# Integer Division by Multiplying with the Double-Width Reciprocal

M. Anton Ertl*

TU Wien

**Abstract.** Earlier work on integer division by multiplying with the reciprocal has focused on multiplying with a single-width reciprocal, combined with a correction and followed by a shift. The present work explores using a double-width reciprocal to allow getting rid of the correction and shift.

## 1 Introduction

Integer division is expensive on many processors; e.g. on the Skylake microarchitecture[1] an unsigned division o a 64-bit number by a 64-bit number takes 35 cycles, and signed division 42 cycles. By contrast, on the same architecture a 64-bit-by-64-bit division with a 128-bit result takes 3 cycles, and one multiplication can be started per cycle [Fog19]. So replacing division by constants with multiplication by the reciprocal improves performance.

Therefore a number of compilers perform this optimization. However, in general this approach requires a $w+1$-bit reciprocal and a shift-right for dividing a $w$-bit number; instead, the compilers use a $w$-bit reciprocal and some divisor-dependent fixup code. In contrast, this paper explores using a $2w$-bit reciprocal without shift, and, for unsigned division, without fixup. This approach is actually fastest in some circumstances. It is also amenable to optimizing loop-invariant divisors, because the code for computing the parameters of the code in the loop body is relatively simple, especially for unsigned division. By contrast, neither gcc-8 nor clang-6.0 optimize loop-invariant divisors.

Still, in many cases it is probably better to implement other published work (see Section 5), and the main contribution of this paper is to show that using a $2w$-bit reciprocal can be beneficial in some cases, and to give an idea of where it works well.

Section 2 provides background on division by multiplication with the reciprocal. Section 3 discusses using a $2w$-bit reciprocal in unsigned division, while Section 4 discusses signed division (both using unsigned multiplication and signed multiplication).

* anton@mips.complang.tuwien.ac.at

[1] The microarchitecture of the mainstream client, server and mobile CPUs sold by Intel since late 2015, e.g., Core ix-6xxx...9xxx.

### 1.1 Symbols

In formulas in the rest of the paper we use the same letters for the same concepts:

| | |
|---|---|
| $n$ | dividend (numerator) |
| $d$ | divisor (denominator) |
| $q$ | quotient |
| $r$ | remainder |
| $w$ | word width in bits |
| $C$ | scaled approximate reciprocal |
| $k$ | $2^k$ is the scale factor for $C$ |
| $C_l$ | least significant word of $C$ |
| $C_h$ | most significant word of $C$ |
| $D$ | correction term (possibly scaled) |

## 2 Background

We first look at unsigned division. The basic idea of integer division by multiplication with the reciprocal is to replace integer division with a cheaper computation:

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

$\lfloor x/2^k \rfloor$ can be implemented with shifts, or, in our approach, by just selecting the right word of a multi-word result. Multiplication is also much cheaper than division on many CPUs. The question is how to determine $C$ and $k$.

A simple way to select $C$ is to compute

$$C = \lceil \frac{2^k}{d} \rceil = \frac{2^k + e}{d}$$

where $e$ ($0 \leq e < d$) indicates the size of the error we get from rounding up. So if we put that into our formula above, we get:

$$\lfloor \frac{n}{d} \rfloor = \lfloor n \frac{2^k + e}{d2^k} \rfloor = \lfloor \frac{n}{d} + \frac{ne}{d2^k} \rfloor$$

By increasing $k$, we can reduce the error, but this also increases $C$, requiring longer multiplication. Earlier work tried to go for the smallest $C$ and the smallest $k$ that produces the correct result. It turns out that, in general, for dealing with $w$-bit numbers, a $w+1$-bit $C$ is needed. In about 70% of the cases [Fis11], a $w$-bit $C$ is good enough (e.g., see Fig. 1, first column). For the other 30%, earlier work has devised various ways to make do with a $w$-bit by $w$-bit multiplication, by applying some form of correction or an alternative formula. E.g., one of these cases is $n/7$. Fig. 1 shows the code produced by gcc (second column) and by Robison [Rob05] and Fish [Fis11] (third column). The latter variant corresponds to the following formula (which works in those cases where the formula above for $C$ would require more than $w$ bits for $C$):

