RC 18601 (81407) 12/21/92
Computer Science 44 pages

Research Report

Changing Division by a Constant to Multiplication in Two's-complement Arithmetic

Henry S. Warren, Jr.

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

IBM Research Division
Almaden • T.J. Watson • Tokyo • Zurich

Changing Division by a Constant to Multiplication in Two's-complement Arithmetic

Henry S. Warren, Jr.

IBM Research
T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract: This report gives proofs that on a two's complement computer, signed and unsigned integer division by any constant can be replaced by a multiplication by another constant and a few other instructions. Algorithms are given for computing the multiplier corresponding to a given divisor.

CONTENTS

List of Illustrations

Introduction

According to programming lore, the quotient of an integer n divided by another integer d can be obtained by multiplying n by a sort of reciprocal of d, namely $2^W/d$, and then extracting the leftmost W bits of the $2W$-bit product. Here W is the word (register) size of the machine; typically $W = 32$ (bits). The observation follows from the identity

$$n/d = ((2^W/d)n)/2^W.$$

But to use this, $2^W/d$ must be rounded to a W-bit integer. Thus it is an approximation method and the multiplier is limited to W bits of accuracy. One has good cause to wonder: Does this method really work? For all divisors? For both signed and unsigned division?

The answer to these questions is "yes," subject to the minor restrictions that $d \neq 0$ and, for signed division, $d \neq \pm 1$, and provided we add a few elementary instructions (shifts and adds) in many cases. For most computers, the number of additional instructions ranges from two to four for signed division, and from about one to four for unsigned division. Thus the transformation is useful on machines for which the fixed-point divide time exceeds the fixed-point multiply time by about four cycles or more.

In the following material we first consider signed division by a positive divisor in some detail. Then we sketch the derivations and proofs for signed division by a negative divisor, and for unsigned division.

Integer division

By "signed division" of integers we mean the form of division that is almost universally used in high level languages and computer instruction sets. It might be called "truncating" division because the result is obtained by discarding the fractional digits from the rational number n/d. If we denote this form of signed division by $n \div d$, then it satisfies

$$(-n) \div d = n \div (-d) = -(n \div d) \quad \text{for} \quad d \neq 0.$$

Care must be exercised when applying this to transform programs, because if n or d is the maximum negative number, $-n$ or $-d$ cannot be represented in W bits. The operation $(-2^{W-1})/(-1)$ is an overflow (the result cannot be expressed as a signed quantity in two's-complement notation), and on most machines the result is undefined or the operation is suppressed.

Signed integer division is related to ordinary rational division by

$$n \div d = \lfloor n/d \rfloor, \quad \text{if} \quad d \neq 0, nd \geq 0, \tag{1a}$$

$$n \div d = \lceil n/d \rceil, \quad \text{if} \quad d \neq 0, nd < 0. \tag{1b}$$

Unsigned integer division, i.e. division in which both n and d are interpreted as unsigned integers, satisfies (1a).

Assembly language notation

To illustrate machine code, we use the instructions shown below. The leftmost operand is the target register.

`abs`	`rt,ra`	Absolute value
`add`	`rt,ra,rb`	Add, rt ← (ra) + (rb)
`addze`	`rt,ra`	rt ← (ra) + carry
`li`	`rt,x'xxx'`	Load immediate; rt ← x'xxx' (could be a load from storage)
`mulhs`	`rt,ra,rb`	Multiply high signed, rt ← high-order half of (ra)×(rb), with operands interpreted as signed integers
`mulhu`	`rt,ra,rb`	Multiply high unsigned, rt ← high-order half of (ra)×(rb), with operands interpreted as unsigned integers
`muli`	`rt,ra,i`	Multiply immediate
`srai`	`rt,ra,i`	Shift right algebraic immediate (sign-fill)
`sri`	`rt,ra,i`	Shift right immediate (0-fill)
`sub`	`rt,ra,rb`	Subtract, rt ← (ra) - (rb)
`xor`	`rt,ra,rb`	Exclusive OR

On many machines, the *multiply high* instructions would be the machine's multiply instruction that gives a double-length product, followed if necessary by an instruction to move the high-order half from some special-purpose register to a general register. Only the high-order half of the product is needed for the methods described here. We need signed multiply for signed division, and unsigned multiply for unsigned division.

