Labor of Division (Episode III): Faster Unsigned Division by Constants

ridiculous_fish
corydoras@ridiculousfish.com
October 19th, 2011

This is a technique *fish* thought up for improving the performance of unsigned integer division by certain "uncooperative" constants. It does not seem to appear in the literature, nor is it implemented in gcc, llvm, or icc, so *fish* is optimistic that it is original.

As is well known (and seen in a previous post), compilers optimize unsigned division by constants into multiplication by a "magic number." But not all constants are created equal, and approximately 30% of divisors require magic numbers that are one bit too large, which necessitates special handling (read: are slower). Of these 30%, slightly less than half (46%) are even, which can be handled at a minimum of increased expense (see below); the remaining odd divisors (659 million, including well known celebrity divisors like 7) need a relatively expensive "fixup" after the multiplication. Or so we used to think. This post gives a variation on the usual algorithm that improves performance for these expensive divisors.

This post presents the algorithm, proves it is correct, proves that it applies in every case we care about, and demonstrates that it is faster. It also contains a reference implementation of the full "magic number" algorithm, incorporating this and all known techniques. In other words, it's so darn big that it requires a table of contents.

Background

Unsigned integer division is one of the slowest operatins on a modern microprocessor. When the divisor is known at compile time, optimizing compilers do not emit division instructions, but instead either a bit shift (for a power of 2), or a multiplication by a sort of reciprocal (for non-powers of 2). This second case involves the identity:

$$\lfloor \frac{n}{d} \rfloor = \lfloor \frac{n}{d} \times \frac{2^k}{2^k} \rfloor = \lfloor \frac{2^k}{d} \times \frac{n}{2^k} \rfloor$$

As d is not a power of 2, $\frac{2^k}{d}$ is always a fraction. It is rounded up to an integer, which is called a "magic number" because multiplying by it performs division, as if by magic. The rounding-up introduces error into the calculation, but we can reduce that error by increasing k. If k is big enough, the error gets wiped out entirely by the floor, and so we always compute the correct result.

The dividend (numerator) is typically an N bit unsigned integer, where N is the size of a hardware register. For most divisors, k can be small enough that a valid magic number can also fit in N bits or fewer. But for many divisors, there is no such magic number. 7, 14, 19, 31, 42...these divisors require an N+1 bit magic number, which introduces inefficiences, as the magic number cannot fit in a hardware register.

Let us call such divisors "uncooperative." The algorithm presented here improves the performance of dividing by these uncooperative divisors by finding a new magic number which is no more than N bits. The existing algorithm that generates an N+1 bit magic number for uncooperative divisors will be referred to as the "round-up algorithm", because it rounds the true magic number up. The version presented here will be called the "round-down algorithm". We will say that an algorithm "fails" or "succeeds" according to whether it produces a magic number of N bits or fewer; we will show that either the round-up or round-down algorithm (or both) must succeed for all divisors.

All quantities used in the proofs and discussion are non-negative integers.

A Shift In Time Saves Fourteen

For completeness, it is worth mentioning one additional technique for uncooperative divisors that are even. Consider dividing a 32 bit unsigned integer by 14. The smallest valid magic number for 14 is 33 bits, which is inefficient. However, instead of dividing by 14, we can first divide by 2, and then by 7. While 7 is also uncooperative, the divide by 2 ensures the dividend is only a 31 bit number. Therefore the magic number for the subsequent divide-by-7 only needs to be 32 bits, which can be handled efficiently.

This technique effectively optimizes division by even divisors, and is incorporated in the reference code provided later. Now we present a technique applicable for odd divisors.

Motivation (aka What Goes Up Can Also Go Down)

First, an appeal to intuition. A divisor is uncooperative in the round-up algorithm because the rounding-up produces a poor approximation. That is, $\frac{2^k}{d}$ is just slightly larger than some integer, so the approximation $\lceil \frac{2^k}{d} \rceil$ is off by nearly one, which is a lot. It stands to reason, then, that we could get a better approximation by floor instead of ceil: $m = \lfloor \frac{2^k}{d} \rfloor$.

