Lecture 17
Bit Operations

In this lecture
- **Background**
- **Left Shifting**
- **Negative Numbers, one's complement and two's complement**
- **Right Shifting**
- **Bit Operators**
- **Masking the Bits**
- **Getting the Bits**
- **Setting the Bits**
- **Binary Files**
- **Bit fields**
- **More Exercises**

C is a powerful language and allows programmer many operations for bit manipulation. Data can be accessed at the bit level to make operations and storage more efficient. As you will see, bit operations can be used to do many things including setting flags, encrypting and decrypting images as we will implement in one of the lab assignments. Bit operations can also be used to efficiently pack data into a more compressed form. For example, an entire array of 16 boolean values can be represented by just 2 bytes of data or an IP address such as 192.168.1.15 can be packed into 32 bits of storage. This can come very handy in cases where data needs to be transmitted through a limited bandwidth network or in cases where we simply need to store data more efficiently for better memory management in the application. For example, when programming mobile devices with limited memory, you may want to work at the bit level to make things more efficient and save memory. Also understanding bit operations can be useful in writing device drivers and many other low level applications. First we will discuss the bit shift operations.

Left Shifting
Think of the following possibility. Suppose we want to multiply an unsigned integer by 2. We can simply shift all bits to the left by one position (assuming no overflow). For example, if a 32-bit unsigned integer 73 is given as

00000000 00000000 00000000 01001001

Then shifting the bits by 1 position to the left gives us the bit pattern for unsigned 146

00000000 00000000 00000000 10010010

So we can write

unsigned int x = 73;
x = x<<1; or simply x <<= 1;

The operator << is used to shift the bit patterns of a variable. For example, x << n shifts the bit patterns of x by n positions resulting in the number $x*2^n$ assuming there is no overflow.

It should be noted that as we left shift, the missing bits on the right are filled by 0's.

Negative Numbers, One's Complement and Two's Complement

Signed data is generally represented in the computer in their two's complement. Two's complement of a number is obtained by adding 1 to its one's complement. So how do we find one's complement of a number? Here is the definition
One's complement of x is given by **~x**. Obtain the one's complement of a number by negating each of its binary bits. For example one's complement of 30 is (represented as a 16-bit short int)
30 = 16 + 8 + 4 + 2 = 00000001 11100000 ⬅ binary 30
~30 = 11111110 00011111 ⬅ its one's complement

The two's complement of the number is obtained by adding 1 to its one's complement. That is, the two's complement of 30 is obtained as follows.
```
   11111110 00011111
              + 1
   ----------------
   11111110 00100000
```
Hence −30 is represented as its two's complement, that is
~30 + 1 = **11111110 00100000**

Exercise 17.1: Perform binary addition of 34 + (−89) using two's complement of the negative number.

Right Shifting

The bit pattern of a variable can be shifted right using bit operations. For example, **x >> n**, shifts the bit pattern of x by n positions to the right. The missing bits on the left are filled by the value of the highest order bit (0 or 1).

Example:
-73 is represented as 11111111 11111111 11111111 10110110
Using a 2's complement.

-73 >> 2 will result in the bit pattern

11111111 11111111 11111111 11101101

Question: What number does this represents?

Bit Operators Table

Operator	Meaning
<<	Left Shift
>>	Right Shift
|	Bitwise OR
&	Bitwise AND
^	Exclusive OR
~	One's complement
1+~x	Two's complement of x

We have learned the basic bit operations such as left shift (<<), right shift (>>), and the above table gives other bit operations such as bitwise OR (|) and bitwise AND (&) etc.

The basic logic circuits for AND, OR or NEGATION are given as follows

The Bitwise AND (&)

A	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

The Bitwise OR (|)

A	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

The Bitwise NEGATION (~)

A	~a
0	1
1	0

We note that just applying a bit operation to a variable x, does not change its value. That is, x | 2 gives a value equivalent to 2nd bit of x set to 1. But it does not change the value of x, unless we do;

x |= 2 ;

Meaning that x = x | 2

Each one is clearly useful for many operations that require bit extraction and masking of bits. For example, to extract the jth bit of a variable w, one can shift w to right by j places and then bitwise AND with 1. That is, **(w>>j)&1** is either 1 or 0 depending on j^{th} bit of w is 1 or 0.

We can set the first 4 bits of x to 1 by doing **x | 0x0F**
In this case, 0x0F represents a hex value of one byte where first 4 bits are 1's and next 4 bits are 0's. That is 00001111

Exercise 17.2: What would be the outcome of x & 0x0F?

Exercise 17.3: What would be the outcome of x & 0127, where 0127 represents the number 127 that is in octal form?