Signed division by a power of two

Although not the main subject of this paper, let us quickly dispense with (signed) division by a power of two. The following branch-free[1] code sets $q = n \div 2^k$, for $1 \le k \le 31$ (Hop), on a 32-bit machine.

```
srai  t,n,k-1           # Form the number
sri   t,t,32-k          # 2**k - 1 if n < 0, else 0.
add   q,n,t             # Add it to n,
srai  q,q,k             # and shift right.
```

(Proof omitted). If $k = 1$ (so that the divisor is two), the first instruction is not necessary. The case $k = 31$ does not make too much sense, because the number 2^{31} is not representable in the machine, but nevertheless the code does produce the correct result in that case (which is $q = -1$ if $n = -2^{31}$ and $q = 0$ for all other n).

The methods to be given are correct for divisors that are a power of two, but the code produced is not as good as that shown above (it will contain a *multiply*).

The IBM RISC System/6000 has an unusual device for speeding up division by a power of two (GGS). The *shift right algebraic* instructions set the machine's carry bit if the number being shifted is negative and one or more 1-bits are shifted out. That machine also has an instruction for adding the carry bit to a register, which we denote *addze*. This allows division by any (positive) power of two to be done in two instructions:

```
srai  q,n,k             # Shift (n) right k positions.
addze q,q               # Add in carry.
```

[1] We prefer branch-free code, on the presumption that branches take more than one fixed-point cycle on average, and branch-free code is often advantageous from the standpoint of compiler optimizations.

Preliminaries

The proofs herein use the following elementary properties of arithmetic, which are not proved here.

Theorem 1. For n, d integers, $d > 0$,

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d+1}{d} \right\rceil \quad \text{and} \quad \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d-1}{d} \right\rfloor.$$

If $d < 0$:

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d-1}{d} \right\rceil \quad \text{and} \quad \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d+1}{d} \right\rfloor.$$

Theorem 2. For x real, d an integer $\neq 0$:

$$\lfloor \lfloor x \rfloor / d \rfloor = \lfloor x/d \rfloor \quad \text{and} \quad \lceil \lceil x \rceil / d \rceil = \lceil x/d \rceil .$$

Corollary. For a, b real, $b \neq 0$, d an integer $\neq 0$:

$$\left\lfloor \left\lfloor \frac{a}{b} \right\rfloor / d \right\rfloor = \left\lfloor \frac{a}{bd} \right\rfloor \quad \text{and} \quad \left\lceil \left\lceil \frac{a}{b} \right\rceil / d \right\rceil = \left\lceil \frac{a}{bd} \right\rceil.$$

Theorem 3. For n, d integers, $d \neq 0$, and x real,

$$\left\lfloor \frac{n}{d} + x \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \text{ if } 0 \le x < \left| \frac{1}{d} \right|, \quad \text{and} \quad \left\lceil \frac{n}{d} + x \right\rceil = \left\lceil \frac{n}{d} \right\rceil \text{ if } -\left| \frac{1}{d} \right| < x \le 0.$$

In the theorems below, $rem(n, d)$ denotes the remainder of n divided by d. For negative d, it is defined by $rem(n, -d) = rem(n, d)$. We do not use $rem(n, d)$ with $n < 0$. Thus for our use, the remainder is always nonnegative.

Theorem 4. For $n \ge 0$, $d \neq 0$,

$$\overset{\text{(a)}}{rem(2n, d) = \begin{cases} 2rem(n, d) & \text{or} \\ 2rem(n, d) - |d|, \end{cases}} \quad \text{and} \quad \overset{\text{(b)}}{rem(2n+1, d) = \begin{cases} 2rem(n, d) + 1 & \text{or} \\ 2rem(n, d) - |d| + 1. \end{cases}}$$

(whichever value is greater than or equal to zero and less than $|d|$).