A naïve attempt to apply this immediately runs into trouble. Let d be any non-power-of-2 divisor, and consider trying to divide d by itself by multiplying with this magic number:

$$\lfloor \frac{2^k}{d} \rfloor < \frac{2^k}{d} \implies$$

$$\lfloor \frac{2^k}{d} \rfloor \times \frac{d}{2^k} < \frac{2^k}{d} \times \frac{d}{2^k} \implies$$

$$\lfloor \lfloor \frac{2^k}{d} \rfloor \times \frac{d}{2^k} \rfloor < 1$$

The result is too small.

(Could we replace the outer floor by a ceil? The floor is implemented by a right shift, which throws away the bits that are shifted off. We could conjure up a "rounding up" right shift, and that might work, though it would likely be more expensive than the instructions it replaces.)

So rounding down causes us to underestimate the result. What if we tried to counteract that by incrementing the numerator first?

$$\lfloor \frac{n}{d} \rfloor \overset{?}{=} \lfloor \lfloor \frac{2^k}{d} \rfloor \times \frac{n+1}{2^k} \rfloor$$

This is the round-down algorithm.

Proof of Correctness

First we must show that the round-down algorithm actually works. We proceed much like the proof for the round-up algorithm. We have a known constant d and a runtime variable n, both N bit values. We want to find some k that ensures:

$$\lfloor \frac{n}{d} \rfloor = \lfloor m \times \frac{n+1}{2^k} \rfloor$$

where:

$m = \lfloor \frac{2^k}{d} \rfloor$
$0 \leq n < 2^N$
$0 < d < 2^N$
d not a power of 2

Introduce an integer e which represents the error produced by the floor:

$m = \lfloor \frac{2^k}{d} \rfloor = \frac{2^k - e}{d}$

$0 < e < d$

Apply some algebra:

$$\begin{aligned}
\lfloor m \times \frac{n+1}{2^k} \rfloor &= \lfloor \frac{2^k - e}{d} \times \frac{n+1}{2^k} \rfloor \\
&= \lfloor \frac{n+1}{d} \times \frac{2^k - e}{2^k} \rfloor \\
&= \lfloor \frac{n+1}{d} \times (1 - \frac{e}{2^k}) \rfloor \\
&= \lfloor \frac{n+1}{d} - \frac{n+1}{d} \times \frac{e}{2^k} \rfloor \\
&= \lfloor \frac{n}{d} + \frac{1}{d} - \frac{e}{d} \times \frac{n+1}{2^k} \rfloor
\end{aligned}$$

We hope that this equals $\lfloor \frac{n}{d} \rfloor$. Within the floor, we see the result, plus two terms of opposite signs. We want the combination of those terms to cancel out to something at least zero, but small enough to be wiped out by the floor. Let us compute the fractional contribution of each term, and show that it is at least zero but less than one.

The fractional contribution of the $\frac{n}{d}$ term can be as small as zero and as large as $\frac{d-1}{d}$. Therefore, in order to keep the whole fractional part at least zero but below one, we require:

$0 \leq \frac{1}{d} - \frac{e}{d} \times \frac{n+1}{2^k} < \frac{1}{d}$

The term $\frac{e}{d} \times \frac{n+1}{2^k}$ is always positive, so the $< \frac{1}{d}$ is easily satisfied. It remains to show it is at least zero. Rearranging:

$0 \leq \frac{1}{d} - \frac{e}{d} \times \frac{n+1}{2^k} \implies \frac{e}{d} \times \frac{n+1}{2^k} \leq \frac{1}{d}$

This is very similar to the condition required in the round-up algorithm! Let's continue to simplify, using the fact that $n < 2^N$.

$\frac{e}{d} \times \frac{n+1}{2^k} \leq \frac{1}{d}$

$e \times \frac{n+1}{2^k} \leq 1$

$\frac{e}{2^{k-N}} \leq 1$

$e \leq 2^{k-N}$

This is the condition that guarantees that our magic number m works. In summary, pick some k ≥ N, and compute $e = 2^k \mod d$. If the resulting $e \leq 2^{k-N}$, the algorithm is guaranteed to produce the correct result for all N-bit dividends.

