CS 359 - Computer Architecture
Project 1: Manipulating Bits

Shanghai Jiaotong University, Fall 2016

Student ID	***Name***
5140719030	**Jun Wang**

1 Introduction

The goal of this project is to become more familiar to bit-level representations of integers and floating point numbers by filling in a series of functions to operate them. There are two compulsory things:

- Fully understand the bit-level representations of integers and floating point numbers.
- Implement the functions using operations as simply as possible, solve the "puzzles" cleverly.

The steps of solution are described in following sections.

2 Bit Manipulations

2.1 bitAnd

Sol.

This function implements **x & y** using only | and ~ .

According to *De Morgan's Laws* , the puzzle can be easily solved.

$$A \,\&\, B = \sim (\sim A \mid \sim B)$$

The code is straightforward.

□

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

2.2 getByte

Sol.

This function returns the **n*th***byte in one word. Steps are followed:

- Calculate how much bits to be shift, which is 8n, can be implemented using n<<3
- Shift the integer right, and do *and* operation with **0xff** to get the first byte.

The code is straightforward.

□

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

2.3 logicalShift

Sol.

This function returns the **logical shift** result of an integer. The difference between the logical shift result and arithmetic shift is that the bits added while shifting in logical shift is zero. To implement this, we can do ***and*** operation with zero in the higher ***n*** bits, which can be produced by shifting and inversing **0x01**. Steps:

- Shift **0x01** 31 bits left to produce **0x80000000**.
- Shift **0x80000000**(n-1) bits right to produce the number which the n higher bits are 1s, others are 0s.
- Inverse it and do ***and*** operation with the origin integer and get result.

The code is straightforward.

□

```
int logicalShift(int x, int n)
{
    int high=0x01<<31;
    int temp=high>>n;
    temp=temp<<1;
    return (x>>n)&(~temp);
}
```

2.4 bitCount

Sol.

This function returns the numbers of "1" in the given integer. The problem can be solve by adding the integer per 2 bits, per 4 bits, per 8 bits, per 16 bits. The add operation can be done using different *mask* numbers. Steps:

- Produce relative ***mask*** numbers used to mask integer when adding by using *shift* and *or* operations.
- Do *shift* and *and* operations to the integer to add bits per 2 and 4 and 8 and 16 bits.

The code is straightforward. Part of codes is following.

□

```
int bitCount(int x)
{
    int result;
    int temp_mask1=(0x55)|(0x55<<8);
    int mask1=(temp_mask1)| (temp_mask1<<16);
    //...
    //produce mask2, mask3, mask4, mask5
    //for per 2, 4, 8, 16 bits in the same way.

    result=(x&mask1)+((x>>1)&mask1);
    result=(result&mask2)+((result>>2)&mask2);
    result=(result&mask3)+((result>>4)&mask3);
    result=(result&mask4)+((result>>8)&mask4);
    result=(result&mask5)+((result>>16)&mask5);

    return result;
}
```

2.5 bang

Sol.

This function returns !n without "!" operator. To implement this, just need to judge the integer is 0 or not.

Since only 0 and 0x80000000 satisfy x=∼x+1, so x|∼x+1==0 if and only if x=0. So the expression *x|∼x+1* can be used to make judgement. Steps:

- Calculate *x|∼x+1* .
- return result.

The code is straightforward.

□

```
int bang(int x) {
    int neg;
    int result;
    neg=~x+1;
    result= ~(neg|x);
    return (result>>31)&0x01;
}
```

3 Two's complement Arithmetic

3.1 tmin

Sol.

This function returns most negative two's complement integer. According to the representation of two's complement of integer, **0x80000000** represents the most negative integer. It can get by shifting 0x01 directly.

The code is straightforward.

□

```
int tmin(void)
{
    return 0x01<<31;
}
```

3.2 fitsBits

Sol.

This function returns 1 iff the integer can be represented by n bits. Finding that if the integer can be represented, its **n*th*** to **32*ed*** bits are all 1 or all 0, the same to the signbit. So firstly record the signbit, then shift the integer and compare with the signbit. Steps:

- Record the signbit.
- Shift the integer $n-1$ bits right.
- According to the rule that the signbit and the shift result must be the same, return the result.

The code is straightforward.

□

```
int fitsBits(int x, int n)
{
    int head = x >> 31;
    int bias = n + ~0;            //n-1
    int pos=~head & !(x>>bias);
    int neg=head & !((~x)>>bias);
    return pos|neg;
}
```

3.3 divpwr2

Sol.

This function returns $x/2^n$. For positives, shift the integer right n bits can get the result. For negatives, because negatives are represented by two's complement, after shift right, we need add 1 if the 1 already added to the complement is shifted out. This can be realized by adding $n-1$ 1s to the shifted number.

So use *head* to record the sign and produce the adding number. The adding number must have $n-1$ 1s iff *head* is not zero (the number is negative).Steps:

- Record the signbit.
- Shift $0x01$ to produce the number which has $n-1$ 1s.
- Use the signbit and the add-number to produce the bias number, which will be added to the shifted number.