Theorem 5. For $n \ge 0$, $d \neq 0$,

$$rem(2n, 2d) = 2rem(n, d).$$

Theorems 4 and 5 are easily proved from the basic definition of remainder, i.e. that for some integer q it satisfies

$$n = qd + rem(n, d), \quad \text{with} \quad 0 \le rem(n, d) < |d|$$

provided $n \ge 0$ and $d \neq 0$ (n and d can be non-integers, but we will use these theorems only for integers).

Some examples

Let us consider a few specific examples for a 32-bit machine. These serve as a sort of warm-up exercise for the main proof in the following section, and they illustrate the code that will be generated by the general method.

We denote registers as follows:

```
n - the input number (numerator)
M - loaded with a "magic number"
t - a work register
q - will contain the quotient
r - will contain the remainder
```

Divide by three

```
li    M,x'55555556'    # Load magic number, (2**32+2)/3.
mulhs q,M,n            # q = floor(M*n/2**32).
sri   t,n,31           # Add 1 to q if
add   q,q,t            # n is negative.

muli  t,q,3            # Compute remainder from
sub   r,n,t            # r = n - q*3.
```

Proof. The *multiply* cannot overflow, as the product of two 32-bit numbers can always be represented in 64 bits. The *mulhs* gives the leftmost 32 bits of the 64-bit product. This is equivalent to dividing the 64-bit product by 2^{32} and taking the "floor" of the result, and this is true whether the product is positive or negative. Thus, for $n \geq 0$ the above code computes

$$q = \left\lfloor \frac{2^{32}+2}{3} \frac{n}{2^{32}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{2n}{3 \cdot 2^{32}} \right\rfloor.$$

Now $n < 2^{31}$, because $2^{31} - 1$ is the largest representable positive number. Hence the "error" term $2n/3 \cdot 2^{32}$ is less than 1/3 (and is nonnegative), so by Theorem 3 we have $q = \lfloor n/3 \rfloor$, which is the desired result (1a).

For $n < 0$, there is an addition of one to the quotient. Hence the code computes

$$q = \left\lfloor \frac{2^{32}+2}{3} \frac{n}{2^{32}} \right\rfloor + 1 = \left\lfloor \frac{2^{32}n + 2n + 3 \cdot 2^{32}}{3 \cdot 2^{32}} \right\rfloor = \left\lceil \frac{2^{32}n + 2n + 1}{3 \cdot 2^{32}} \right\rceil$$

where we have used Theorem 1. Hence

$$q = \left\lceil \frac{n}{3} + \frac{2n+1}{3 \cdot 2^{32}} \right\rceil.$$

For $-2^{31} \leq n \leq -1$,

$$-\frac{1}{3} + \frac{1}{3 \cdot 2^{32}} \leq \frac{2n+1}{3 \cdot 2^{32}} \leq -\frac{1}{3 \cdot 2^{32}}.$$

The error term is nonpositive and greater than $-1/3$, so by Theorem 3 $q = \lceil n/3 \rceil$, which is the desired result (1b).

This establishes that the quotient is correct. That the remainder is correct follows easily from the fact that the remainder must satisfy

$$n = qd + r,$$

the multiplication by three cannot overflow (because $-2^{31}/3 \le q \le (2^{31}-1)/3$), and the *subtract* cannot overflow because the result must be in the range -2 to $+2$.

The *muli* can be done with two *add*'s, if that gives an improvement in execution time.

The above code can be implemented on the IBM RISC System/6000 by using a load from storage for the *li*, moving the *sri* back to just after the load, to cover the one-cycle load delay, and using the *mul* instruction for the multiply, disregarding the low-order 32 bits of the product that are placed in the MQ register. The *mul* instruction takes five cycles, giving a total time of eight cycles to compute the quotient. For comparison, the *divide* instruction on that machine takes 19 cycles, which together with the *load immediate* of three takes 20 cycles.

