Jeremy Rios-Martinez

COMSC 265

Assignment 20

[**Bitwise operation**](http://en.wikipedia.org/wiki/Bitwise_operation)

Bitwise operations are necessary for much low-level programming, such as writing device drivers, low-level graphics, communications protocol packet assembly and decoding.

**op1 & op2** The **AND** operator compares two bits and generates a result of 1 if both bits are 1; otherwise, it returns 0.

**op1 | op2** The **OR** operator compares two bits and generates a result of 1 if the bits are complementary; otherwise, it returns 0.

**op1*^* op2** **EXCLUSIVE-OR** operator compares two bits and returns 1 if either of the bits are 1 and it gives 0 if both bits are 0 or 1.

**~op1** The **COMPLEMENT** operator is used to invert all of the bits of the operand.

**op1 >> op2** The **SHIFT** **RIGHT** operator moves the bits to the right, discards the far right bit, and assigns the leftmost bit a value of 0. Each move to the right effectively divides op1 in half.

**op1 << op2** The **SHIFT** **LEFT** operator moves the bits to the left, discards the far left bit, and assigns the rightmost bit a value of 0. Each move to the left effectively multiplies op1 by 2.

Although machines often have efficient built-in instructions for performing arithmetic and logical operations, in fact all these operations can be performed by combining the bitwise operators and zero-testing in various ways.

For example, here is how to multiply two arbitrary integers a and b (a greater than b) using only bitshifts and addition:

c := 0

**while** b != 0

**if** (b **and** 1) != 0

c := c + a

shift a left by one

shift b right by one

**return** c

This implementation of [ancient Egyptian multiplication](http://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication), like most [multiplication algorithms](http://en.wikipedia.org/wiki/Multiplication_algorithm), involves bitshifts. In turn, even addition can be written using just bitshifts and zero-testing:

c := b **and** a

**while** c != 0

c := b **and** a

b := b **xor** a

shift c left by one

a := c

**return** b

**Real world Applications:**

* **Bit fields (flags)**  
  They're the most efficient way of representing something whose state is defined by several "yes or no" properties. ACLs are a good example; if you have let's say 4 discrete permissions (read, write, execute, change policy), it's better to store this in 1 byte rather than waste 4. These can be mapped to enumeration types in many languages for added convenience.
* **Communication over ports/sockets**  
  Always involves checksums, parity, stop bits, flow control algorithms, and so on, which usually depend on the logic values of individual bytes as opposed to numeric values, since the medium may only be capable of transmitting one bit at a time.
* **Compression, Encryption**  
  Both of these are heavily dependent on bitwise algorithms. Look at the [deflate](http://en.wikipedia.org/wiki/DEFLATE) algorithm for an example - everything is in bits, not bytes.
* **Finite State Machines**  
  I'm speaking primarily of the kind embedded in some piece of hardware, although they can be found in software too. These are combinatorial in nature - they might literally be getting "compiled" down to a bunch of logic gates, so they have to be expressed as AND, OR, NOT, etc.
* **Graphics** There's hardly enough space here to get into every area where these operators are used in graphics programming. XOR (or ^) is particularly interesting here because applying the same input a second time will undo the first. Older GUIs used to rely on this for selection highlighting and other overlays, in order to eliminate the need for costly redraws. They're still useful in slow graphics protocols (i.e. remote desktop).

**Pseudocode Examples:**

Low-level programming. You may need to write a specific bit to a memory-mapped register to make some piece of hardware do what you want it to:

volatile uint32\_t \*register = (volatile uint32\_t \*)0x87000000;

uint32\_t          value;

uint32\_t          set\_bit   = 0x00010000;

uint32\_t          clear\_bit = 0x00001000;

value = \*register;            // get current value from the register

value = value & ~clear\_bit;   // clear a bit

value = value | set\_bit;      // set a bit

\*register = value;            // write it back to the register

Also, htonl() and htons() are implemented using the & and | operators (on machines whose endianness doesn't match network order).

htons() and htonl() are POSIX functions to swap a short or a long from the host (h) endianness to the network (n) byte order.

#define htons(a) ((((a) & 0xff00) >> 8) | \

                  (((a) & 0x00ff) << 8))

#define htonl(a) ((((a) & 0xff000000) >> 24) | \

                  (((a) & 0x00ff0000) >>  8) | \

                  (((a) & 0x0000ff00) <<  8) | \

                  (((a) & 0x000000ff) << 24))

Instead of using a bunch of boolean flags, you can store them all in an int.

and get them out using bitwise-AND. For example:

int flags;

if (flags & 0x10) {

  // Turn a feature on.

}

if (flags & 0x08) {

  // Turn a second feature on.

}

This is an example to read colours from a bitmap image in byte format

byte imagePixel = 0xCCDDEE; /\* Image in RRGGBB format R=Red, G=Green, B=Blue \*/

//To only have red

byte redColour = imagePixel & 0xFF0000; /\*Bitmasking with AND operator \*/

//Now, we only want red colour

redColour = (redColour >> 24) & 0xFF;  /\* This now

An efficient way to multiply and divide.

2 << 3 == 2 \* 8

32 >> 4 == 32 / 16

Sending secure messages from one installation to another.

    public enum MemoView :int

    {

        InboundMemos = 1,                   //     0000 0001

        InboundMemosForMyOrders = 3,        //     0000 0011

        SentMemosAll = 16,                  //     0001 0000

        SentMemosNotReceived = 48,          //     0011

        SentMemosReceivedNotRead = 80,      //     0101

        SentMemosRead = 144,                //     1001

        Outbox = 272,                       //0001 0001 0000

        OutBoxErrors = 784                  //0011 0001 0000

    }