| gcc $n/10$<br>6 cycles latency | gcc $n/7$<br>8 cycles latency | Fish $n/7$<br>6.25 cycles latency | this paper $n/7$<br>6 cycles latency |
|---|---|---|---|
| `movabs $C,%rdx`<br>`mov %rdi,%rax`<br>`mul %rdx`<br>`mov %rdx,%rax`<br>`shr $0x3,%rax` | `movabs $C,%rdx`<br>`mov %rdi,%rax`<br>`mul %rdx`<br>`sub %rdx,%rdi`<br>`shr %rdi`<br>`lea (%rdx,%rdi),%rax`<br>`shr $0x2,%rax` | `movabs $C,%rax`<br>`mov %rax,%rcx`<br>`mul %rdi`<br>`add %rcx,%rax`<br>`adc $0x0,%rdx`<br>`shr $0x2,%rdx`<br>`mov %rdx,%rax` | `movabs $Cl,%rax`<br>`mul %rdi`<br>`mov %rdx,%rcx`<br>`movabs $Ch,%rax`<br>`mul %rdi`<br>`add %rcx,%rax`<br>`adc $0x0,%rdx`<br>`mov %rdx,%rax` |

**Fig. 1.** One way to divide by 10 and three ways to divide by 7; $n$ is in `%rdi`, result in `%rax`; latency numbers refer to latency from `%rdi` becoming ready until `%rax` is ready; they are measured on a Skylake (Core i5-6600K).

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

## 3 Using wider multiplication for unsigned division

The reason for trying to use a multiplier $C < 2^w$ is the presumption that multiplication with bigger C is expensive. While $w$-bit by $2w$-bit multiplication is more expensive than $w$-bit by $w$-bit multiplication, it is not that much more expensive, and using it can actually be cheaper than the workarounds for making do with $w$-bit by $w$-bit multiplication. E.g., the AMD64 code for unsigned 64-bit division by 7 is shown in the right column of Fig. 1; on Skylake[2] it has the same latency as the $n/10$ code produced by gcc, but it also works for the $n/7$ case.

This code works with $C = \lceil 2^{2w}/d \rceil$. As a result, we need $w$-bit by $2w$-bit multiplication, but we do not need a shift, because the result is in the most significant word of the multiplication result. We split $C$ into two words $C = C_l + 2^w C_h$, so the computation is

$$q = \lfloor \frac{nC_l + 2^w nC_h}{2^{2w}} \rfloor = \lfloor \frac{nC_l}{2^{2w}} + \frac{nC_h}{2^w} \rfloor$$

The least significant word of the first multiplication is eliminated by the flooring, and can be ignored; we add the most significant word of the first multiplication to the result of the second multiplication; the most significant word of this sum is our quotient.

The nice thing about this computation is that the two multiplications can theoretically be performed in parallel. In practice, current CPUs have only one

[2] Skylake is the microarchitecture of mainstream Intel CPUs since the 6th generation of Core i processors in 2015 and is also used in Intel CPUs of the 7th, 8th, and 9th generation, and some CPUs of the upcoming 10th generation.

multiplier, but this multiplier is pipelined, so the second multiplication can be started one cycle after the first. By using $k = 2w$, we eliminate the final shift-right used by the other variants, and its impact on latency (2 cycles on Skylake).

### 3.1 Computing $C$

Computing $C = \lceil 2^{2w}/d \rceil$ requires division of a $2w + 1$-bit number by a $w$-bit number, giving a $2w$-bit result. For just-in-time compilation and for dealing with loop-invariant divisors, we do not want to call a (slow) general multi-precision library. Fortunately, if we have a $2w/w \to w$ division (as present in AMD64), this computation can be performed relatively cheaply:

$$C = \lceil \frac{2^{2w}}{d} \rceil = \lfloor \frac{2^{2w} + d - 1}{d} \rfloor$$

$$C_h = \lfloor \frac{2^w}{d} \rfloor, r_h = 2^w \bmod d$$

$$C_l = \lfloor \frac{2^w r_h + d - 1}{d} \rfloor$$

In AMD64 assembly language:

```
#in:  d = %rdi
mov    $1,%rdx
mov    $0,%rax
div    %rdi
mov    %rax,%rsi
lea    -1(%rdi),%rax
div    %rdi
mov    %rsi,%rdx
#out: ch=%rdx cl=%rax
```

### 3.2 Special cases

Our approach works for $d > 1$.

For $d = 1$, $C = 2^w$, which does not fit in the two words that we reserve for reciprocals. If $d$ is known at compile-time, treating this as a special case is easy. If we use this approach for loop-invariant divisors, we can fork between $d = 1$ and $d > 1$ before the loop, and have a copy of the loop optimized for $d = 1$ (or maybe for powers of 2).