To compute the remainder on the IBM RISC System/6000, the above code takes 11 cycles (using two *add*'s for the *muli*), as compared to 21 using the machine's *divide* (the extra cycle is to move the remainder from the MQ to a general purpose register).

Divide by five

For division by five, we would like to use the same code as for division by three, except with a multiplier of $(2^{32}+4)/5$. Unfortunately, the error term is then too large; the result is off by one for about 1/5 of the values of $n \ge 2^{30}$ in magnitude. However, we can use a multiplier of $(2^{33}+3)/5$ and add a *shift right algebraic* instruction. The code is:

```
li    M,x'66666667'    # Load magic number, (2**33+3)/5.
mulhs q,M,n            # q = floor(M*n/2**32).
srai  q,q,1
sri   t,n,31           # Add 1 to q if
add   q,q,t            # n is negative.

muli  t,q,5            # Compute remainder from
sub   r,n,t            # r = n - q*5.
```

Proof. The *mulhs* produces the leftmost 32 bits of the 64-bit product, and then the code shifts this right by one position, algebraically. This is equivalent to dividing the product by 2^{33} and then taking the "floor" of the result. Thus for $n \ge 0$ the code computes

$$q = \left\lfloor \frac{2^{33}+3}{5}\,\frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{5} + \frac{3n}{5 \cdot 2^{33}} \right\rfloor.$$

For $0 \le n < 2^{31}$, the error term $3n/5 \cdot 2^{33}$ is nonnegative and less than 1/5, so by Theorem 3, $q = \lfloor n/5 \rfloor$.

For $n < 0$, the above code computes

$$q = \left\lfloor \frac{2^{33}+3}{5} \frac{n}{2^{33}} \right\rfloor + 1 = \left\lceil \frac{n}{5} + \frac{3n+1}{5 \cdot 2^{33}} \right\rceil .$$

The error term is nonpositive and greater than $-1/5$, so $q = \lceil n/5 \rceil$.

That the remainder is correct follows as in the case of division by three.

The *muli* can be done with a *shift left* of two and an *add.*

Divide by seven

For dividing by seven we have a new problem. Multipliers of $(2^{32}+3)/7$ and $(2^{33}+6)/7$ give error terms that are too large. A multiplier of $(2^{34}+5)/7$ would work, but it's too large to represent in a 32-bit signed word. We can multiply by this large number by multiplying by $(2^{34}+5)/7 - 2^{32}$ (a negative number), and then correcting the product by inserting an *add.* The code is:

```
li    M,x'92492493'     # Load magic number, (2**34+5)/7 - 2**32.
mulhs q,M,n             # q = floor(M*n/2**32).
add   q,q,n             # q = floor(M*n/2**32) + n.
srai  q,q,2             # q = floor(q/4).
sri   t,n,31            # Add 1 to q if
add   q,q,t             # n is negative.

muli  t,q,7             # Compute remainder from
sub   r,n,t             # r = n - q*7.
```

Proof. It is important to note that the instruction "*add q,q,n*" above cannot overflow. This is because q and n have opposite signs, due to the multiplication by a negative number. Therefore this "computer arithmetic" addition is the same as real number addition. Hence for $n \geq 0$ the above code computes

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) /4 \right\rfloor = \left\lfloor \left\lfloor \frac{2^{34}n + 5n - 7 \cdot 2^{32}n + 7 \cdot 2^{32}n}{7 \cdot 2^{32}} \right\rfloor /4 \right\rfloor$$
$$= \left\lfloor \frac{n}{7} + \frac{5n}{7 \cdot 2^{34}} \right\rfloor ,$$

where we have used the corollary of Theorem 2.

For $0 \leq n < 2^{31}$, the error term $5n/7 \cdot 2^{34}$ is nonnegative and less than $1/7$, so $q = \lfloor n/7 \rfloor$.

