Advanced bit manipulation-fu

There are lots of things I think a good programmer should understand, and understand well. One, of course, is that OOP amounts to BSE for programmers. Another, and the topic of today, is that good programmers should know how to manipulate bits in their sleep. (No, not those bits!)

It goes without saying that in order even to be called a programmer one must know the truth tables of the basic operations of Boolean algebra (**and**, **or**, **xor**, and **negation** in particular) backwards and forwards. What I'm thinking of here though is that programmers should be able to do bitwise arithmetic on two's complement numbers as if their lives depended on it. (Which it does, for e.g. graphics programmers, as low-level twiddling and other clever optimization is their livelihood.)

Basic bit manipulations

Before moving on to some more advanced stuff, let's start out with the very basics of bit manipulation. Given a binary 2's-complement number **x**, we have the following useful identities:

Identity	Example	Identity and explanation
x	00101100	x, the original value
~x	11010011	~x, complement of x
-x	11010100	-x = ~x+1, negation of x (this you MUST know about 2's-complement arithmetic!)
x & -x	00000100	x & -x, extract lowest bit set
x | -x	11111100	x | -x, mask for bits above (AND including) lowest bit set
x ^ -x	11111000	x ^ -x, mask for bits above (NOT including) lowest bit set
x & (x – 1)	00101000	x & (x – 1), strip off lowest bit set

x | (x – 1)	00101111	x | (x – 1), fill in all bits below lowest bit set
x ^ (x – 1)	00000111	x ^ (x – 1) = ~x ^ -x, mask for bits below (AND including) lowest bit set
~x & (x – 1)	00000011	~x & (x – 1) = ~(x | -x) = (x & -x) – 1, mask for bits below (NOT including) lowest bit set
x | (x + 1)	00101101	x | (x + 1), fills in lowest zero bit
x / (x & -x)	00001011	x / (x & -x), shift number right so lowest bit set ends up at bit 0

If you're not used to twiddling bits these identities may not seem immediately obvious, but a few minutes with pen and paper and you should be able to see how any one of them works. Several of these should be committed to memory, including, at the very least, how to extract or strip off the lowest bit set.

From the above identities we can derive some truly fabulous expressions, like e.g. **-~x** for **x + 1**, and **~-x** for **x – 1**! (These are cool in a geeky way, but please don't use these expressions in your code, unless you're writing an entry for IOCCC.) In addition to geeking out, we can also do some really useful stuff with these identities, as I will show next.

A problem requiring advanced bit-fu

Once you know a large number of the identities above (and you should), you are equipped to solve problems that would otherwise be nontrivial. Let us consider the interesting problem of creating a function **`uint NextKBitNumber(uint)`** which given an unsigned integer with K bits set, returns the immediately larger unsigned integer also with K bits set.

"Huh," you say? Well, as always, the first thing to do is to understand the problem, so let's study some data. Let's start with a 5-bit number, with 2 bits set. From combinatorics we know there should be (5 choose 2) = 10 alternatives. We can easily list them by hand:

```
0 0 0 1|1
0 0 1|0 1
0 0 1|1 0
```

```
0 1|0 0 1
0 1|0 1 0
0 1|1 0 0
1|0 0 0 1
1|0 0 1 0
1|0 1 0 0
1|1 0 0 0
```

As you can see, I inserted a vertical bar on each line to highlight an obvious recurring pattern: the lower bit travels from right to left until it becomes adjacent with the upper bit. At that point we bump the upper bit one position to the left, and reset the lower bit back to the rightmost position. Let's verify this pattern for three bits set. Here we have 7-bit numbers, with 3 bits set:

```
0 0 0 0 1|1|1
0 0 0 1|0 1|1
0 0 0 1|1|0 1
0 0 0 1|1|1 0
0 0 1|0 0 1|1
0 0 1|0 1|0 1
0 0 1|0 1|1 0
0 0 1|1|0 0 1
0 0 1|1|0 1 0
0 0 1|1|1 0 0
0 1|0 0 0 1|1
0 1|0 0 1|0 1
0 1|0 0 1|1 0
0 1|0 1|0 0 1
0 1|0 1|0 1 0
0 1|0 1|1 0 0
0 1|1|0 0 0 1
0 1|1|0 0 1 0
0 1|1|0 1 0 0
0 1|1|1 0 0 0
1|0 0 0 0 1|1
1|0 0 0 1|0 1
1|0 0 0 1|1 0
1|0 0 1|0 0 1
1|0 0 1|0 1 0
```

```
1|0 0 1|1 0 0
1|0 1|0 0 0 1
1|0 1|0 0 1 0
1|0 1|0 1 0 0
1|0 1|1 0 0 0
1|1|0 0 0 0 1
1|1|0 0 0 1 0
1|1|0 0 1 0 0
1|1|0 1 0 0 0
1|1|1 0 0 0 0
```

OK, so the low bit still travels from right to left and gets bumped back to bit 0 after it becomes adjacent with another bit, but there's clearly more going on here: sometimes we have two bits bumped back. Studying the bit patterns for a while should make it apparent that what happens (conceptually) is that once the lowest bit has become adjacent with a sequence of one or more bits it "jumps" those bits to the next higher zero bit and all the bits that were jumped are bumped back to bit 0.