The other special case is $d = 0$. If $d$ is known at compile-time, the compiler can just produce the same code as in the unoptimized case, resulting in the same behaviour. For loop-invariant divisors, we want to minimize the number of special cases. What the compiler should do depends on the programming language, and, for some programming languages, on the compiler.

If the programming language specifies a specific behaviour for division-by-zero, we have to implement that. If the programming language does not define

the behaviour, some compiler writers think that they can do anything, and if you want to follow that path, you can simply ignore that case.

But I argue [Ert17] that even for these cases, the optimized code should exhibit the same behaviour as the unoptimized case. One way to deal with this is to define a specific behaviour (e.g., an exception) at the compiler level; then you have to implement that. Another way is to just use the hardware divide instruction in the unoptimized case and also for the "optimized" case; then a way to reduce special cases is to combine this case with $d = 1$ and use a loop that uses the divide instruction (i.e., without optimizing the division) for $d < 2$.

Most architectures define the behaviour on divide-by-zero, and in these cases we may be able to do better:

E.g., the AMD64 architecture specifies that division-by-zero produces an exception. For the loop-invariant case, this exception will occur when computing $C$, and we do not need to worry about $d = 0$ in the rest of the loop. However, this means that for $d \neq 1$ we have to peel the first iteration of the loop, and compute $C$ at the place of that first-iteration division (in order to preserve the order of exceptions).

Aarch64 specifies that division-by-zero produces 0. Then we can just use $C = 0$, and using that in $\lfloor nC/2^{2w} \rfloor$ gives 0, just like the original div instruction. However, at least Cortex-A53, -A72 and -A73 have very fast dividers, so a compiler probably should forego replacing division by multiplication with the reciprocal for these CPUs. Still, if there are CPUs with slow division instructions where division-by-zero produces 0, you can apply these considerations.

### 3.3 Possible usage

As a compiler writer, the simplest way to make use of this paper's approach is to use it for all $d > 1$. If you want to invest more development resources into this topic, you can detect the 70% of divisors where a $C < 2^w$ is sufficient, and use a multiply followed by a shift-right for that. You can also optimize dividing by powers-of-two into shift-right.

Finally, if you have enough development resources, you can also detect and optimize loop-invariant divisors. For this use, every special case needs a separate copy of the loop. The benefit of this paper's approach is that it reduces the number of cases that need separate loops; in particular, you can handle $d > 1$ with one loop.

One usage case for loop-invariant divisors is the conversion of integers to strings with arbitrary base (e.g., Java's `Integer.toString(int i, int radix)`). This usage has relatively low trip counts, but also few different radixes (and often the radix is the same as in the last invocation). Given the relatively high cost to compute $C$, the low trip counts would be a problem. This can be mitigated by memoizing the computation of $C$.

### 3.4 Remainder

The remainder of the division can be computed from the quotient in the obvious way.

$$r = n \bmod d = n - qd$$

This is used with every way to compute $q$ (except that on some architectures (e.g., AMD64), the division instruction produces both $q$ and $r$), including the various ways to performed signed division, so there is no need to discuss it in the rest of this paper.

## 4 Signed division

### 4.1 2s-complement numbers

In this section we work with the 2s-complement representation of negative integers. This representation is used in all significant architectures introduced since 1970. In 2s-complement representation, a negative number $x$ is represented by $x' = x + 2^w$. As a result signed multiplication of $a < 0$ with $b$ becomes

$$ab = (a' - 2^w)b = a'b - 2^w b$$

and likewise for $b < 0$. Widening signed multiplication[3] instructions perform these corrections internally. If we use unsigned multiplication instructions, we have to generate code to perform them.

### 4.2 Signed integer division

In symmetric (aka truncated) division $n/d$, the quotient is rounded towards 0 (truncated), and consequently a non-zero remainder is negative iff the signs of $n$ and $d$ differ.

In floored division, the quotient is rounded towards $-\infty$ (floored), and consequently, a non-zero remainder has the same sign as $d$.

There is also Euclidean division, where $0 \le r < |d|$ [Bou92].

Different programming languages specify different forms of division and, in particular, remainder operations; some of them support several (using different operators, e.g., `mod` and `rem` in Ada).[4]

[3] $2^w b \equiv 0 \pmod{2^w}$, so in non-widening multiplication the difference between signed and unsigned multiplication vanishes.

[4] https://en.wikipedia.org/wiki/Modulo_operation

### 4.3 Signed division by unsigned reciprocal multiplication

We first look at using unsigned multiplication.