For $n < 0$, the above code computes

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) /4 \right\rfloor + 1 = \left\lceil \frac{n}{7} + \frac{5n+1}{7 \cdot 2^{34}} \right\rceil .$$

The error term is nonpositive and greater than $-1/7$, so $q = \lceil n/7 \rceil$.

The *muli* can be done with a *shift left* of three and a *subtract.*

Signed division by divisors ≥ 2

At this point the reader may wonder if other divisors present other problems. We will see in this section that they do not: the three examples given illustrate the only cases that arise (for $d \geq 2$).

Given a word size $W \geq 3$ and a divisor d, $2 \leq d < 2^{W-1}$, we wish to find the least integer m and integer p such that

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \qquad \text{for } 0 \leq n < 2^{W-1}, \quad \text{and} \tag{2a}$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \quad \text{for } -2^{W-1} \leq n \leq -1, \tag{2b}$$

with $0 \leq m < 2^W$ and $p \geq W$.

The reason we want the *least* integer m is that a smaller multiplier may give a smaller shift amount (possibly zero) or may yield code similar to the "divide by five" example, rather than the "divide by seven" example. We must have $m \leq 2^W - 1$ so the code has no more instructions than that of the "divide by seven" example (i.e we can handle a multiplier in the range 2^{W-1} to $2^W - 1$ by means of the *add* that was inserted in the "divide by seven" example, but we would rather not deal with larger multipliers). We must have $p \geq W$ because the generated code extracts the left half of the product mn, which is equivalent to shifting right W positions. Thus the total right shift is W or more positions.

There is a distinction between the multiplier m and the "magic number," denoted M. The magic number is the value used in the *multiply* instruction. It is given by

$$M = \begin{cases} m, & \text{if } 0 \leq m < 2^{W-1}, \\ m - 2^W, & \text{if } 2^{W-1} \leq m < 2^W. \end{cases}$$

Since (2b) must hold for $n = -d$, $\lfloor -md/2^p \rfloor + 1 = -1$, or

$$\frac{md}{2^p} > 1. \tag{3}$$

Let n_c be the largest (positive) value of n such that $rem(n_c, d) = d - 1$. n_c exists because one possibility is $n_c = d - 1$. It can be calculated from $n_c = \lfloor 2^{W-1}/d \rfloor d - 1 = 2^{W-1} - rem(2^{W-1}, d) - 1$. n_c is one of the highest d admissible values of n, so

$$2^{W-1} - d \leq n_c \leq 2^{W-1} - 1, \tag{4a}$$

and clearly

$$n_c \geq d - 1. \tag{4b}$$

Since (2a) must hold for $n = n_c$,

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d},$$

or

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}.$$

Combining this with (3):

$$\frac{2^p}{d} < m < \frac{2^p}{d}\,\frac{n_c + 1}{n_c}. \tag{5}$$

Since m is to be the least integer satisfying (5), it is the next integer greater than $2^p/d$, i.e.

$$\boxed{m = \frac{2^p + d - rem(2^p, d)}{d}.} \tag{6}$$

Combining this with the right half of (5) and simplifying gives:

$$\boxed{2^p > n_c(d - rem(2^p, d)).} \tag{7}$$

The algorithm

Thus the algorithm to find the magic number M and the shift amount s from d is to first compute n_c, and then solve (7) for p by trying successively larger values. If $p < W$, set $p = W$ (the theorem below shows that this value of p also satisfies (7)). When the smallest $p \geq W$ satisfying (7) is found, m is calculated from (6). This is the smallest possible value of m, because we found the smallest acceptable p, and from (5) clearly smaller values of p yield smaller values of m. Finally, $s = p - W$ and M is simply a reinterpretation of m as a signed integer (which is how the *mulhs* instruction interprets it).

Forcing p to be at least W is justified by the following:

Theorem 6. If (7) is true for some value of p then it is true for all larger values of p.