Proof of Universality (aka Your Weakness Is My Strength)

When will this condition be met? Recall the hand-waving from before: the round-up algorithm failed because rounding up produced a poor approximation, so we would expect rounding down to produce a good approximation, which would make the round-down algorithm succeed. Optimistically, we'd hope that round-down will succeed any time round-up fails! Indeed that is the case, and we can formally prove it now.

Here e_{up} refers to the difference produced by rounding 2^k up to a multiple of d, as in the round-up algorithm, while e_{down} refers to the difference produced by rounding down to a multiple of d as in round-down. An immediate consequence is $e_{up} + e_{down} = d$.

Recall from the round-up algorithm that we try successive values for k, with the smallest k guaranteed to work equal to $N + \lceil log_2 d \rceil$. Unfortunately, this k produces a magic number of N+1 bits, and so too large to fit in a hardware register. Let's consider the k just below it, which produces a magic number of N bits:

$$k = N + \lceil log_2 d \rceil - 1 = N + \lfloor log_2 d \rfloor$$

Assume that d is uncooperative, i.e. the magic number for this power was not valid in the round-up algorithm. It would have been valid if $e_{up} < 2^{\lfloor log_2 d \rfloor}$; because it was not valid we must have $e_{up} \geq 2^{\lfloor log_2 d \rfloor}$. Substituting in:

$$\begin{aligned} e_{up} &\geq 2^{\lfloor log_2 d \rfloor} \implies \\ d - e_{down} &\geq 2^{\lfloor log_2 d \rfloor} \implies \\ e_{down} &\leq d - 2^{\lfloor log_2 d \rfloor} \implies \\ e_{down} &\leq 2^{\lceil log_2 d \rceil} - 2^{\lfloor log_2 d \rfloor} \implies \\ e_{down} &\leq 2 \times 2^{\lfloor log_2 d \rfloor} - 2^{\lfloor log_2 d \rfloor} \implies \\ e_{down} &\leq 2^{\lfloor log_2 d \rfloor} \implies \\ e_{down} &\leq 2^{k-N} \end{aligned}$$

Thus we've satisfied the condition determined in the proof of correctness. This is an important and remarkable result: the round-down algorithm is guaranteed to have an efficient magic number whenever round-up does not. If the implementation of round-down can be shown to be more efficient, the overflow case for the round-up algorithm can be discarded entirely.

Recap

Here's the practical algorithm. Given a dividend n and a fixed divisor d, where $0 \leq n < 2^N$ and $0 < d < 2^N$, and where the usual round-up algorithm failed to find an N-bit magic number:

1. Consider in turn values of p in the range 0 through $\lfloor log_2 d \rfloor$, inclusive.
2. If $2^{N+p} \mod d \leq 2^p$, then we have found a working p. The last value in the range is guaranteed to work.
3. Once we have a working p, precompute the magic number $m = \lfloor \frac{2^{N+p}}{d} \rfloor$, which will be strictly less than 2^N.
4. Compute $q = (m \times (n+1)) \gg N$. This is typically implemented via a "high multiply" instruction.
5. Perform any remaining shift: $q = q \gg p$.

Overflow Handling

This algorithm has a wrinkle. Because n is an N-bit number, it may be as large as $2^N - 1$, in which event the n+1 term will be an N+1 bit number. If the value is simply incremented in an N-bit register, the dividend will wrap to zero, and the quotient will in turn be zero. Here we present two strategies for efficiently handling the possibility of modulo overflow.

Distributed Multiply Strategy

An obvious approach is to distribute the multiply through, i.e.:

$$m \times (n+1) = m \times n + m$$

This is a 2N-bit quantity and so cannot overflow. For efficient implementation, this requires that the low half of the m x n product be available "for free," so that the sum can be performed and any carry transmitted to the high half. Many modern architectures produce both halves with one instruction, such as Intel x86 (the *MUL* instruction) or ARM (*UMULL*). It is also available if the register width is twice the bit size of the type, e.g. performing a 32 bit divide on a 64 bit processor.