For $d > 1$, we can use the same $C$ as in the unsigned-division case. However, if $n < 0$, we have to compute

$$nC = (n' - 2^w)C = n'C - 2^wC$$

But there is more: $C$ is a little bigger than the ideal multiplier $2^{2w}/d$. For $n < 0$, this means that the result is a little smaller than $n/d$, and rounding it towards $-\infty$ produces a result that is one less than the result of symmetric division.

So, for symmetric division, if $n < 0$, the total correction term is

$$D_s = 2^{2w} - 2^wC$$

This is used in

$$q = \lfloor \frac{nC_l + 2^w nC_h + D}{2^{2w}} \rfloor \text{ if } n < 0$$

For floored and Euclidean division (they are the same for $d > 0$), rounding towards $-\infty$ is the right thing in principle, but we have to correct for that fact that $C$ is a little too big, which, combined with rounding towards $-\infty$, produces a division result that is 1 too low when $n$ is divisible by $d$. One way to correct for that is to add no more than $1/d$ to the end result before flooring. In terms of the scaled correction term this turns out to be $C - 1$ ($C$ itself is a bit too large in general). So we could use the following total correction term:

$$D_{f'} = C - 1 - 2^wC$$

This would require a $3w$-bit addition. In practice we can make do with a $2w$-bit addition by using

$$D_f = 2^w(\lfloor \frac{C-1}{2^w} \rfloor - C)$$

If you also want to optimize negative divisors (they are very rare), one way to deal with them for symmetric and floored division is to perform division by $-d$, with appropriate corrections elsewhere:

For symmetric division, you negate either the quotient $q$ (cheaper), or the dividend $n$.

For floored division, you negate the dividend $n$.

For Euclidean division, you negate the quotient $q$.

Special cases: In addition to $d = 0$ and $d = 1$, signed division also has the special case $d = -1$; this special case has an additional problem compared to $d = 1$: The result of $-2^{w-1}/-1 = 2^{w-1}$ is not representable as a $w$-bit 2s-complement number. The simplest way to deal with $d = -1$ is not to optimize it.

### 4.4 Using signed multiplication

It seems obvious to use signed multiplication for signed division, and approaches that use single-width reciprocals often do so. However, even these approaches perform corrections, either for $n < 0$ or independent of the sign, so at least for $d > 0$ these approaches do not offer an advantage over using unsigned multiplication (and $d < 0$ is very rare). Still, this section reports on my explorations in this direction.

Let's consider $d > 2$ first. For $n \geq 0$, this coincides with unsigned division. For $n < 0$:

$$\lfloor \frac{nC}{2^{2w}} \rfloor = \lfloor \frac{n'C - 2^w C}{2^{2w}} \rfloor = \lfloor \frac{n'C_l - 2^w C_l}{2^{2w}} + \frac{n'C_h - 2^w C_h}{2^w} \rfloor$$

So signed multiplication can be used for multiplying both parts of $C$. There is one complication, though: The result of the low-order multiplication can be negative and needs to be sign-extended to the full width before performing the addition.

For symmetric division, we still need to perform the correction depending on the sign of $n$.

For floored and Euclidean division, one can perform a correction that is constant across the whole range of $n$, resulting in the computation

$$q = \lfloor \frac{n'C_l - 2^w C_l}{2^{2w}} + \frac{n'C_h - 2^w C_h}{2^w} + \frac{D_g}{2^{2w}} \rfloor$$

and we use

$$D_{g'} = 2^{w-1}(\frac{C}{2^{2w}} - \frac{1}{d})$$

$$D_g = \lceil 2^{2w} D_g \rceil$$

$D'_g$ is the maximum amount by which our approximation $C/2^{2n}$ deviates from the exact $1/d$ in the negative range of $n$. $D_g$ is the next integer scaled by $2^{2w}$ (as used in the formula for $q$).

The advantage of this approach is that there is no need for a branch or other kind of conditional code. However, in the frequent case that $n \geq 0$ for almost all invocations, it is probably better to use unsigned multiplication with conditional correction for $n < 0$.

For $d < -1$, signed multiplication works nicely (with $C = -\lceil 2^{2w}/-d \rceil = \lfloor 2^{2w}/d \rfloor$), and avoids the need for a separate negation step and is therefore probably better than the variant using unsigned multiplication above. The correction term has to be adjusted appropriately; in particular, the correction factor for Euclidean division is now different than for floored division.