Proof. Suppose (7) is true for $p = p_0$. Multiplying (7) by two gives

$$2^{p_0+1} > n_c(2d - 2rem(2^{p_0}, d)).$$

From Theorem 4(a), $rem(2^{p_0+1}, d) \geq 2rem(2^{p_0}, d) - d$. Combining,

$$2^{p_0+1} > n_c(2d - (rem(2^{p_0+1}, d) + d)), \text{ or}$$
$$2^{p_0+1} > n_c(d - rem(2^{p_0+1}, d)).$$

Therefore (7) is true for $p = p_0 + 1$, and hence for all larger values.

Thus one could solve (7) by a binary search, although a simple linear search (starting with $p = W$) is probably preferable, because usually d is small, and small values of d give small values of p.

Proof that the algorithm is feasible

We must show that (7) always has a solution and that $0 \le m < 2^W$. (It is not necessary to show that $p \ge W$, because that is forced).

We will show that (7) always has a solution by getting an upper bound on p. As a matter of general interest we also derive a lower bound under the assumption that p is not forced to be at least W. To get these bounds on p, observe that for any positive integer x, there is a power of two greater than x and less than or equal to $2x$. Hence from (7),

$$n_c(d - rem(2^p, d)) \; < \; 2^p \; \le \; 2n_c(d - rem(2^p, d)).$$

Since $0 \le rem(2^p, d) \le d - 1$,

$$n_c + 1 \le 2^p \le 2n_c d. \tag{8}$$

From inequalities (4), $n_c \ge \max(2^{W-1} - d, d - 1)$. The lines $f_1(d) = 2^{W-1} - d$ and $f_2(d) = d - 1$ cross at $d = (2^{W-1} + 1)/2$. Hence $n_c \ge (2^{W-1} - 1)/2$. Since n_c is an integer, $n_c \ge 2^{W-2}$. Since $n_c, d \le 2^{W-1} - 1$, (8) becomes

$$2^{W-2} + 1 \le 2^p \le 2(2^{W-1} - 1)^2,$$

or

$$W - 1 \le p \le 2W - 2. \tag{9}$$

The lower bound $p = W - 1$ can occur, e.g. for $W = 32$, $d = 3$, but in that case we set $p = W$.

If p is not forced to equal W, then from (5) and (8),

$$\frac{n_c + 1}{d} < m < \frac{2n_c d}{d} \frac{n_c + 1}{n_c}.$$

Using (4b):

$$\frac{d - 1 + 1}{d} < m < 2(n_c + 1).$$

Since $n_c \le 2^{W-1} - 1$ (4a),

$$2 \le m \le 2^W - 1.$$

If p is forced to equal W, then from (5),

$$\frac{2^W}{d} < m < \frac{2^W}{d} \frac{n_c + 1}{n_c}.$$

Since $2 \le d \le 2^{W-1} - 1$ and $n_c \ge 2^{W-2}$,

$$\frac{2^W}{2^{W-1}-1} < m < \frac{2^W}{2}\frac{2^{W-2}+1}{2^{W-2}},$$
$$3 \le m \le 2^{W-1} + 1.$$

Hence in either case m is within limits for the code schema illustrated by the "divide by seven" example.

Proof that the product is correct

We must show that if p and m are calculated from (7) and (6), then (2a) and (2b) are satisfied.

Equation (6) and inequality (7) are easily seen to imply (5). (In the case that p is forced to be equal to W, (7) still holds, as shown by Theorem 6). In what follows we consider separately the following five ranges of values of n:

$$\begin{gathered} 0 \le n \le nc, \\ n_c + 1 \le n \le n_c + d - 1, \\ -n_c \le n \le -1, \\ -n_c - d + 1 \le n \le -n_c - 1, \quad \text{and} \\ n = -n_c - d. \end{gathered}$$

