Masking, Shifting, Casting, and more

Let's review how bitwise AND can isolate bits and how shifting can get those bits where we want them.

Let's do this in 32 bits so that it fits on the page. Let's say we want the 3rd byte of a 4-byte unsigned int to be treated as an unsigned char. We will show those bits as letters and we will show bits we don't want as '*' characters. So we need 8 letters to show the bits we want

```
**** **** STUV WXYZ **** **** **** ****     The int: We want STUV WXYZ
```

Masking

So we need to get rid of the bits marked with * and bitwise AND will do that for us. We will use the value 0x00FF0000 to keep the bits we want and to clear the bits we don't want.

```
**** **** STUV WXYZ **** **** **** ****     The int: We want STUV WXYZ

0000 0000 1111 1111 0000 0000 0000 0000    This is our mask 0x00FF0000

Bitwise and on those unfortunately gives:

0000 0000 STUV WXYZ 0000 0000 0000 0000    Our number is 2^16 too big
```

So that isolated the bits we want, but the result doesn't put the bits where we want them. If we assign this to an unsigned char, we will get zero since the high order bits that don't fit will be truncated.

Shifting

We can fix the problem by shifting. Because we are storing our value in an unsigned int, the compiler uses logical right shift when we right shift. This places zeroes in the high order bits.

```
0000 0000 STUV WXYZ 0000 0000 0000 0000    Our number is 2^16 too big
```

We want all of the blue bits to go away when we shift. We right shift by 16 places to get

```
0000 0000 0000 0000 0000 0000 STUV WXYZ     Our value is right
```

When we shift, logical right shift adds the leftmost zero bits marked in color. The blue bits all fell off the right end and are gone.

Casting and Assignment

The result we have has too many bits for the container we want to put it in. We want an unsigned char and we have an unsigned int. If we cast to (unsigned char), the upper bits are truncated.

```
STUV WXYZ     Our value is right and is the right size (8 bits).
```

Assignment would also force the truncation, but our lab demands that we cast before we do so.

Alternatives

Not shown is the fact that the order of the mask and the shift is not important. It might be easier to get the data right if we shift first and then mask. Let's do an example where we grab 3 bits out of a short and treat them like a number.

```
**** **** STU* ****        The short: We want bits S, T, and U
```

First we shift 5 places to make STU the lowest order bits.

```
0000 0*** **** *STU        This value includes bits we don't need
```

Now we mask to get rid of the bits we don't want. We use the value 7 (111b) to keep the lowest three bits and to clear all of the rest of the bits.

```
0000 0*** **** *STU        This value includes bits we don't need
0000 0000 0000 0111        Our mask is 7 (0x07)
---- ---- ---- ----        We bitwise and those two values
0000 0000 0000 0STU        Our result is exactly what we want
```

We can then return that value as the value we want.

Reversing the process

You can use left shifting and bitwise OR to pack values. Be aware of the sizes of your data. You don't want to lose bits in your left shifting. Once you have the bits shifted to the right places, OR them together. You have to make sure that no leftover data is in the place where you OR the bits you want.

To selectively change some bits, we clear out the old bits and then OR in the new bits. We can get rid of bits by ANDing them with zeroes.

Let's use our int example:

```
**** **** STUV WXYZ **** **** **** ****     The int: Change STUV WXYZ
                                            to ABCD EFGH and keep all
                                            of the * bits as they are
```

We will employ our same mask.

```
0000 0000 1111 1111 0000 0000 0000 0000    This is our mask 0x00FF0000
```

But we want to clear those 8 bits and keep all of the others. So we use ~ to give us the 1's complement of our mask.

```
1111 1111 0000 0000 1111 1111 1111 1111    Our mask inverted
```

We and that to our int:

```
1111 1111 0000 0000 1111 1111 1111 1111    Our mask inverted
**** **** STUV WXYZ **** **** **** ****     Our bits
```

```
---- ---- ---- ---- ---- ---- ---- ----     Bitwise AND
**** **** 0000 0000 **** **** **** ****     Partial Result
```

All of the * bits are unchanged because they were ANDed with a 1. We take the partial result and OR it with the bits we want in the place where we cleared.

```
**** **** 0000 0000 **** **** **** ****     Partial Result
0000 0000 ABCD EFGH 0000 0000 0000 0000     The bits we want set
---- ---- ---- ---- ---- ---- ---- ----     Bitwise OR
**** **** ABCD EFGH **** **** **** ****     Final result
```

All of the * bits are unchanged because they were ORed with a 0. This example doesn't show how we got ABCD EFGH bits into place, but you should be able to work out how that could be done.