Followed that? Good! Let's see how we can implement this logic using the bit manipulations from above.

First attempt

We can accomplish the left-shifting of the lowest bit by extracting the lowest bit set (using **b = x & -x**) and adding this onto the original number (**t = x + b**). This also takes care of the "jumping over" part, but in doing so it clears out all the bits we jumped past so we need to add those back in. We can create those bits by taking the lowest bit in **t** (using **c = t & -t**), dividing it by **b** to shift it down, and then creating the appropriate mask **m = (c / b >> 1) – 1** by shifting by two (as we counted one bit too many) and subtracting one (to form the mask). The final result is then **r = t | m**. That was a mouthful, so let's look at an example:

Operation	**Example**	**Operation and explanation**
x	01011100	x, the original value
b = x & -x	00000100	b = x & -x, extract lowest set bit b in x
t = x + b	01100000	t = x + b, shift lowest set bit to the left, here causing "jump" past two adjacent one bits

c = t & -t	00100000	c = t & -t, extract lowest set bit c in t
c / b	00001000	c / b, shifts c down by factor of b
m = (c / b >> 1) – 1	00000011	m = (c / b >> 1) – 1, form bitmask for the (two) bits we jumped past
r = t | m	01100011	r = t | m, form final result

That took 9 operations. Now that we know how to do it, can we do better? Let's see.

Second attempt

It should be clear that we have to perform the division to shift **c** down to form the mask for the bits we jumped past, but can we perhaps somehow avoid the shift and subtract part in forming the mask? Let's look at a different way of computing **c** and **m**. Going back to the table at the top of the article, we see that we can try **c = t ^ (t – 1)** instead. That forms all bits below the lowest bit set in **t** so after dividing and shifting, we no longer need to subtract one in forming **m**. (But we do have to shift by two instead of by one after making this change.) Here's the updated table of operations:

Operation	**Example**	**Operation and explanation**
x	01011100	x, the original value
b = x & -x	00000100	b = x & -x, extract lowest set bit b in x
t = x + b	01100000	t = x + b, shift lowest set bit to the left, here causing "jump" past two adjacent one bits
c = t ^ (t – 1)	00111111	c = t ^ (t – 1), mask for bits below (AND including) lowest bit set in t
m = (c >> 2) / b	00000011	m = (c >> 2) / b, form bitmask for the (two) bits we jumped past
r = t | m	01100011	r = t | m, form final result

Now we're down to 8 operations, can we do better?

Third and final attempt

Turns out we **can** do better! We don't really need to create one bits all the way down to bit 0 as we did with **c = t ^ (t – 1)** in the previous attempt. We only need to create enough one bits so that after we shift them down to reside at bit position 0, we have the same number of bits as we jumped past. This realization allows us to use **c = x ^ t** instead, which shaves off yet another operation, resulting in the final code:

Operation	Example	Operation and explanation
x	01011100	x, the original value
b = x & -x	00000100	b = x & -x, extract lowest set bit b in x
t = x + b	01100000	t = x + b, shift lowest set bit to the left, here causing "jump" past two adjacent one bits
c = x ^ t	00111100	c = x ^ t, form "sufficiently long" sequence of one bits, for shifting down
m = (c >> 2) / b	00000011	m = (c >> 2) / b, form bitmask for the (two) bits we jumped past
r = t | m	01100011	r = t | m, form final result

That's 7 operations. Can we do better? I don't think so, but feel free, go ahead, prove me wrong!

Improving your bit-fu

The above problem was just a brief taste of the nifty stuff you can do once you've truly grokked bit manipulation. And if you think "bit tricks" are just esoterica, think again. For example, the vector units on the PlayStation 2 do not have an **xor** instruction. In the traditional discrete math, logic, or abstract algebra class you probably learned that (A xor B) = ((not(A) and B) or (A and not(B)) so you could synthesize xor in 5 instructions. Ah, but no! The experienced bit manipulator knows that you can do it in 3 instructions as (A xor B) = ((A or B) – (A and B)). (That's the kind of stuff you pull to get PS2 VU code fast enough to do games like God of War and God of War 2.)

You can also find lots of applications of these sort of "tricks" in my **book**, used to optimize e.g. spatial partitioning methods to make them industrial strength (for collision detection, visibility determination, and other applications).

If you want to become one with the bit, other than lots of practical workouts in the bit-gym, the best printed resource is unarguably **Henry Warren Jr's Hacker's Delight** which I highly recommend (also listed on my recommended list). BTW, the above problem is discussed on page 14 of Hacker's Delight (though not quite in this much detail.)