Saturating Increment Strategy

However, other processors compute the low and high halves separately, such as PowerPC; in this case computing the lower half of the product would be prohibitively expensive, and so a different strategy is needed. A second, surprising approach is to simply elide the increment if n is already at its maximum value, i.e. replace the increment with a "saturating increment" defined by:

$$\mathrm{SaturInc}(x) = \begin{cases} x+1 & \text{if } x < 2^N - 1 \\ x & \text{if } x = 2^N - 1 \end{cases}$$

It is not obvious why this should work: we needed the increment in the first place, so how can we just skip it? We must show that replacing increment with SaturInc will compute the correct result for 2^N - 1. A proof of that is presented below.

Proof of Correctness when using Saturating Increment

Consider the practical algorithm presented above, with the +1 replaced by saturating increment. If $n < 2^N - 1$, then saturating increment is the same as +1, so the proof from before holds. Therefore assume that $n = 2^N - 1$, so that incrementing n would wrap to 0.

By inspection, $\mathrm{SaturInc}(2^N - 1) = \mathrm{SaturInc}(2^N - 2)$. Because the algorithm has no other dependence on n, replacing the +1 with SaturInc effectively causes the algorithm to compute the quotient $\lfloor \frac{2^N-2}{d} \rfloor$ when n = 2^N-1.

Now d either is or is not a factor of 2^N-1. Let's start by assuming it is not a factor. It is easy to prove that, if x and y are positive integers and y is not a factor of x, then $\lfloor \frac{x}{y} \rfloor = \lfloor \frac{x-1}{y} \rfloor$. Therefore it must be true that $\lfloor \frac{2^N-1}{d} \rfloor = \lfloor \frac{2^N-2}{d} \rfloor$, so the algorithm computes the correct quotient.

Now let us consider the case where d is a factor of 2^N-1. We will prove that d is cooperative, i.e. the round-up algorithm produced an efficient N-bit result for d, and therefore the round-down algorithm is never employed. Because d is a factor of 2^N-1, we have $2^N \mod d = 1$. Consider once again the case of the "last N-bit magic number," i.e.:

$k = N + \lceil log_2 d \rceil - 1 = N + \lfloor log_2 d \rfloor$

Recall that the round-up algorithm computes $e_{up} = d - (2^k \mod d)$. This power is acceptable to the round-up algorithm if $e_{up} \leq 2^{k-N} = 2^{\lfloor log_2 d \rfloor}$. Consider:

$$\begin{aligned} 2^k \mod d &= 2^{N + \lfloor log_2 d \rfloor} \mod d \\ &= 2^N \times 2^{\lfloor log_2 d \rfloor} \mod d \\ &= 1 \times 2^{\lfloor log_2 d \rfloor} \mod d \\ &= 2^{\lfloor log_2 d \rfloor} \end{aligned}$$

Substituting in:

$$\begin{aligned} e_{up} &= d - 2^{\lfloor log_2 d \rfloor} \\ e_{up} &< 2^{\lceil log_2 d \rceil} - 2^{\lfloor log_2 d \rfloor} \\ &< 2 \times 2^{\lfloor log_2 d \rfloor} - 2^{\lfloor log_2 d \rfloor} \\ &< 2^{\lfloor log_2 d \rfloor} \end{aligned}$$

Thus the power k is acceptable to the round-up algorithm, so d is cooperative and the round-down algorithm is never employed. Thus a saturating increment is acceptable for all uncooperative divisors. Q.E.D.

(As an interesting aside, this last proof demonstrates that all factors of 2^N-1 "just barely" have efficient N-bit magic numbers. For example, the divisor 16,711,935 is a factor of 2^{32}-1, and its magic number, while N bits, requires a shift of 23, which is large; in fact it is the largest possible shift, as the floor of the base 2 log of that divisor. But increase the divisor by just one (16711936) and only a 16 bit shift is necessary.)

