ECE 241

Logical Operations and How to used them:

Processors are capable of a variety of operations on data. Previous classes have dealt with addition and subtraction. There is a set of more basic operations that prove to be extremely useful. These are the logical operators of and, or, exclusive or and inversion.

These operators typically operate on data in a bit-wise fashion and so are defined based on two input bits. The following table shows the form of these operations.

A	B	A&B (and)	A|B (or)	A^B (xor)	~A (inversion)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Consider the output A&B, which is a one, if A and B are both one. Hence the name "and". Similarly the output of A|B is one if A or B are one or both are one. The exclusive or (A^B) is one if A or B is a one if A or B is a one, but not both. Finally the inversion (~A) is a one if A is zero and zero if A is one.

The above definitions are using two input bits. When using these on the computer (Arduino), the operations are done on a variable in a bit wise fashion. In other words each of the bits in one is operated by a similar bit it a second number. For example, in C programming the "anding" for two variables will look like

Result = InputA & InputB;

Now assume InputA = 0x3c and InputB = 0x66, then the operation would look something like

```
InputA  →   0011 1100
InputB  →   0110 0110
Result  →   0010 0100
```

Now a common way to characterize the action of the AND operation is shown in the following table, where a bit in one of the inputs is zero, the output will be zero regardless of the other input. This is commonly described as forcing a bit low. In the example above the first two and last two bits of InputA are zero, thus the first and last two bits in Result are zero.

A	B	A&B (and)	
0	0	0	When A is 0, the output is forced to zero regardless of B.
0	1	0	
1	0	0	However when A is 1, the output is equal to the value of B.
1	1	1	

Another term for this type of operation is called masking. In its typical use, one of the inputs is thought of as the "mask", which will cover up most of the other number, but only works on certain bits. As a demonstration of this consider the following and operation.

```
Result = InputA & 0xEF; // Force bit 4 low.

InputA  →   1011 1100
Mask    →   1110 1111
Result  →   1010 1100
```

It can be seen that Result is the same as InputA, except that bit 4 is forced to be zero or low and all other bits in InputA are unaffected.

In the case of the OR operation, we can see in the following table how the or is used to set a bit to one.

A	B	A|B (or)	
0	0	0	When A is 0, the output is equal to the value of B.
0	1	1	
1	0	1	However when A is 1, the output is forced to one regardless of B.
1	1	1	

Write this out as a masking operation, similar to the and we have.

```
Result = InputA | 0x02; // Force bit 1 high.

InputA  →   1011 1100
Mask    →   0000 0010
Result  →   1011 1110
```

Which in this case, bit 1 is forced one or high and all other bits in InputA are unaffected.

Finally the EXCLUSIVE-OR operation can be used to invert a chosen bit.

A	B	A^B (xor)	
0	0	0	When A is 0, the output is equal to the value of B.
0	1	1	
1	0	1	However when A is 1, the output is opposite (inverse) of B.
1	1	0	

```
InputA  →   1011 1100
Mask    →   0000 0110
Result  →   1011 1010
```

Here bits 2 and 1 are inverted and all others are unaffected.

As a recap of these operations, AND can be used to force a bit low, OR can be used to force a bit high, while XOR can be used to invert (toggle) a bit.

Now when writing a program we may want to use a mask with multiple bits being changed. However if you only want to change one bit, common operation, the Arduino environment provides two functions for this called bitSet and bitClear. These functions are shown here, taken for a header file that is part of the Arduino environment.

```
#define bitSet(value, bit) ((value) |= (1UL << (bit)))
#define bitClear(value, bit) ((value) &= ~(1UL << (bit)))
```

These functions can be called to either set (force to a one) or clear (force to zero) a single bit in a variable. So for example, instead of the OR operation that we used above we would write

```
bitSet( InputA, 1 ); // Note that bit numbering starts with 0.
```

If we look at the function, we see that it uses an OR operation (|) and then creates a mask that has a one in only one bit, by starting with the number 1 (ignore the UL, which is an unnecessary detail) and then shifts it to the left (<<) over by "bit" bits.

In a similar fashion, bitClear uses an AND operation, and constructs a mask, first that has a one in the bit to be changed. But then in order to make that bit low and all other high, the masked is inverted (~).

One final note, the operators AND, OR and XOR can be used in a short-hand form of

```
InputA &=  0xEF; // Equivalent to InputA = InputA & 0xEF;

InputA |=  0x02; // Equivalent to InputA = InputA | 0x02;

InputA ^=  0x06; // Equivalent to InputA = InputA ^ 0x06;
```

Examples of Video that can help

https://www.youtube.com/watch?v=d0AwjSpNXR0

http://crasseux.com/books/ctutorial/Masks.html

http://www.c4learn.com/c-programming/c-bitwise-masking/

Programming links

https://www.youtube.com/watch?v=2NWeucMKrLI

https://www.youtube.com/watch?v=nXvy5900m3M

https://www.youtube.com/watch?v=-CpG3oATGIs

https://www.youtube.com/watch?v=rk2fK2IIiiQ