Lab 6: Bit Operations in C (subtitle: "The Trouble with Nibbles" [1])

Goals

1. Understand operations on individual bits of an integer value
2. Develop a better understanding of the underlying representation of data on a system.

Background - bit operations

A bit can have one of two values: 0 or 1. The C language provides four operators that can be used to perform bitwise operations on the individual bits of any integer data type (signed or unsigned char, short, int, and long). The *complement* operator is *unary* in that it requires a single operand. The other three are binary and require two operands.

1. *complement:* The operator *complement (~)* converts 0 bits to 1 and 1 bits to 0. For example,
```
x =   01100010
~x =  10011101
```
2. *and:* The *and (&)* operator takes two operands. A bit in the result is 1 if and only if both of the corresponding bits in the operands are 1. For example:
```
x =   01100010
y =   10111010
x&y = 00100010
```
3. *or:* The *or (|)* operator also takes two operands. A bit in the result is 1 if and only if either one or both of the corresponding bits in the operands are 1. For example
```
x =   01100010
y =   10111010
x|y = 11111010
```
4. *xor:* The *xor (^)* operator takes two operands. A bit in the result is 1 if and only if exactly one of the corresponding bits in the operand is 1. For example
```
x =   01100010
y =   10111010
x^y = 11011000
```

In addition to the boolean operations, C also provides binary operators that can be used to shift all the bits of any integer type to the left or the right.

5. The *left shift* operator << is used to shift bits to the left. The left operand is the integer value to be shifted, and the right is the number of bits to shift it. For example,
```
x =     01100010
x<<4 =  00100000
```
   Assuming this is an integer of type char which contains only 8 bits, the high order four bits(from the left) are lost and 4 new 0 bits are shifted in from the right. Left shifting by *n* bits is a very efficient way to multiply a value by 2 to the *nth* power.

6. The *right shift* operator >> is used to shift bits to the right. The operands are identical to those used in left shift. On the left of the operator goes the bit string and on the right goes the number of bits to shift. However, an important difference is that the value of new bits shifted in from the left is machine dependent. Nevertheless, it is commonly the case that the new bit will be 0 unless (1) the left operand is a

[1] This sub-title is a measure of your "geek" quotient. Given I came up with this that should tell you something (mainly my age). If this sub-title doesn't ring a bell then try Goggling "The Trouble with Tribbles".

signed integer and (2) the high order bit was originally a 1. In this case the integer value represents a negative number and the shifting in of 1 bits preserves the negativity. An example,

```
x =     01100010
x>>4 = 00000110
```

A right shift of *n* bits is a very efficient way to divide by 2 to the *nth* power.

Masking

Masking is a commonly used technique in processing individual bits of an integer word. In this technique, bits in the *mask* corresponding to bits of interest in the target word are set to 1 and the others are set to 0. The mask and the target word are then and'ed. Here are some examples of masking (assume x is an integer):

```
if ((x & 0x1) == 1) printf("x is odd\n");    /*Simple test for even/odd     */
right2 = x & 0x3;                            /*Extract right 2 bit values   */
bit10 = (x>>10) & 0x1;                       /*Extract bit 10 value         */
righthalf = x & 0xffff;                      /*Extract right half of integer*/
```

Nibbles

The term "nibble" is used to describe 4 bits. In particular, a byte (8-bits) is composed of 2 nibbles. A 4-byte (32 bit) integer is composed of 8 nibbles. So we might view an integer as follows:

nibble 0	**nibble 1**	**nibble 2**	**nibble 3**	**nibble 4**	**nibble 5**	**nibble 6**	**nibble 7**

The decimal value "2946_{10}" in binary (base 2) is equal to 101110000010_2. In memory the value would be stored as illustrated by the following 8 nibbles:

0000	**0000**	**0000**	**0000**	**0000**	**1011**	**1000**	**0010**

Note that nibble 7 has the value 0010_2, and nibble 5 has the value 1011_2.

Hexadecimal

In addition to decimal (base 10) and binary (base 2), hexadecimal (base 16) is commonly used to represent values in our field. A single digit in hexadecimal can be represented in 4 bits (a nibble). The decimal, binary and hexadecimal values of the decimal values 0-15 are shown in the following table:

Decimal	Binary	Hexadecimal		Decimal	Binary	Hexadecimal
0	0000	0		8	1000	8
1	0001	1		9	1001	9
2	0010	2		10	1010	A
3	0011	3		11	1011	B
4	0100	4		12	1100	C
5	0101	5		13	1101	D
6	0110	6		14	1110	E
7	0111	7		15	1111	F

Note that the letters "A-F" are used to represent the 6 additional "digits" in the hexadecimal system. Keep in mind that 15 in decimal, and 1111 in binary and F in hexadecimal are ALL the same value -- they are just different ways of representing the same value (we could extend this to roman numerals, i.e. "XV" -- this is just another representation of the same number).

In C we can represent a hexadecimal value by prefixing it with a "0x". For example "x=0xf;" sets "x" to the decimal value 15 (or the binary value 1111). The statement "x=0xff" would set x to the binary value "11111111", which is 255 in decimal.

When we do masking, we often find that using the hexadecimal notation is useful (since it corresponds more directly to bits and nibbles).