From (5), since m is an integer:

$$\frac{2^p}{d} < m \le \frac{2^p(n_c+1)-1}{dn_c}.$$

Multiplying by $n/2^p$, for $n \ge 0$ this becomes

$$\frac{n}{d} \le \frac{mn}{2^p} \le \frac{2^p n(n_c+1)-n}{2^p dn_c}, \quad \text{so that}$$
$$\left\lfloor \frac{n}{d} \right\rfloor \le \left\lfloor \frac{mn}{2^p} \right\rfloor \le \left\lfloor \frac{n}{d} + \frac{(2^p-1)n}{2^p dn_c} \right\rfloor.$$

For $0 \le n \le n_c$, $0 \le (2^p - 1)n/2^p dn_c < 1/d$, so by Theorem 3,

$$\left\lfloor \frac{n}{d} + \frac{(2^p-1)n}{2^p dn_c} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor.$$

Hence (2a) is satisfied in this case ($0 \le n \le n_c$).

For $n > n_c$, n is limited to the range

$$n_c + 1 \le n \le n_c + d - 1, \tag{10}$$

because $n \ge n_c + d$ contradicts the choice of n_c as the largest value of n such that $rem(n_c, d) = d - 1$ (alternatively, from (4a), $n \ge n_c + d$ implies $n \ge 2^{W-1}$).

From (5), for $n \ge 0$,

$$\frac{n}{d} < \frac{mn}{2^p} < \frac{n}{d}\,\frac{n_c+1}{n_c}.$$

By elementary algebra, this can be written

$$\frac{n}{d} \;<\; \frac{mn}{2^p} \;<\; \frac{n_c+1}{d} + \frac{(n-n_c)(n_c+1)}{dn_c}. \qquad (11)$$

From (10), $1 \le n - n_c \le d-1$, so

$$0 < \frac{(n-n_c)(n_c+1)}{dn_c} \le \frac{d-1}{d}\,\frac{n_c+1}{n_c}.$$

Since $n_c \ge d-1$ (4b) and $(n_c+1)/n_c$ has its maximum when n_c has its minimum,

$$0 \;<\; \frac{(n-n_c)(n_c+1)}{dn_c} \;\le\; \frac{d-1}{d}\,\frac{d-1+1}{d-1} \;=\; 1.$$

In (11), the term $(n_c+1)/d$ is an integer. The term $(n-n_c)(n_c+1)/dn_c$ is less than or equal to one. Therefore (11) becomes

$$\left\lfloor \frac{n}{d} \right\rfloor \le \left\lfloor \frac{mn}{2^p} \right\rfloor \le \frac{n_c+1}{d}.$$

For all n in the range (10), $\lfloor n/d \rfloor = (n_c+1)/d$. Hence (2a) is satisfied in this case $(n_c+1 \le n \le n_c+d-1)$.

For $n < 0$, from (5) we have, since m is an integer,

$$\frac{2^p+1}{d} \le m < \frac{2^p}{d}\,\frac{n_c+1}{n_c}.$$

Multiplying by $n/2^p$, for $n < 0$ this becomes

$$\frac{n}{d}\,\frac{n_c+1}{n_c} < \frac{mn}{2^p} \le \frac{n}{d}\,\frac{2^p+1}{2^p},$$

or

$$\left\lfloor \frac{n}{d}\,\frac{n_c+1}{n_c} \right\rfloor + 1 \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lfloor \frac{n}{d}\,\frac{2^p+1}{2^p} \right\rfloor + 1.$$

Using Theorem 1:

$$\left\lceil \frac{n(n_c+1) - dn_c + 1}{dn_c} \right\rceil + 1 \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n(2^p+1) - 2^p d + 1}{2^p d} \right\rceil + 1,$$

$$\left\lceil \frac{n(n_c+1)+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n(2^p+1)+1}{2^p d} \right\rceil .$$

Since $n + 1 \le 0$, the right inequality can be weakened, giving

$$\left\lceil \frac{n}{d} + \frac{n+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil . \tag{12}$$

For $-n_c \le n \le -1$,

$$\frac{-n_c+1}{dn_c} \le \frac{n+1}{dn_c} \le 0, \quad \text{or}$$
$$-\frac{1}{d} < \frac{n+1}{dn_c} \le 0.$$

Hence by Theorem 3,

$$\left\lceil \frac{n}{d} + \frac{n+1}{dn_c} \right\rceil = \left\lceil \frac{n}{d} \right\rceil ,$$

so that (2b) is satisfied in this case ($-n_c \le n \le -1$).

