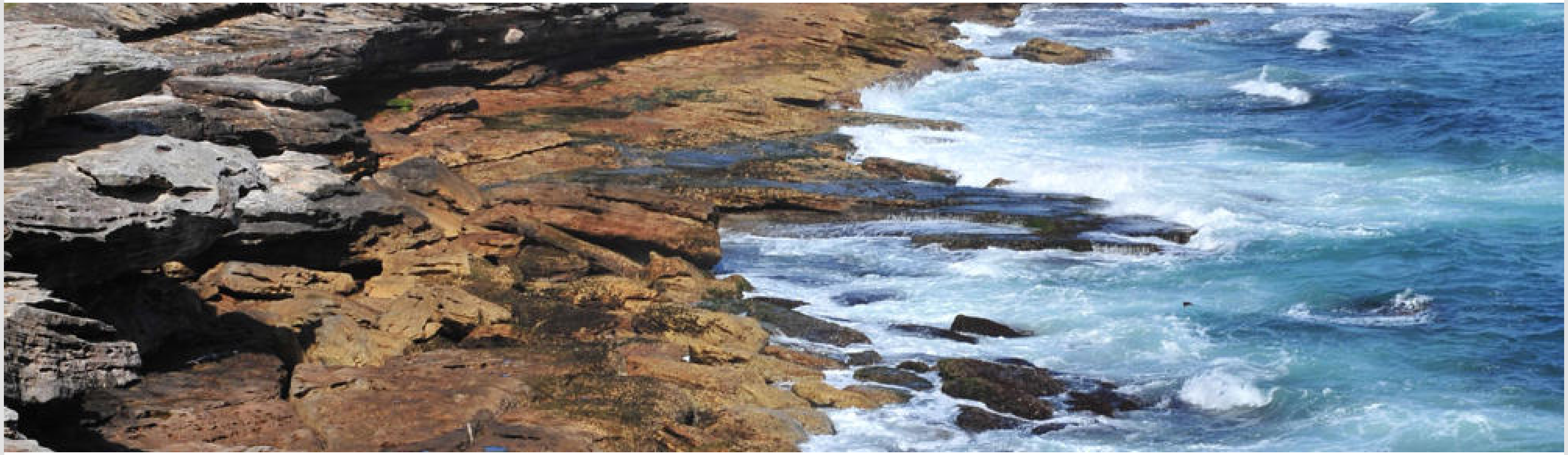
CS 207 Blog

Search

Lecture notes and thoughts on software engineering, C++, and computational science



Home

Posted on **March 1, 2012**

← Previous

Lecture 12: Bits

This was a Friday lecture, and Friday lectures are often less formal than other days, so we offered the class a choice of several topics. The overwhelming answer was "bits."

Question: What does this do?

```
x;
y = x & -x;
c = x + y;
x = (((x ^ c) >> 2) / y) | c;
```

Wait, ampersand? And plus? *And* exclusive or? **And** right-shift? And **divide**?? With a bitwise-or chaser?!

Huh. First, we can tell that `x` is something like a computer integer type, since it is subject to a bunch of bitwise operators. Here they are. (Bitwise operations most often use unsigned integers and integers known not to be negative. With negative numbers, some operations, like the shifts, have undefined results.)

- `a & b` is **bitwise and**. The result has a 1 bit in position *i* if and only if both `a` and `b` have 1s in that position.
- `a | b` is **bitwise or**. The result has a 1 bit in position *i* if and only if *either* `a` or `b` has a 1 in that position.
- `a ^ b` is **bitwise exclusive or (xor)**. The result has a 1 bit in position *i* if and only if one of `a` and `b` has a 1 in that position, and the other doesn't.
- `a >> x` is **bitwise right shift**. The result equals $a / 2^x$—the `x` lowest-order bits of `a` are removed, and 0 bits are shifted in at the higher-order positions—but is usually faster to compute.
- `a << x` is **bitwise left shift**. The result equals $a \times 2^x$, but is usually faster to compute.
- `~a` is **bitwise complement**. The result has 1 bits everywhere `a` has 0 bits, and vice versa.