XOR operation

Another operator of interest is the XOR gate. The bitwise **exclusive OR** operator is given by **^**

Here is the table for XOR

First bit a	Second bit b	XOR a^b
0	0	0
0	1	1
1	0	1
1	1	0

Example: **0001 ^ 0101** will result in **0100**

Adding Two Numbers

We can use ^ operator to implement a function that adds two numbers, bit by bit. For example, sum of just two bits x and y is given by **x^y** with a carry **x&y**. A full adder that goes through all bits and adds values is given as follows. A_i denotes the ith bit of A and B_i denotes the ith bit of B and
C_{in} denotes the carry in and C_{out} denotes the carry out.

$$S_i = (A_i \wedge B_i) \wedge C_{in}$$
$$C_{out} = (A_i \& B_i) | ((A_i \wedge B_i) \& C_{in})$$

We will not go into the details of how to derive the sum and carry functions as shown above. However, completing a table that shows bit values, carry in, sum and carry out may give some insight into how formulas are derived. See classwork folder in democode for an implementation of addition for two bytes.

Masking the Bits

In many instances we need to find a specific bit or a group of bits. For example, given an IP address 192.168.2.16 as a 32-bit unsigned int we may want to extract 192 or last 8 bits of the 32-bit word. This can be done using an appropriate mask (perhaps with shifting) and applying a bitwise AND operation. Another interesting application of bit manipulation is finding the remainder of an unsigned integer when divided by, say 2. You simply have to find out the first bit of the number. We can accomplish this by "masking" the number with specific value. Lets us consider 73. The bit pattern is

```
00000000 00000000 00000000 01001001
```

Suppose we take **"bitwise AND"** or apply operator & as follows

```
00000000 00000000 00000000 01001001  = 73
             &
00000000 00000000 00000000 00000001  =  1
```

This results in the value 0 or 1 depending on the last bit of 73. In this case we get 1. Therefore the remainder when 73 divided by 2 is 1. For example we can test if a number of odd or even by simply saying,

```
int x = 73;
if (x&1) printf("The number is odd\n");
else printf("the number is even\n");
```

Accessing Bits

Masking a variable allows us to look at individual bits of the variable. For example, let us assume that we need to write a function called **getBit(int w, unsigned j)** that allows us to access the jth bit of an int variable w. The following code does it.

```
#define MASK(j)    (1<<j)
int getBit(int w, unsigned j){
   return (( w & MASK(j)) == 0) ? 0 : 1;
}
```

Note that 1 is shifted to left by j bits and then taken bitwise & with w.

Alternatively, we can write a macro getBit(w,i) as follows.

```
#define getBit(w,i) ((w>>i)&1)
```

Exercise 17.4: Write a function, printBinary(unsigned int w) that prints the binary representation of w.

Setting the Bits

Another useful application of bit operations is setting the bits of a variable. For example, UNIX file system uses a 16-bit quantity known as the file mode. Assuming a 16-bit word,

f e d c b a 9 8 7 6 5 4 3 2 1 0

The bits 0-8 are reserved for file access permission r w x for other, group and owner. The next 3 bits are reserved for the execution style, and the last 4 bits are reserved for the file type. For example, a file that gives read, write, execute access to group, other and owner may have its file access permission as **rwxrwxrwx** (bits 0-8)

A good application to use setbit is to consider the chmod command

➢ **chmod 761 file.txt;**
sets the rights to the file.txt as follows.

-rwxrw---x 1 guna staff 3 Feb 19 12:07 file.txt

This would require us to set the bits of the word that represents the file access permission. So if we are writing our own chmod function, we can manipulate the bits of the file access permission structure to achieve the results.

First, let us learn how to set a specific bit of a word. We can use the following function.

```
#define MASK(j)    (1<<j)
int setBit(int w, unsigned j, short value){
  if (value == 0) return (w & ~MASK(j));
  else if (value == 1) return w | MASK(j);
  else return w;
}
```

So to set the bit 4 of w = 000000000 to 1, we can use

w = setBit(w,4,1)

The result is **w = 000010000**

In this function we introduced a new bit operator, the **"bitwise OR"** or |
Bitwise | applies the "OR" operator to each corresponding bit of the two words x and y. For example
00011001 | 10000100 = 10011101

Binary Files

Any file can be regarded as a stream of bytes. If the format of the file is understood, like an ASCII file, then we can use functions like fscanf or fprintf to read and write formatted I/O. However, a binary file is regarded just as a string of bytes. So how do we read files that are considered binary files such as a .bmp file? The header file <stdio.h> contains a function fread that can be used to read data from a binary file. The fread and fwrite prototypes are given as follows.

NAME
      fread, fwrite – binary stream input/output

SYNOPSIS
      #include <stdio.h>

      size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

      size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

DESCRIPTION
      The function fread reads nmemb elements of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.

      The function fwrite writes nmemb elements of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr.