- Shift the integer and add the bias to get the result.

The code is straightforward.

□

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

3.4 negate

Sol.

This function produces $-x$ without using "$-$" operator. For both positives and negatives, inverse and plus one can get $-x$.

The code is straightforward.

□

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

3.5 isPositive

Sol.

This function returns 1 iff the integer >0. Use the signbit to judge the negative and positive. Besides, must return 0 when it comes 0. This can realized using $!x$ and logical arithmetic with signbit.

- Record the signbit.

- Do logical arithmetic with x itself.
- Return result.

The code is straightforward.

□

```
int isPositive(int x)
{
        int head= (x>>31)&0x01;
        return !(head^(!x));
}
```

3.6 isLessOrEqual

Sol.

This function returns 1 iff the integer >0. Use the signbit to judge the negative and positive. Besides, must return 0 when it comes 0. This can realized using !x and logical arithmetic with signbit.

- Record the signbit.
- Do logical arithmetic with x itself.
- Return result.

The code is straightforward.

□

```
int isPositive(int x)
{
        int head= (x>>31)&0x01;
        return !(head^(!x));
}
```

3.7 ilog2

Sol.

This function computes the log2 value of the input integer. This value equals to the highest bit which is 1.

Use variables to record bits or bytes which does not equals to zero firstly. Steps:

- Spilt the integer to four bytes.
- Use four variables to record the first byte which does not equal to zero.
- Select the first byte which is not zero according to variables in Step 2.
- Use eight variables to record the first bit in the byte which is not zero.
- Calculate the number of bits which is the first 1 according the variables.
- Return the number calculated.

The code is straightforward.

□

```
int ilog2(int x) {
    int byte3=(x>>24)&0xff;
    //byte2,byte1,byte0 is the same.

    int i3=!!byte3;
    //i2,i1,i0 is the same
    int i=i3+i2+i1+i0+~0;
    int highbyte=x>>(i<<3);

    int b7=(highbyte>>7)&1;
    //b6,..,b0 is the same

    int f7=b7;
    int f6=f7|b6;
    // f5,...,f0 is the same
    int f=f7+f6+f5+f4+f3+f2+f1+f0+~0;

    return (i<<3)+f;
}
```

4 Floating-Point Operations

4.1 float-neg

Sol.

This function returns the negative of the input float, which is transmitted in unsigned. Because the function needs to return a "nan" when input is "nan", the "nan" needs to be recognized.

To recognized "nan", firstly, spilt the integer to get exponential and magnitude portion. Compare these bits with $0x7f800000$, if it is not a "nan", change the signbit.

- Spilt the integer to get exponential and magnitude portion.
- Compare to $0x7fffffff$.
- Change the sign bit.

The code is straightforward.

□

```
unsigned float_neg(unsigned uf) {
    unsigned temp=uf & 0x7fffffff;
    if(temp>0x7f800000) return uf;
    else return uf^0x80000000;
}
```

4.2 float-i2f

Sol.

This function returns the representation of floating point numbers. To get it, we need to figure out the exponential,magnitude portion and the sign bit. Use a while-loop to shift the integer to the bit which is second to the first 1, and this is the magnitude portion. Use a variable to count the times of the shift, and this is the exponential portion. The sign bit is the same to the integer.

- Pretreat the input integer.
- Use a while-loop to shift the number to get exponential and magnitude portion.
- Round the exponential portion.
- Calculate the representation.

The code is straightforward.

□

```
unsigned float_i2f(int x)
{
    unsigned shiftLeft=0;
    unsigned temp, aftershift , round;
    unsigned absX=x;
    unsigned sign=x&0x80000000;
    //special case
    if (!x)      return 0;
    //if x < 0, abs_x = -x
    if (sign)
    absX=-x;
    aftershift=absX;
    //count shift_left and after_shift
    while (1)
    {
        temp=aftershift;
        aftershift=aftershift <<1;
        shiftLeft=shiftLeft+1;
        if (temp & 0x80000000)  break;
    }
    if ((aftershift & 0x01ff)>0x0100)
        round=1;
    else if ((aftershift & 0x03ff)==0x0300)
        round=1;
    else
        round=0;

    return sign + (aftershift>>9) + ((159-shiftLeft)<<23) + round;
}
```

4.3 float-twice

Sol.

This function returns $2 * f$ of the input f. This question have several conditions. If the expo-portion is all 1s, the output needs not change. If the expo-portion is zero. the magnitude portion needs to shift. Else, the expo-portion needs to plus one.

- Spilt the integer to exponential and magnitude portion.
- Judge the exponential portion.
- According to the judgement to change the exponential or magnitude portion

The code is straightforward.

□

```
unsigned float_twice(unsigned uf)
{
    unsigned sign=uf&0x80000000;
    unsigned expo=uf&0x7f800000;
    if(expo)
    {
            if(expo!=0x7f800000)
            uf=uf+0x00800000;
      }
      else
         uf= (uf<<1)+sign;
      return uf ;
}
```