Returning to the example, then, what is `x & -x`?

Negative

First, what's the bitwise significance of `-x`, if `x` is a 64-bit unsigned integer? Well, `-x` is the unique `y` with `x + y == 0`. And that is 2^{64}-`x`, for then `x + y ==` 2^{64}, which overflows and produces 0. For instance, `-2 == 18446744073709551614` on a 64-bit machine, since `2 + 18446744073709551614 == 18446744073709551616 == 2`^{64}` == 0` on a 64-bit machine.

There's another cool fact about `-x`, namely that `-x == ~x + 1`: we negate `x` by flipping all its bits and adding 1. Why is this? Let's work with a simple example, namely 5.

```
      5 == 0 0 0 0 0 1 0 1  (8 bits for simplicity)
```

If we flip all the bits, we get ~5, which (in 8 bits) equals 250:

```
     ~5 == 1 1 1 1 1 0 1 0
```

If we add `x + ~x`, the result will *always* have all 1s. (`~x` has a 1 every place `x` has a 0, and vice versa; so every bitwise addition adds `0 + 1` to get `1`, and there are no carries to cause ripple effects.)

```
      5 == 0 0 0 0 0 1 0 1
   + ~5 == 1 1 1 1 1 0 1 0
  ------------------------
           1 1 1 1 1 1 1 1
```

Thus, `x + ~x` is always the largest representable integer of `x`'s type. Add `1` to that number and you get `0`: the addition in the ones place causes a carry that wipes out all the higher `1`s!

```
     5 == 0 0 0 0 0 1 0 1
 + ~5 == 1 1 1 1 1 0 1 0
------------------------
          1 1 1 1 1 1 1 1
 +   1 == 0 0 0 0 0 0 0 1
------------------------
       (1)0 0 0 0 0 0 0 0
```

(Neat fact: If we use the same circuitry in the processor for adding signed and unsigned numbers, then `-2` in signed arithmetic must have the same bit pattern as `18446744073709551614`. This representation is called *two's complement arithmetic* and it's used on all the computers you're likely to program.)

Lowest one bit

But what is `(x & -x) == (x & (~x + 1))`? First some examples, then some reasoning.

```
     1 == 0 0 0 0 0 0 0 1
 & -1 == 1 1 1 1 1 1 1 1
```

```
  ------------------------
        1 == 0 0 0 0 0 0 0 1

        6 == 0 0 0 0 0 1 1 0
   & -6 == 1 1 1 1 1 0 1 0
  ------------------------
        2 == 0 0 0 0 0 0 1 0

       16 == 0 0 0 1 0 0 0 0
  & -16 == 1 1 1 1 0 0 0 0
  ------------------------
       16 == 0 0 0 1 0 0 0 0
```

Every combination is `≤ x`, and every combination also equals a power of two (has exactly one `1` bit).

In fact, `x & -x` equals the *least significant one bit of* `x` (or `0` if `x == 0`). To see why, take it apart. `x & ~x == 0` for all `x`, since `~x` has 1 bits only where `x` doesn't. But we are actually calculating `x & (~x + 1)`. Adding `1` flips at least one of the bits of `~x`. In fact, it might flip more than one through carries, which ripple up. Assume that `x` has the form "`A...Z10`^{n}": some number of arbitrary bits (`0` or `1`), followed by a `1`, followed by `n` `0` bits, for `n≥0`. Then

```
     x ==  A  B  C ...  X  Y  Z  1  0 ... 0   (n≥0 t
    ~x == ~A ~B ~C ... ~X ~Y ~Z  0  1 ... 1
```

```
  ~x + 1 == ~A ~B ~C ... ~X ~Y ~Z  1  0 ... 0
```

And

```
       x ==  A  B  C ...  X  Y  Z  1  0 ... 0
  &   -x == ~A ~B ~C ... ~X ~Y ~Z  1  0 ... 0
  -------------------------------------------
             0  0  0 ...  0  0  0  1  0 ... 0
```

`(x & -x) == “0...010`^{n}`”` is exactly the least significant one bit of `x`.