In summary, distributing the multiplication or using a saturating increment are both viable strategies for avoiding wrapping in the n+1 expression, ensuring that the algorithm works over the whole range of dividends. Implementations can use whichever technique is most efficient[1].

Practical Implementation

The discussion so far is only of theoretical interest; it becomes of practical interest if the round-down algorithm can be shown to outperform round-up on uncooperative divisors. This is what will be demonstrated below for x86 processors.

x86 processors admit an efficient saturating increment via the two-instruction sequence add 1; sbb 0; (i.e. "add; subtract 0 with borrow"). They also admit an efficient distributed multiply. The author implemented this optimization in the LLVM compiler using both strategies in turn, and then compiled the following C code which simply divides a value by 7, using clang -O3 -S -arch i386 -fomit-frame-pointer (this last flag for brevity):

```
unsigned int sevens(unsigned int x) {
    return x / 7;
}
```

Here is a comparison of the generated i386 assembly, with corresponding instructions aligned, and instructions that are unique to one or the other algorithm shown in red. (x86-64 assembly produced essentially the same insights, and so is omitted.)

```
Round-Up (Stock LLVM)       Distributive                Saturating Increment
_sevens:                    _sevens:                    _sevens:
  movl   4(%esp), %ecx                                    movl   4(%esp), %eax
                                                          addl   $1, %eax
                                                          sbbl   $0, %eax
  movl   $613566757, %edx     movl   $1227133513, %eax    movl   $1227133513, %ecx
  movl   %ecx, %eax
  mull   %edx                 mull   4(%esp)              mull   %ecx
  subl   %edx, %ecx           addl   $1227133513, %eax
  shrl   %ecx                 adcl   $0, %edx
  addl   %edx, %ecx
  shrl   $2, %ecx             shrl   %edx                 shrl   %edx
  movl   %ecx, %eax           movl   %edx, %eax           movl   %edx, %eax
  ret                         ret                         ret
```

The round-down algorithms not only avoid the three-instruction overflow handling, but also avoid needing to store the dividend past the multiply (notice the highlighted MOVL instruction in the round-up algorithm). The result is a net saving of two instructions. Also notice that the variants require fewer registers, which suggests there might be even more payoff (i.e. fewer register spills) when the divide is part of a longer code sequence.

(In the distributive variant the compiler has made the dubious choice to emit the same immediate twice instead of placing it in a register. This is especially deleterious in the loop microbenchmark shown below, because loading the immediate into the register could be hoisted out of the loop. To address this, the microbenchmark tests both the assembly as generated by LLVM, and a version tweaked by hand to address this suboptimal codegen.)

As illustrated, both strategies require only two instructions on x86, which is important because the overhead of the round-up algorithm is three to four instructions. Many processor architectures admit a two-instruction saturating increment through the carry flag[2].

Microbenchmark

To measure the performance, the author compiled a family of functions. Each function accepts an array of unsigned ints, divides them by a particular uncooperative divisor, and returns the sum; for example:

```
uint divide_7(const uint *x, size_t count) {
    uint result = 0;
    while (count--) {
        result += *x++ / 7;
    }
    return result;
}
```

