Unsigned Binary Representation of Numbers

Integers in the computer are stored in binary representation called "Two's Complement", but for the purposes of this class, we'll only deal with unsigned integers.

A typical integer value in a computer is stored in 32 bits, or 32 on-off switches. This representation is known as binary, 0 for off, 1 for on. In an unsigned representation of binary, known as base 2, each bit has a value of 2^k, where k represents the number of bits from the right end of the number.

For example, consider the binary number 101110:

Its decimal (base 10) value is

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 46.$$

This is how to convert from base 2 (unsigned binary) to base 10 (our regular base, decimal), we multiply each bit set to 1 by its corresponding value 2^k based on the location of the bit and then add these.

To do the opposite (converting from decimal to binary), we repeatedly divide our number by 2 keeping track of each remainder as follows until the quotient is 0:

2 | 46	
2 | 23	R 0
2 | 11	R 1
2 | 5	R 1
2 | 2	R 1
2 | 1	R 0
2 | 0	R 1

Then, just read the remainders in reverse order from bottom to top to get the binary equivalent of 46: 101110.

Bitwise Operators

In logic, we perform and, or and xor operations between Boolean variables. We can view a single bit as a Boolean value as well 0 = false, 1 = true. Thus, a binary integer can be viewed as a sequence of Boolean variables.

Naturally, we can define and, or and xor between integers by applying the binary operation to each corresponding set of bits between two integers. The following bitwise operations are fairly consistent between various computer programming langauges.

Using the technique shown above, we can obtain the binary representation of 47:

00101111

Also, 72 in binary is

01001000.

Bitwise operators take each corresponding bit in the two input numbers and calculate the output of the same operation on each set of bits.

For example, a bitwise AND is represented with a single ampersand sign: &. This operation is carried out by taking the and of two bits. If both bits are one, the answer is one. Otherwise, the answer is zero.

Here is the bitwise and operation on 47 and 72:

```
  0 0 1 0 1 1 1 1
& 0 1 0 0 1 0 0 0
--------------------
  0 0 0 0 1 0 0 0 (which has a value of 8.)
```

Thus, we only set the bit in the result to 1 if both bits in that corresponding column are one. By default, a bit that doesn't exist in one of the inputs is 0.

Here is a chart of the other bitwise operators that correspond to their logical namesakes:

Function	**Operator**	**Meaning**
and	**&**	**1&1 = 1, rest = 0**
or	**|**	**0 | 0 = 0, rest = 1**
xor	**^**	**1 ^ 0 = 0 ^ 1 = 1** **0 ^ 0 = 1 ^ 1 = 0**

Now, let's calculate the other bitwise operations between 47 and 72:

```
   0 0 1 0 1 1 1 1
|  0 1 0 0 1 0 0 0
--------------------
   0 1 1 0 1 1 1 1 (which has a value of 111.)

   0 0 1 0 1 1 1 1
^  0 1 0 0 1 0 0 0
--------------------
   0 1 1 0 0 1 1 1 (which has a value of 103.)
```

Left and Right Shift Operators

The left-shift operator is <<.
The right-shift operator is >>.

When we left-shift a value, we must specify how many bits to left-shift it. What a left-shift does is move each bit in the number to the left a certain number of places. In essence, so long as there is no overflow, a left-shift of one bit multiplies a number by two (since each bit will be worth twice as much).

It follows that a left shift of 2 bits multiplies a number by 4 and a left shift of 3 bits multiples a number by 8. In general, a left-shift of k bits multiplies a number by 2^k.

A right-shift of 1 bit moves every bit over to the right by 1 and discards the least significant bit. Effectively, this does an integer division on the original number by 2. In general, a right-shift of k bits performs an integer division by 2^k on a number. An integer division is simply taking the real number division and throwing away the part after the decimal point.

Here are a couple examples:

13 << 2 is equal to 52
47 >> 3 is equal to 5

To see this in bits, 13 is 1101. When it's left-shifted by 2 bits it becomes 110100, which when converted back to decimal is 52.

47 is 101111. When it's right-shifted by three bits, we just chop off the last three bits to get 101, which is 5, when converted back to decimal.