(A related trick: `x` is a power of 2 or zero if and only if `(x & (x - 1)) == 0`.)

Continuing on

What is `c == x + (x & -x)`? Well, since `x & -x` is the least significant one bit of `x`, we know that the addition will flip that bit back to `0`. For example, take `x == 5` or `22`:

```
             5 == 0 0 0 0 0 1 0 1
  +    (5 & -5) == 0 0 0 0 0 0 0 1
  -------------------------------
             6 == 0 0 0 0 0 1 1 0
             22 == 0 0 0 1 0 1 1 0
+  (22 & -22) == 0 0 0 0 0 0 1 0
-------------------------------
             24 == 0 0 0 1 1 0 0 0
```

As with the `+ 1` in `~x + 1` above, the single added bit ripples up through a series of `1` bits via carries. The ripple stops when the carry process reaches a `0` bit, which it flips to `1`. In notation, let's write `x == "A...Z01`^{m}`0`^{n}`"`: `x` is some arbitrary bits, followed by a `0`, followed by `m≥1 1` bits, followed by `n≥0 0` bits. Then `c` is:

```
          x ==  A ...  Z  0  ......1•m......  ...0•n
+   (x & -x) ==  0 ...  0  0  ...0•(m-1)... 1  ...0•n
-----------------------------------------------------
          c ==  A ...  Z  1  .........0•(m+n).......
```

Completing the hack

One step at a time:

```
           x ==  A ...  Z  0  ......1•m......  ...0•r
^          c ==  A ...  Z  1  .........0•(m+n)......
-----------------------------------------------------
     (x ^ c) ==  0 ...  0  1  ......1•m......  ...0•r
```

```
                 ==   0 ...   0   .....1•(m+1)......   ...0•r
```

Note how the exclusive or got rid of all the higher-order `A...Z` bits.

Now shift right by two (which might actually shift off one or two of the `1` bits—in our notation we assume `0•-1` eats up bits to the left):

```
  (x ^ c) >> 2 ==   0 ...   0   ......1•(m+1)......   .0•(r
```

Now, the divide. But wait a minute: we are dividing by `y == (x & -x)`, which has a special form: we know it is a power of two! Specifically, `y == 2`^{n} `== “10`^{n}`”`. And dividing by a power of two has another meaning in binary arithmetic: it is the same as a *right shift*. To calculate `((x ^ c) >> 2) / y`, we simply shift `(x ^ c) >> 2` right by `n` places! This eats up all our right-hand `0` bits and two of our `1`s.

```
     ((x ^ c) >> 2) / y ==   0 ........   0   ....1•(m-1)
```

Finally, we bitwise-or this with `c`:

```
     ((x ^ c) >> 2) / y ==   0 ...   0   0   0 ...... 0   ..
  |                    c ==   A ...   Z   1   .........0•(r
------------------------------------------------------
               result ==  A ...  Z  1  ..0•(n+1).. ..

 (Reminder:          x ==  A ...  Z  0  .....1•m.....
```

What are some things you notice about this result?

- The high-order bits are exactly the same as `x`'s.
- The result is greater than `x`. It has a `1` bit immediately after `Z`, where `x` had a `0` bit; and this is the most significant difference between them.
- The lower end of the result has exactly `m` `1` bits in it, *just like* `x` *did*.
- Since the upper end (`A ... Z`) has the same number of `1` bits in both numbers, *the result and* `x` *have the same number of* `1` *bits.*
- Other than the `1` bit immediately following `Z`, the rest of the result's `1` bits are packed at the least significant end of the integer. This means that *there is no* `k` where `x < k < result` and `x` and `k` have the same number of `1` bits.
- Putting it all together, the result of this procedure is *the next larger number with the same number of* `1` *bits as the input.*

Isn't this magical? The discoverer of this procedure is immortal, for this and other reasons: his name is William Gosper, and this is Gosper's hack.

