TRICKS IN COMPUTER ARITHMETIC

ANDREW FOOTE

People who do a lot of mental arithmetic often make use of "tricks" to carry out calculations. An example familiar to most people is that you can multiply an integer represented by its decimal expansion by 10 by simply adding an extra 0 digit: for example, 321 times 10 is 3210. Another trick, which not so many people are familiar with, is that in order to determine whether an integer is divisible by 3 it suffices to examine the sum of the digits in its decimal expansion: the original integer is divisible by 3 if and only if this sum is. For example, 321 must be divisible by 3 because $3 + 2 + 1 = 6$, and 6 is divisible by 3. The defining characteristic of tricks like these is that they enable people to do less calculations to reach their result, reducing the time taken, the cognitive effort required and the likelihood of error, while at the same time only being applicable in a limited range of circumstances, so that they are unable to fully replace the more cumbersome but more general algorithms that are taught to us in school.

It might come as a surprise to learn that computers also make use of such tricks. Computers can achieve much greater speeds and have a greater working memory capacity than humans, and it hardly matters how much effort they have to go to calculate things; and they hardly ever make errors, provided they are making use of a correct algorithm.[1] So one might think they wouldn't need to resort to tricks. But computers often need to perform lots of arithmetic calculations in a short amount of time, and in such circumstances, any slight speed up in one of the individual operations can have an arbitrarily large effect on the speed of the whole procedure, depending on how many times the operation needs to be repeated. So it's primarily the fact that tricks increase the speed of calculation that makes them worthwhile for computers.

A big difference between computers and humans, when it comes to arithmetic, is that computers represent integers by their binary expansions, rather than their decimal expansions. But a lot of the tricks humans use can still be transferred fairly straightforwardly to the binary environment. For example, whereas adding a zero digit multiplies the value of a decimal expansion by 10, adding a zero digit multiplies the value of a binary expansion by 2. So computers are best at multiplying by 2, rather than by 10. This is actually a lot more useful—it's more often necessary to double a quantity than to multiply it by 10.

What about the trick for checking whether an integer is divisible by 3? Does that have a binary counterpart? Well, let's think about how this trick works. It's

[1]Absolutes are rarely true, of course. All else being equal, it's better for a computer to have a longer battery life, and this is facilitated by it not having to carry out too many complex operations. But this isn't a critical concern, compared to things like the capacity of the computer to do what the user wants in a reasonable amount of time, considering that its battery can always be recharged. Likewise, there is a small chance of a freak mechanical failure with every operation, so the more operations are done, the more likely such errors are; however, the base chance is still so low that this hardly ever becomes a matter of concern to users.

basically a consequence of the fact that 10 is congruent to 1 modulo 3. If we have an integer x whose decimal expansion is made up of the digits d_0, d_1, ... and d_n (in that order, from least significant to most significant), then we have the equation

$$x = d_0 + 10d_1 + \cdots + 10^n d_n = \sum_{k=0}^{n} 10^k d_k.$$

Now, if we only care about congruence modulo 3, we can replace the terms in the sum on the right-hand side with terms that are congruent modulo 3 to the original terms. We can also replace factors within those terms by factors that are congruent modulo 3 to the original factors. In particular, we can replace the 10s by 1s. Since 1 is the multiplicative identity, this allows us to eliminate the factors of 10. Therefore, we have the congruence

$$x \equiv \sum_{k=0}^{n} d_k \pmod 3.$$

That is, the value of x modulo 3 is the same as the value modulo 3 of the sum of the digits in the decimal expansion of x. This proves that the trick works, because x being divisible by 3 is equivalent to x being congruent to 0 modulo 3. In fact it gives us two more tricks: an integer is 1 plus a multiple of 3 if and only if the sum of its digits is also 1 plus a multiple of 3, and an integer is 1 minus a multiple of 3 if and only if the sum of its digits is also 1 minus a multiple of 3.

Now, if d_0, d_1, ... and d_n are binary bits rather than decimal digits, then we must start with the equation

$$x = d_0 + 2d_1 + \cdots + 2_n d^n = \sum_{k=0}^{n} 2^k d_k,$$

with the digits being multiplied by powers of 2 rather than powers of 10. The number 2 is congruent to -1, not 1, modulo 3. But this still allows us to do some simplification, since $(-1)^k$ is 1 for even integers k and -1 for odd integers k. The congruence simplifies to

$$x \equiv \sum_{k=0}^{n} (-1)^k d_k \pmod 3,$$

showing that x is congruent modulo 3 to the *alternating* sum of its bits.