Each function in the family had very similar machine code; a representative sample is:

```
Standard Round-Up          Distributive (hand tweaked)   Saturating Increment
_divide_7:                 _divide_7:                    _divide_7:
  pushl  %ebp                pushl  %ebp                   pushl  %ebp
  movl   %esp, %ebp          movl   %esp, %ebp             movl   %esp, %ebp
  pushl  %ebx                pushl  %ebx                   pushl  %ebx
  pushl  %edi                pushl  %edi                   pushl  %edi
  pushl  %esi                pushl  %esi                   pushl  %esi
  xorl   %ecx, %ecx          xorl   %ecx, %ecx             xorl   %ecx, %ecx
  movl   12(%ebp), %edi      movl   12(%ebp), %esi         movl   12(%ebp), %esi
  testl  %edi, %edi          testl  %esi, %esi             testl  %esi, %esi
  je     LBB1_3              je     LBB0_3                 je     LBB1_3
  movl   8(%ebp), %ebx       movl   8(%ebp), %edi          movl   8(%ebp), %edi
                             movl   $1227133513, %ebx      movl   $1227133513, %ebx
LBB1_2:                    LBB0_2:                       LBB1_2:
  movl   (%ebx), %esi        movl   (%edi), %eax           movl   (%edi), %eax
  movl   %esi, %eax                                        addl   $1, %eax
  movl   $613566757, %edx                                  sbbl   $0, %eax
  mull   %edx                mull   %ebx                   mull   %ebx
  subl   %edx, %esi          addl   %ebx, %eax
  shrl   %esi                adcl   $0, %edx
  addl   %edx, %esi
  shrl   $2, %esi            shrl   %edx                   shrl   %edx
  addl   %esi, %ecx          addl   %edx, %ecx             addl   %edx, %ecx
  addl   $4, %ebx            addl   $4, %edi               addl   $4, %edi
  decl   %edi                decl   %esi                   decl   %esi
  jne    LBB1_2              jne    LBB0_2                 jne    LBB1_2
LBB1_3:                    LBB0_3:                       LBB1_3:
  movl   %ecx, %eax          movl   %ecx, %eax             movl   %ecx, %eax
  popl   %esi                popl   %esi                   popl   %esi
  popl   %edi                popl   %edi                   popl   %edi
  popl   %ebx                popl   %ebx                   popl   %ebx
  popl   %ebp                popl   %ebp                   popl   %ebp
  ret                        ret                           ret
```

A simple test harness was constructed and the above functions were benchmarked to estimate the time per divide. The benchmark was compiled with clang on -O3, and run on a 2.93 GHz Core i7 iMac. Test runs were found to differ by less than .1%.

Nanoseconds Per Divide

	Divisor	Round Up	Saturating Increment		Distribute (as generated)		Distribute (hand tweaked)	
i386	7	1.632	1.484	9.1%	1.488	8.9%	1.433	12.2%
uint32	37	1.631	1.483	9.1%	1.486	8.9%	1.433	12.1%
	123	1.633	1.484	9.1%	1.488	8.9%	1.432	12.3%
	763	1.632	1.483	9.1%	1.487	8.9%	1.432	12.2%
	1247	1.633	1.484	9.1%	1.491	8.7%	1.433	12.2%
	9305	1.631	1.484	9.0%	1.491	8.6%	1.439	11.7%
	13307	1.632	1.483	9.1%	1.489	8.7%	1.437	11.9%
	52513	1.631	1.483	9.1%	1.490	8.7%	1.432	12.2%
	60978747	1.631	1.484	9.0%	1.488	8.8%	1.434	12.1%
	106956295	1.631	1.484	9.0%	1.489	8.7%	1.433	12.1%
x86_64	7	1.537	1.307	14.9%	1.548	-0.7%	1.362	11.4%
uint32	37	1.538	1.307	15.0%	1.548	-0.7%	1.362	11.4%
	123	1.537	1.319	14.2%	1.547	-0.6%	1.361	11.5%
	763	1.536	1.306	15.0%	1.547	-0.8%	1.356	11.7%
	1247	1.538	1.322	14.1%	1.549	-0.7%	1.358	11.7%
	9305	1.543	1.322	14.3%	1.550	-0.5%	1.361	11.8%
	13307	1.545	1.322	14.4%	1.550	-0.3%	1.357	12.1%
	52513	1.541	1.307	15.2%	1.550	-0.6%	1.361	11.7%
	60978747	1.538	1.322	14.0%	1.549	-0.7%	1.358	11.7%
	106956295	1.537	1.322	14.0%	1.551	-0.9%	1.360	11.5%
x86_64	7	1.823	1.588	12.9%	1.505	17.4%	n/a	
uint64	39	1.821	1.589	12.7%	1.506	17.3%	n/a	
	123	1.821	1.592	12.6%	1.506	17.3%	n/a	
	763	1.822	1.592	12.6%	1.505	17.4%	n/a	
	1249	1.822	1.589	12.8%	1.506	17.4%	n/a	
	9311	1.822	1.587	12.9%	1.507	17.3%	n/a	
	11315	1.822	1.588	12.8%	1.506	17.4%	n/a	
	52513	1.823	1.591	12.7%	1.506	17.4%	n/a	
	60978749	1.822	1.590	12.7%	1.507	17.3%	n/a	
	106956297	1.821	1.588	12.8%	1.506	17.3%	n/a	

Microbenchmark results for tested division algorithms on a Core i7. The top group is for 32 bit division in a 32 bit binary, while the bottom two groups are 32 bit and 64 bit division (respectively) in a 64 bit binary.