We can ask C to print a value in hexadecimal (as opposed to decimal) by using the "%X" format item, for example if y=12, then

```
        printf("y=%d (%X)\n", y, y);
```

would print the value of y in both decimal and hexadecimal, e.g.:

```
        y=12 (C)
```

Tasks

Create a “**lab6**” directory and copy the lab files to the directory using:

```
        scp access.cs.clemson.edu:/group/course/cpsc210/Public/lab6/lab6.tar .
```

or

```
        scp userid@access.cs.clemson.edu:/group/course/cpsc210/Public/lab6/lab6.tar .
```

if the userid you are logged in under is different from your Clemson userid. Remember that the “.” at the end is important.

Next untar the file, i.e.:

```
        tar –xvf lab6.tar
```

You should now have the files “**nibbletest.c**”, “**nibbles.c**” and “**nibbles.h**” in your lab6 directory. **Note:** the provided “nibbles.c” includes dummy “do-nothing” routines – but it will save you from having to enter the function headers.

Also note that there is a “Makefile”. You can compile the code by simply typing “`make`”.

In this lab you will create three functions which perform "nibble" operations on an unsigned 32 bit int.   The following standard terminology will be used: the right-most bit of an integer will be called bit 31, and the right-most nibble will be called nibble 7.

Modify the stubbed routines in nibbles.c to include implementations of t the following functions:

1. **`unsigned int nget(unsigned int val, int position)`**

   This function should return the value of the nibble in position "position" of the integer. For example, if "x" is:

1111	**0000**	**0000**	**0000**	**0000**	**1011**	**1000**	**0010**

   then `nget(x, 5)` should return the value 1011_2 in binary (which is the value 11 in decimal).

2. **`unsigned int nset(unsigned int val, unsigned int nVal, int position);`**

The nibble at "position" in "val" should be set to "nVal" and the value returned. For example, if x is:

0000	**0000**	**0000**	**0000**	**0000**	**1011**	**1000**	**0010**

then "`y = nset(x, 0xe, 6);`" would return:

0000	**0000**	**0000**	**0000**	**0000**	**1011**	**1110**	**0010**

Note that "0xe" is hexadecimal for "1110_2" in binary (or 14 in decimal).

This is probably the most challenging of the three functions. You can only set the bits using AND (&) and OR (|) operations. This is definitely a place where masking comes into play. Suppose we had an integer that was all binary 1's EXCEPT for the bit positions that correspond to the nibble's position, e.g. for nibble 6:

Mask	**1111**	**1111**	**1111**	**1111**	**1111**	**1111**	**0000**	**1111**

Note that if we then did an AND of this mask with the original value that would clear the nibble's value to binary zeros but leave the rest of the integer unchanged, i.e. for nibble 6:

Original after AND	**0000**	**0000**	**0000**	**0000**	**0000**	**1011**	**0000**	**0010**

Now assume we have shifted the new nibble value that we want to insert to the corresponding bits of the nibble, e.g. for 0xe and nibble 6:

New nibble value left shifted	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**1110**	**0000**

Now if we did an OR the OR would only change the nibble's value and would give the desired result.

Final Result	**0000**	**0000**	**0000**	**0000**	**0000**	**1011**	**1110**	**0010**

The remaining trick is how do we get the initial mask of all 1's except in nibble's position? How about if we start off with an integer value that is all zeros except for the right-most 4 bits (e.g. 0xf). Now shift that value left to the nibble's position, and then take the COMPLEMENT (~). So for nibble 6 the sequence to create an appropriate mask would be:

a) Initial mask

Initial mask	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**1111**

b) left shift 4 bits to position of nibble 6,

After left shift	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**1111**	**0000**

c) and complement value:

Mask after complement	**1111**	**1111**	**1111**	**1111**	**1111**	**1111**	**0000**	**1111**

3. **`unsigned int nlrotate(unsigned int val)`**

This function should rotate nibbles LEFT 4 bit positions (1 nibble). The nibble in position 0 should be moved to nibble position 7. For example, if x is:

1111	0000	0000	0000	0000	1011	1000	0010

Then `"y=nlrotate(x);"` would return the value:

0000	0000	0000	0000	1011	1000	0010	1111

Note the "1111_2" in nibble position 7 that originally was the value of the left nibble.

Hint: You can call nget() and nset() from within nlrotate() if you wish.

Note: to be on the safe side, values that will be shifted should be declared as "unsigned int".

Compiling and Testing

A “make” file is provided which produces the executable file “./lab6”. Here is the expected output.

Submission

Use the submit script we installed previously to submit your list.c and listtest.c modules, i.e.:

```
        ~/submit cpsc210 section1/lab6 nibbles.c
```

or

```
        ~/submit cpsc210 section2/lab6 nibbles.c
```

depending on your section number. If the userid you are logged into does not match your Clemson userid, you should instead enter the command:

```
        ~/submit -u clemson-userid cpsc210 section1/lab6 nibbles.c
```

or

```
        ~/submit -u clemson-userid cpsc210 section2/lab6 nibbles.c
```

It is your responsibility to make sure that you have correctly and successfully completed the submission.