Using signed multiplication turns $d = 2$ into a special case, because $C = 2^{2w-1}$ is not representable as signed $2w$-bit number. It can be handled by treating all $d = 2^i$ as separate case that uses arithmetic shift right; this would also cover $d = 1$.

## 5 Related work

If you read only one paper about the topic, my recommendation is Robison's [Rob05]. The main benefit of that work is that division is replaced by the computation

$$\lfloor \frac{n}{d} \rfloor = \lfloor \frac{an + b}{2^k} \rfloor$$

where $a$ and $b$ are $w$-bit numbers and not much harder to compute than $C$ in the present paper (and $b = 0$ or $b = a$). So this approach can be used for loop-invariant divisors. Depending on the circumstances, Robison's computation can be faster or slower than the present paper's computation.

Fish [Fis11] arrives more or less at Robison's computation through different reasoning and suggests an alternative way of coding it.

Early work on division by constants by Artzy et al. [AHS76] did not frame it as using an approximation to the reciprocal.

Alverson [Alv91] presents unsigned and signed division by multiplying with a $w + 1$-bit reciprocal and shifting. Alverson performs signed division on absolute values using unsigned multiplication, with a correction term (bias) for floored division, and correcting the sign in the end.

Granlund and Montgomery [GM94] use signed multiplication (with correction) for the symmetric signed division case, and but use unsigned multiplication and sign manipulation for the signed division case. They also discuss using floating-point multiplication, division of a double-word dividend, the case where it is known that the remainder is 0, implementation in gcc, and results.

Möller and Grandlund [MG11] perform double-word by single-word division using single-word multiplication plus corrections; in a way, the opposite of the present paper, which uses a wider multiplier to eliminate corrections, and only produces a single-word result.

Muller et al. [MTdDM05] discuss implementing 32-bit by 32-bit division using 16-bit by 16-bit multiplication, also in the opposite direction of the present work.

Other works on division by reciprocal multiplication are by Warren [War03, Chapter 10], Cavignino and Werbrouck [CW08,CW11], and Drane et al. [DCC12].

## 6 Conclusion

In division by multiplying with the reciprocal, using a double-wide reciprocal eliminates the final shift, which can reduce the latency of the whole operation. The benefit is most pronounced for unsigned division, while for signed division conditional or unconditional corrections are needed, which may make using a double-wide reciprocal less attractive.

## References

AHS76. E. Artzy, J. A. Hinds, and H. J. Saal. A fast division technique for constant divisors. *Communications of the ACM*, 19(2):98–101, February 1976. 5

Alv91. Robert Alverson. Integer division using reciprocals. In *Proceedings 10th IEEE Symposium on Computer Arithmetic*, pages 186–190, Grenoble, France, June 1991. 5

Bou92. Raymond T. Boute. The Euclidian definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems*, 14(2):127–144, April 1992. 4.2

CW08. D. Cavagnino and A. E. Werbrouck. Efficient algorithms for integer division by constants using multiplication. *The Computer Journal*, 51(4):470–480, July 2008. 5

CW11. D. Cavagnino and A. E. Werbrouck. An analysis of associated dividends in the DBM algorithm for division by constants using multiplication. *The Computer Journal*, 54(1):148–156, January 2011. 5

DCC12. Theo Drane, Wai-Chuen Cheung, and George A. Constantinides. Correctly rounded constant integer division via multiply-add. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1243–1246. IEEE, 2012. 5

Ert17. M. Anton Ertl. The intended meaning of *Undefined Behaviour* in C programs. In Wolfram Amme and Thomas Heinze, editors, *19. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'17)*, pages 20–28, 2017. 3.2

Fis11. Fish. Labor of division (Episode III): Faster unsigned division by constants. Blog Entry `https://ridiculousfish.com/blog/posts/labor-of-division-episode-iii.html`, October 2011. 2, 5

Fog19. Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. `https://www.agner.org/optimize/instruction_tables.pdf`, 2019. 1

GM94. Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 61–72, Orlando, Fla., June 1994. 5

MG11. Niels Möller and Torbjörn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, February 2011. 5

MTdDM05. Jean-Michel Muller, Arnaud Tisserand, Benoit de Dinechin, and Christophe Monat. Division by constant for the ST100 DSP microprocessor. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, 2005. 5

Rob05. Arch D. Robison. $N$-bit unsigned division via $N$-bit multiply-add. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, 2005. 2, 5

War03. Henry S. Warren, Jr. *Hacker's Delight*. Addison-Wesley, 2003. 5