A computer could calculate this alternating sum by simply iterating over the bits of x, alternating between an adding and subtracting state. However, this wouldn't be very efficient; it would require as many iterations as x has bits, which means it would likely be no quicker than simply using the division algorithm a lot of the time.

It is possible to do better. Rather than taking each bit as a unit, we can take each two-bit segment as a unit. This will eliminate the need to explicitly alternate between addition and subtraction. So, assuming n is odd (so that x has an even number of bits, including the one with place value 1), we may write

$$x \equiv \sum_{k=0}^{\frac{n-1}{2}} (d_{2k} - d_{2k+1}) \pmod 3.$$

Now here's something neat: -1 is congruent to 2 modulo 3! So we can also write this congruence as

$$x \equiv \sum_{k=0}^{\frac{n-1}{2}} (d_{2k} + 2d_{2k+1}) \pmod 3.$$

Now, for every integer k between 0 and $\frac{n-1}{2}$ inclusive, the sum $d_{2k} + 2d_{2k+1}$ is just the value of the two-bit integer whose bits are d_{2k} (least significant) and d_{2k+1} (most significant). So we can state the binary rule for divisibility by 3 as follows:

> A binary expansion has a value divisible by 3 if and only if the sum of the values of its two-bit segments, interpreted as independent binary expansions, is divisible by 3. More generally, its value is congruent to this sum modulo 3.

This rule can be straightforwardly applied in a computer program. It's just a matter of summing segments. Once we have the sum, we can use a lookup table to determine its value modulo 3, since the set of possible values of the sum will be much smaller than the set of possible values of the original integer. The summing of the segments can be done in parallel using bitwise operations in an unrolled loop. Here's an implementation in the C programming language.

```
unsigned mod3(unsigned x) {
    static unsigned TABLE[48] = {
        0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2,
        0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2,
        0, 1, 2, 0, 1, 2
    };

    /* Sum adjacent 2-bit segments in parallel. Note that 0x33333333 is
    8 repetitions of the bit sequence 0011, and 0xcccccccc is 8
    repetitions of the bit sequence 1100. */
    x = (x & 0x33333333) + ((x & 0xcccccccc) >> 2);

    /* Sum adjacent 4-bit segments in parallel. Note that 0x0f0f0f0f is
    4 repetitions of the bit sequence 00001111, and 0xf0f0f0f0 is 4
    repetitions of the bit sequence 11110000. */
    x = (x & 0x0f0f0f0f) + ((x & 0xf0f0f0f0) >> 4);

    /* Sum adjacent 8-bit segments in parallel. Note that 0xff00ff00 is
    2 repetitions of the bit sequence 0000000011111111, and 0xf0f0f0f0
    is 2 repetitions of the bit sequence 1111111100000000. */
    x = (x & 0xff00ff00) + ((x & 0x00ff00ff) >> 8);

    /* Sum the two 16-bit segments. */
    x = (x & 0x0000ffff) + ((x & 0xffff0000) >> 16);

    return TABLE[x];
}
```

Unfortunately, this implementation does not appear to actually be more efficient than a regular modulo operation. I wrote a profiler for the routine (the source code is available on GitHub at `https://github.com/Andrew-Foote/odds-and-ends/`

`blob/master/mod3.c`—note that it is not written to be portable) and ran it on Windows using Microsoft's Visual C++ compiler. I also profiled the calculation of values modulo 3 using the ordinary modulo operator for comparison. 16777216 calls to the my subroutine took about 200 milliseconds, but 16777216 ordinary modulo operations took only about 150 milliseconds.

Of course, it may be that the method could be more efficient than a regular modulo operation if my code was better. I'm not very experienced with this kind of programming.

1. Another trick

Although our trick of summing the 2-bit segments didn't pay off, we can find a trick that *does* pay off by simply taking a C program that computes values modulo 3 in the obvious way, using the modulo operator, and compiling this program with an *optimizing* compiler. An optimizing compiler will optimize whatever it can, so if there is a trick that can be used to calculate values modulo 3 more efficiently, the compiler should make use of it.

To see the assembly output from a compiler, there's no need to actually run a local compiler: Matt Godbolt's Compiler Explorer tool at `https://godbolt.org` has got you covered. It's a very neat website that lets you choose from an array of different compilers for different languages hosted on its own servers, so that you can quickly compare outputs.