In order to use the fread function, we need to define an array of bytes, say ptr, and allocate enough memory to hold **nitems** of **size** each. All bytes will come from a FILE stream. For example, suppose we would like to read first 54 bytes of image.bmp file. The first 54 bytes hold the header information of the file.

FILE* infile = fopen("image.bmp", "r");
void* ptr = malloc(54);
fread(ptr,54,1,infile);

Upon successful completion of the fread function, it returns the number of items read. If a read error occurs or end-of-file encountered, then it returns a number less than nitems.

Similarly, the function fwrite can be used to write binary output to an output stream.

Bit Fields

In implementing something like UNIX file access information, we can use a concept called Bit Fields. Bit fields provide a way to **pack** integer components into a memory block that is smaller than the size typically required. For example, we can define

```
struct {
   unsigned leading : 3;
   unsigned flag1 : 1;
   unsigned flag2 : 1;
   trailing : 11;
} flags;
```

This packs three fields into 16-bits. The :3 indicates for example, 3 bits are assigned for the leading field. The fields can be accessed by flags.leading = 1;   flags.flag1 = 0; etc. However the fields within the struct are not variables and therefore cannot be used with & the address operator. The whole struct cannot be printed using printf, but individual fields can be printed using printf as follows.

```
printf("The leading field is %d \n", flags.leading);
```

We can also print the bit pattern of flags.leading using the getBit function defined as follows.

```
#define MASK(j)    (1<<j)
int getBit(int w, unsigned j){
   return (( w & MASK(j)) == 0) ? 0 : 1;
}

for (i=2;i>=0;i--)
   printf("%d",getBit(flags.leading,i));
```

The bit packing is an efficient way to package data so the technique can be used in a device with limited memory or where bandwidth for sending data is limited (eg: Bluetooth).

Bit packing is used in maintaining file permission of a file in the unix system as follows.

For example access rights to a file can be stored using a structure as follows.

```
struct access {
     unsigned int o_x : 1; // execute permission for others
     unsigned int o_w: 1; // write permission for others
     unsigned int o_r : 1; // read permission for others
     unsigned int g_x: 1; // execute permission for group
     unsigned int g_w : 1; // write permission for group
     unsigned int g_r: 1; // read permission for group
```

```
        unsigned int u_x : 1; // execute permission for owner
        unsigned int u_w: 1; // write permission for owner
        unsigned int u_r : 1; // read permission for owner
        unsigned trailing : 7; // some bits for other fields
} access;
```

The 1-bit fields are aligned in the word starting with the lowest order bit. For example to give execute rights to "other" we can do the following.
access.o_x = 1;

More Exercises

17. 4   Write a function that prints an unsigned integer in its octal representation (without using %o). Hint: shifting can be used here.

17.5   Write a function that prints an unsigned integer in its hexadecimal representation(without using %x)

17.6   Write a function that multiplies two unsigned short ints using just bit shifting and bit addition. Hint: Any number can be represented as a sum of the powers of two.

17.7   Write a function setbits(w,i,j,value) that sets the bits from i to j (inclusive) to given value (0 or 1). Assume w is an unsigned int (32-bits). Check the ranges to make sure i and j falls within the range 0-31

17.8   Explain how you can use the bit operators to see if a number is a power of 2.

17.9   Find the sum of 67 + (-32) using bitwise addition.

17.10   Write a function, int countBits(w,value) that counts and returns the bits of w that are set to value. Assume w is a 32-bit unsigned int and value can be 0 or 1.

17.11   A device controller is a program that communicates with Operating system to make sure it provides correct information about the device to the OS so the operating system can perform the functions required by the device. For example, device variables (or flags) such as device_ready(1-

bit),device_spinning(1-bit),device_error(1-bit),write_protected(1-bit), device_sector(5-bits),device_errorcode(8-bits), device_command(5-bits),device_track(8-bits)etc..can use bit fields to pack information about the status of these flags. Describe how you would pack all this information into a 32-bit register. Based on your design, write the functions, setflag(w,flag_type) and getflag(w,flag_type) that returns the value of a flag (ready, sector, errorcode etc) using a mask flag_type.

17.12   needs to be sent over a low bandwidth communication network. Three parameters, flag1, flag2 and flag3 needs to be packed into a struct flagtype. The value ranges for flag1, flag2, and flag3 are 0-7, 0-63, and 0-255 respectively. Design the smallest possible struct that can hold a record that contains flag1, flag2 and flag3.

17.13   Suppose you are to write a function, setFlag(unsigned value, unsigned flag) that assigns the value to a specific flag. The flag can be 1, 2, or 3. Your function needs to check the value ranges to make sure the flag values are valid.

17.14   Write a function getFlag(unsigned flag, unsigned start, unsigned end), that will print the bit pattern of flag from start to end. Flag is given as 1,2, or 3. You need to check if start and end falls within the limits of the specific flag.