Times are in nanoseconds per divide (lower is better). Percentages are percent improvement from the Round Up algorithm (higher is better).

These results indicate that the round-down algorithms are indeed faster by 9%-17% (excluding the crummy codegen, which should be fixed in the compiler). The benchmark source code is available at http://ridiculousfish.com/files/division_benchmarks.tar.gz.

Extension to Signed Division

A natural question is whether the same optimization could improve signed division; unfortunately it appears that it does not, for two reasons:

- The increment of the dividend must become an increase in the magnitude, i.e. increment if $n > 0$, decrement if $n < 0$. This introduces an additional expense.
- The penalty for an uncooperative divisor is only about half as much in signed division, leaving a smaller window for improvements.

Thus it appears that the round-down algorithm could be made to work in signed division, but will underperform the standard round-up algorithm.

Reference Code

The reference implementation for computing the magic number due to Henry Warren ("Hacker's Delight") is rather dense, and it may not be obvious how to incorporate the improvements presented here. To ease adoption, we present a reference implementation written in C that incorporates all known optimizations, including the round-down algorithm.

This new reference implementation is available at
https://raw.github.com/ridiculousfish/libdivide/master/divide_by_constants_codegen_reference.c

Conclusion

The following algorithm is an alternative way to do division by "uncooperative" constants, which may outperform the standard algorithm that produces an N+1 bit magic number. Given a dividend n and a fixed divisor d, where $0 \le n < 2^N$ and $0 < d < 2^N$, and where the standard algorithm failed to find a N-bit magic number:

1. Consider in turn values of p in the range 0 through $\lfloor log_2 d \rfloor$, inclusive.
2. If $2^{N+p} \mod d \le 2^p$, then we have found a working p. The last value in the range is guaranteed to work (assuming the standard algorithm fails).
3. Once we have a working p, precompute the magic number $m = \lfloor \frac{2^{N+p}}{d} \rfloor$, which will be strictly less than 2^N.
4. To divide n by d, compute the value q through one of the following techniques:
   - Compute $q = (m \times n + m)) \gg N$, OR
   - Compute $q = (m \times (n+1)) \gg N$. If n+1 may wrap to zero, it is acceptable to use a saturating increment instead.
5. Perform any remaining shift: $q = q \gg p$.

On a Core i7 x86 processor, a microbenchmark showed that this variant "round down" algorithm outperformed the standard algorithm in both 32 bit and 64 bit modes by 9% to 17%, and in addition generated shorter code that used fewer registers. Furthermore, the variant algorithm is no more difficult to implement than is the standard algorithm. The author has provided a reference implementation and begun some preliminary work towards integrating this algorithm into LLVM, and hopes other compilers will adopt it.

Footnotes

1. Of course, if n can statically be shown to not equal 2^N-1, then the increment can be performed without concern for modulo overflow. This likely occurs frequently due to the special nature of the value 2^N-1.

2. Many processor architectures admit a straightforward saturating increment by use of the carry flag. PowerPC at first blush appears to be an exception: it has somewhat unusual carry flag semantics, and the obvious approach requires three instructions:

```
li r2, 0
addic r3, r3, 1
subfe r3, r2, r3
```

However PowerPC does admit a non-obvious two-instruction saturating increment. It does not seem to appear in the standard literature, and for that reason it is provided below. Given that the value to be incremented is in r3, execute:

```
subfic r2, r3, -2
addze r3, r3
```

The result is in r3. r2 can be replaced by any temporary register; its value can be discarded.