Here's the code for a C function which does an ordinary modulo operation. It deals with unsigned (non-negative) integers only, to keep things maximally simple.

```
unsigned mod3(unsigned x) {
    return x % 3;
}
```

Compiling with the GNU C Compiler (GCC), version 8.3, on an x86-64 architecture and using the `-O3` flag (for maximal standards-compliant optimization), we get this assembly output:

```
mod3:
  mov eax, edi
  mov edx, -1431655765
  mul edx
  mov eax, edx
  shr eax
  lea eax, [rax+rax*2]
  sub edi, eax
  mov eax, edi
  ret
```

This clearly isn't just doing a `div`. So what's going on? In case you can't read x86-64 assembly, the assembly subroutine is effectively using the following formula[2]

[2]Careful readers might wonder whether this formula is oversimplified, since the multiplication of $\lfloor \frac{2863311531x}{2^{33}} \rfloor$ by 3 might result in overflow, in which case it would be $(3\lfloor \frac{2863311531x}{2^{33}} \rfloor) \bmod 2^{32}$ that would be subtracted from x, not the full product $3\lfloor \frac{2863311531x}{2^{33}} \rfloor$. However, this multiplication will actually never overflow. You can convince yourself of this by considering the case where x is as large as possible, i.e. $x = 2^{32} - 1$.

to compute the value modulo 3 of the argument x:

$$x \bmod 3 = x - 3\left\lfloor \frac{2863311531x}{2^{33}} \right\rfloor.$$

In general, the remainder of an integer a on division by another integer b can be calculated by subtracting the quotient yielded by the same division from a. So the assembly code is really calculating $\lfloor \frac{x}{3} \rfloor$ first, and then calculating the remainder using this value. The interesting part is the way in which it computes $\lfloor \frac{x}{3} \rfloor$. Apparently, for nonnegative integers less than 2^{32}, we have

$$\left\lfloor \frac{x}{3} \right\rfloor = \left\lfloor \frac{2863311531x}{2^{33}} \right\rfloor. \quad (1)$$

Indeed, if you replace the modulo operation in our `mod3` function with an integer division operation (and rename the function with the more appropriate name of `quo3`) you'll see the assembly output below in the Compiler Explorer:

```
quo3:
  mov eax, edi
  mov edx, -1431655765
  mul edx
  mov eax, edx
  shr eax
  ret
```

This is just equation (1) in x86 assembly language.

So, why does this equation hold? Well, let's have a look at this mysterious constant 2863311531 that turns up in it. Often, when computers appear to be using a mysterious constant, things make more sense when you look at the binary expansion of the constant. The binary expansion of 2863311531 is this:

$$10101010101010101010101010101011.$$

Aha! It's just 15 repetitions of the two-bit sequence 10, with an extra two 1 bits on the end. Another way to put it is that it's $m + 1$ where m is the integer whose binary expansion is 16 repetitions of the two-bit segment 10.

What can we do with this knowledge? Well, a repeating sequence of a number of bits or digits is nothing more than a geometric series. Let's write m as a geometric series:

$$m = \sum_{k=0}^{15} 2^{2k+1} = \sum_{k=0}^{15} 2 \cdot 2^{2k} = 2\sum_{k=0}^{16} 2 \cdot 4^k.$$

This geometric series has initial value 2, common ratio 4 and 16 terms. Therefore, it can be evaluated as the fraction

$$2\frac{4^{16} - 1}{4 - 1} = 2\frac{2^{32} - 1}{3} = \frac{2^{33} - 2}{3}.$$

Now a divisor of 3 has turned up, which is promising.

In the formula, we actually multiply x by $m + 1$, not m itself. If we add 1 to the fraction above, we get

$$\frac{2^{33} + 1}{3}.$$

Multiplying by $2^{-33}x$, this comes out as

$$\frac{x + 2^{-33}x}{3} = \frac{x}{3} + \frac{2^{-33}x}{3}.$$

Now, since $x < 2^{32}$, we have

$$\frac{2^{-33}x}{3} < \frac{1}{6}.$$

Since $\frac{x}{3}$ is an integer divided by 3, it is impossible for its floor to change when you add a real number less than 1/3 to it. Since $\frac{2^{-33}x}{3} < \frac{1}{6}$, this proves equation (1). QED.

This technique readily generalizes to even word sizes other than 32, by the way. If the word size is an arbitrary even integer n, then to compute $\lfloor \frac{x}{n} \rfloor$ we just have to calculate

$$\left\lfloor 2^{-(n+1)}x \left(\sum_{k=0}^{n/2-1} 2^{2k+1} \right) \right\rfloor.$$

What about moduli other than 3? If you play around with the Compiler Explorer, you'll see that GCC uses roughly the same sequence of operations to calculate values modulo any constant, so there is a general trick at work here. I may try to reverse-engineer it in another post soon (or I may not; I don't have a great track record of completing planned sequences of posts on this blog :))