# Bit-wise operations...

# The NOT

- The **bitwise NOT**, or **complement**, is a unary operation which performs logical negation on each bit, forming the ones' complement of the given binary value.
- Digits which were 0 become 1, and vice versa.
- For example: NOT 0111 = 1000
- The bitwise NOT operator is represented as a "~" (tilde).
- This operator must not be confused with the "logical not" operator, "!" (exclamation point), which treats the entire value as a single Boolean — changing a true value to false, and vice versa.
- The "logical not" is *not* a bitwise operation.

# NOT...

- 35 = 00100011 (In Binary)
- Bitwise complement Operation of 35

$$
\begin{array}{r}
\sim 00100011 \\
\hline
11011100
\end{array}
\quad = 220 \text{ (In decimal)}
$$

# Tale of the *twisted* Tilde ~

- The bitwise complement of 35 (~35) is -36 and not 220 contrary to what we have just seen.
- For any integer n, the bitwise complement of n will be -(n+1).
- To comprehend this, we need to look into the concept of 2's complement.

# 2's complement

11011100

Find the 2's complement of the number

```
 00100011
+       1
------------
 00100100      = -36
```

## ~ NOT...

```c
#include <stdio.h>
int main()
{
    printf("Output = %d\n",~35);
    printf("Output = %d\n",~-12);
    return 0;
}
```

Output = -36 Output = 11

# Bitwise Shifts

- There are two bitwise shift operators, namely
- Shift left      <<
- Shift right      >>

- Shift bits to the left or to the right.
- The syntax for a shift operation is as follows:

**[integer] [operator] [number of places];**

- A statement of this form *shifts the bits in **[integer]** by the number of places indicated, in the* ***direction specified by the operator***.

# Bitwise Shifts

Prior to shifting: 0000 0110 1001 0011

`x = x << 1;`

Post shifting: 0000 1101 0010 0110

- The MSB of x is lost, because there isn't another place to shift it to.
- Similarly, after a shift left, the LSB of x will always be 0 since there is no position to the right of the LSB, and so there's nothing to shift into the LSB.

# Bitwise shift

Prior to shifting: 0110 1111 1001 0001

`x = x >> 4;`

Post shifting: 0000 0110 1111 1001

# Shifting...

```c
#include <stdio.h>
int main()
{
     int num=212, i;
     for (i=0; i<=2; ++i)
         printf("Right shift by %d: %d\n", i, num>>i);
      printf("\n");
      for (i=0; i<=2; ++i)
         printf("Left shift by %d: %d\n", i, num<<i);

      return 0;
}
```

**Right Shift by 0: 212**
**Right Shift by 1: 106**
**Right Shift by 2: 53**

**Left Shift by 0: 212**
**Left Shift by 1: 424**
**Left Shift by 2: 848**

# Bitwise Operators: So what's the big deal?

| F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 |
|---|---|---|---|---|---|---|---|

Bitwise operators have two main applications.

Combine several values into a single variable.

Suppose you have an application where there are series of flag variables which will always have only one of two values: 0 or 1.

Would it be good to allocate each flag a byte when each of them require only a bit (0/1)?

Bitwise operators allows you to combine data in this way .

The second application for bitwise operators is that you can use them to accomplish certain arithmetic operations.

# Arithmetic: Fast multiplication

- Multiplying by *any* power of two means shifting left the same number of places.
- If you wish to multiply a number by 16, which is $2^4$, you need simply shift it 4 places leftward.
- The following pairs of statements are all equivalent to one another:

```
x = y * 8;       x = y << 3;
x = y * 64;      x = y << 6;
x = y * 32768;   x = y << 15;
```

# Arithmetic: Fast division

- If shift left is equivalent to multiplication by two, shift right equivalent to division by 2
- Note that this is integer division only;
- No fractional value or a remainder can be obtained.
- Division in hardware is not the fastest thing in the world, though it is getting better, so this is definitely a fast method (with compromises made).
- The following pairs of statements are equivalent to one another:

```
x = y / 4;        x = y >> 2;
x = y / 32;       x = y >> 5;
```

# Applications

- Bitwise operators are used in Digital Signal Processing (DSP)
- Compression and encryption algorithms
- GUIs and other application domains as an efficient way of manipulating strings of bits.

# Manipulating Colour

- In some 32-bit colour systems, the colour is represented as four distinct values.
- The low byte (bits 0 through 7) is the value for blue. The next most significant byte is a value for green, then a byte for red, and finally, the high byte is an alpha (luminance) value.

So the color dword looks like this in memory:

AAAA AAAA RRRR RRRR GGGG GGGG BBBB BBBB

AAAA AAAA RRRR RRRR GGGG GGGG BBBB BBBB

- What would you do if you wish to ascertain the value for green i.e. GGGG GGGG .

```
  AAAA AAAA RRRR RRRR GGGG GGGG BBBB BBBB
& 0000 0000 0000 0000 1111 1111 0000 0000
------------------------------------------
  0000 0000 0000 0000 GGGG GGGG 0000 0000
```

Mask Pattern for Green

# Oops: How do I interpret this no.?

0000 0000 0000 0000 GGGG GGGG 0000 0000

How do I interpret this no.?

Just shift it right by 8 bits and lo and behold we have the value of green!

Previous:

0000 0000 0000 0000 GGGG GGGG 0000 0000

Shift: >> 8

---------------------------------------------------------

Result: 0000 0000 0000 0000 0000 0000 GGGG GGGG

= GGGG GGGG

# Inserting/Combining

AAAA AAAA RRRR RRRR oooo oooo BBBB BBBB

| oooo oooo oooo oooo GGGG GGGG oooo oooo ← Mask

AAAA AAAA RRRR RRRR GGGG GGGG BBBB BBBB

But how do we create such a mask?

# Manufacturing a mask

0000 0000 0000 0000 0000 0000 GGGG GGGG

Shift: << 8

------------------------------------------------------------------------

Result:

0000 0000 0000 0000 GGGG GGGG 0000 0000

Here is the mask for previous slide

# Lessons Learned

- Bitwise operations are extremely fast for the processor to handle,
- Nice and quick,
- A good way to rid the use of a temporary variable