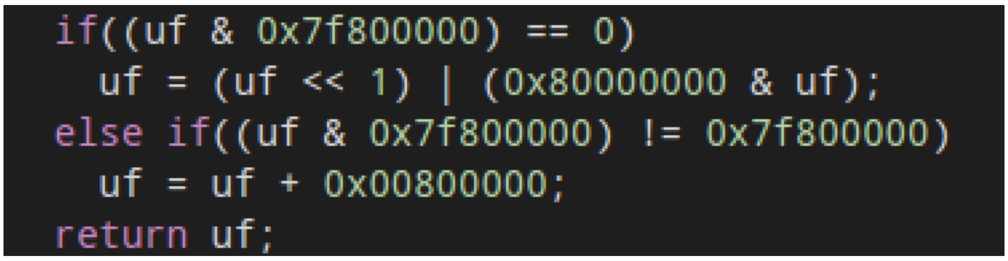


```
mantissa = (tmp >> 8) & 0x007fffff;
return res + expo + mantissa + carry;
```

3.15 float_twice

Requirement: float_twice(unsigned uf) return bit-level equivalent of expression 2*f for floating point argument f

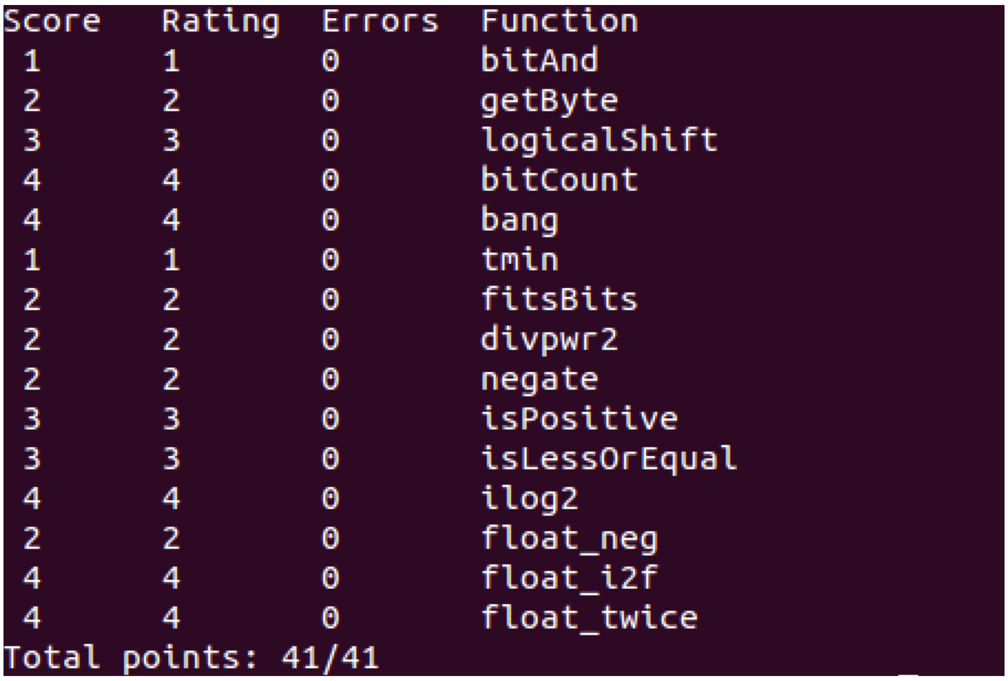
Here I notice if the expotential field is all 1, we can return the number directly. Otherwise, if the expotential field is all 0, we should do the shift, if not, we add 1 to the expotential field.



```
if((uf & 0x7f800000) == 0)
  uf = (uf << 1) | (0x80000000 & uf);
else if((uf & 0x7f800000) != 0x7f800000)
  uf = uf + 0x00800000;
return uf;
```

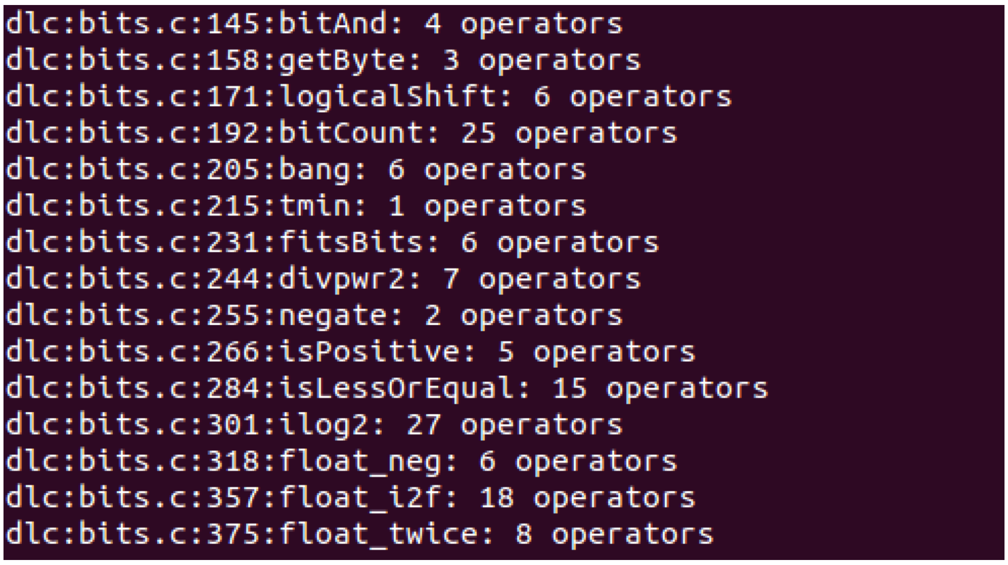
4 Result

btest



```
Score   Rating  Errors  Function
 1       1       0       bitAnd
 2       2       0       getByte
 3       3       0       logicalShift
 4       4       0       bitCount
 4       4       0       bang
 1       1       0       tmin
 2       2       0       fitsBits
 2       2       0       divpwr2
 2       2       0       negate
 3       3       0       isPositive
 3       3       0       isLessOrEqual
 4       4       0       ilog2
 2       2       0       float_neg
 4       4       0       float_i2f
 4       4       0       float_twice
Total points: 41/41
```

dlc(operator use)



```
dlc:bits.c:145:bitAnd: 4 operators
dlc:bits.c:158:getByte: 3 operators
dlc:bits.c:171:logicalShift: 6 operators
dlc:bits.c:192:bitCount: 25 operators
dlc:bits.c:205:bang: 6 operators
dlc:bits.c:215:tmin: 1 operators
dlc:bits.c:231:fitsBits: 6 operators
dlc:bits.c:244:divpwr2: 7 operators
dlc:bits.c:255:negate: 2 operators
dlc:bits.c:266:isPositive: 5 operators
dlc:bits.c:284:isLessOrEqual: 15 operators
dlc:bits.c:301:ilog2: 27 operators
dlc:bits.c:318:float_neg: 6 operators
dlc:bits.c:357:float_i2f: 18 operators
dlc:bits.c:375:float_twice: 8 operators
```

5 Summary

From this project, I learned a lot about bit-level manipualtions. Some puzzles are very interesting and difficult, thanks to the help of my mates and Internet, finally I accomplished the project.