Some repeated applications of Gosper's hack, which we'll write `G(x)`: (Changed bits are highlighted.)

```
                    1 == 0 0 0 0 0 0 0 1     (m=1, n=0)
      G(1)     ==   2 == 0 0 0 0 0 0 1 0     (m=1, n=1)
    G(G(1))    ==   4 == 0 0 0 0 0 1 0 0     (m=1, n=2)
  G(G(G(1)))   ==   8 == 0 0 0 0 1 0 0 0     (m=1, n=3)

                    5 == 0 0 0 0 0 1 0 1     (m=1, n=0)
      G(5)     ==   6 == 0 0 0 0 0 1 1 0     (m=2, n=1)
    G(G(5))    ==   9 == 0 0 0 0 1 0 0 1     (m=1, n=0)
  G(G(G(5)))   ==  10 == 0 0 0 0 1 0 1 0     (m=1, n=1)

                   30 == 0 0 0 1 1 1 1 0     (m=4, n=2)
     G(30)     ==  39 == 0 0 1 0 0 1 1 1     (m=3, n=0)
   G(G(30))    ==  43 == 0 0 1 0 1 0 1 1     (m=2, n=0)
  G(G(G(30)))  ==  45 == 0 0 1 0 1 1 0 1     (m=1, n=0)
G(G(G(G(30))))==  46 == 0 0 1 0 1 1 1 0     (m=3, n=1)
   G^5(30)     ==  51 == 0 0 1 1 0 0 1 1     (m=2, n=0)
```

Gosper's hack and subset enumeration

So what? Who cares about numbers with the same count of `1` bits?

The answer is that this hack has great applications, once we think of computer integers not just as numbers, but as compact, convenient representations for vectors of bits.

Take the subset enumerator from last time. We used a 64-bit integer “subset chooser” to cycle through all the subsets of a set of ≤63 elements, represented as an iterator range. The *i*th element was in the subset if and only if the *i*th bit of the chooser was 1.

When plugged in to the subset sum problem, a subset chooser using normal arithmetic will find the *lexically lowest subset* that solves the problem. But what if we wanted to find one of the *smallest* subsets that solved the problem? Consider the following range of elements:

```
[1, 1, 1, 1, 1, 1, 1, 1, -8, 8]
```

With a normal subset chooser, the subset sum code will find the subset `{1, 1, 1, 1, 1, 1, 1, 1, -8}`. But a much smaller subset also solves the problem: `{-8, 8}`. What if we wanted to find that subset before finding the larger one?

Simple: Use Gosper’s hack. A subset chooser can enumerate through all the subsets of size *k* through repeated application of `G`. When all these subsets are enumerated, it can switch to the next largest subset size. This has exactly the same complexity as a normal subset chooser, since Gosper's hack is O(1). And even more wonderfully, using C++ iterators it is possible to encapsulate all of this in a clean "subset chooser" type, allowing us to switch choosing methods however we want—or come up with our own. Although our current choosers work only on sets of size ≤63, we could also apply the hack to *arrays* of 64-bit integers, and hide that detail underneath a clean subset chooser interface.

A "subset chooser by size" must recognize when all the subsets of the current size have been visited, and then switch to the next higher size. Think about how to do this.

Gosper's hack visits subsets in numerically (lexically) increasing order, so we know we are done with the current size when we visit a subset that refers to a nonexistent element. If `size` is the size of the entire set, then let `highbit` equal `1 << size`. We would have:

```
uint64_t next_subset(uint64_t x, uint64_t highbit) {
  uint64_t y = x & -x;
  uint64_t c = x + y;
```

```
    x = (((x ^ c) >> 2) / y) | c;
    if (x & highbit)
      x = ((x & (highbit - 1)) << 2) | 3;
    return x;
  }
```

Note that we can get to the next size with bitwise operations without knowing what the current size is! If you don't understand, write out some examples in notation. The code does not work for subsets of size 0, and does not detect when to stop (namely, after the single valid subset containing `size` elements). Can you fix it?

Some aspects of this presentation were influenced by Knuth's presentation in TAOCP Volume 4A.

This entry was posted in **Lectures** by **kohler**. Bookmark the **permalink**.

Comments are closed.