For $n < -n_c$, n is limited to the range

$$-n_c - d \le n \le -n_c - 1. \tag{13}$$

(From (4a), $n < -n_c - d$ implies that $n < -2^{W-1}$, which is impossible). Performing elementary algebraic manipulation of the left comparand of (12) gives

$$\left\lceil \frac{-n_c-1}{d} + \frac{(n+n_c)(n_c+1)+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil . \tag{14}$$

For $-n_c - d + 1 \le n \le -n_c - 1$,

$$\frac{(-d+1)(n_c+1)}{dn_c} + \frac{1}{dn_c} \le \frac{(n+n_c)(n_c+1)+1}{dn_c} \le \frac{-(n_c+1)+1}{dn_c} = -\frac{1}{d} .$$

The ratio $(n_c + 1)/n_c$ is a maximum when n_c is a minimum, i.e. $n_c = d - 1$. Therefore

$$\frac{(-d+1)(d-1+1)}{d(d-1)} + \frac{1}{dn_c} \le \frac{(n+n_c)(n_c+1)+1}{dn_c} < 0, \quad \text{or}$$
$$-1 < \frac{(n+n_c)(n_c+1)+1}{dn_c} < 0.$$

From (14), since $(-n_c - 1)/d$ is an integer and the quantity added to it is between 0 and -1,

$$\frac{-n_c-1}{d} \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil.$$

For n in the range $-n_c-d+1 \le n \le -n_c-1$,

$$\left\lceil \frac{n}{d} \right\rceil = \frac{-n_c-1}{d}.$$

Hence $\lfloor mn/2^p \rfloor + 1 = \lceil n/d \rceil$, i.e. (2b) is satisfied.

The last case, $n = -n_c - d$, can occur only for certain values of d. From (4a), $-n_c - d \le -2^{W-1}$, so if n takes on this value, we must have $n = -n_c - d = -2^{W-1}$, and hence $n_c = 2^{W-1} - d$. Therefore $rem(2^{W-1}, d) = rem(n_c + d, d) = d - 1$ (i.e. d divides $2^{W-1} + 1$).

For this case ($n = -n_c - d$), (7) has the solution $p = W - 1$ (the smallest possible value of p), because for $p = W - 1$,

$$\begin{aligned} n_c(d - rem(2^p, d)) &= (2^{W-1} - d)(d - rem(2^{W-1}, d)) \\ &= (2^{W-1} - d)(d - (d-1)) = 2^{W-1} - d < 2^{W-1} = 2^p. \end{aligned}$$

Then from (6),

$$m = \frac{2^{W-1} + d - rem(2^{W-1}, d)}{d} = \frac{2^{W-1} + d - (d-1)}{d} = \frac{2^{W-1}+1}{d}.$$

Therefore

$$\begin{aligned} \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 &= \left\lfloor \frac{2^{W-1}+1}{d} \frac{-2^{W-1}}{2^{W-1}} \right\rfloor + 1 = \left\lfloor \frac{-2^{W-1}-1}{d} \right\rfloor + 1 \\ &= \left\lceil \frac{-2^{W-1}-d}{d} \right\rceil + 1 = \left\lceil \frac{-2^{W-1}}{d} \right\rceil = \left\lceil \frac{n}{d} \right\rceil, \end{aligned}$$

so that (2b) is satisfied.

This completes the proof that if m and p are calculated from (6) and (7), then equations (2) hold for all admissible values of n.

Signed division by divisors ≤ -2

Dividing by a negative constant divisor must be extremely rare, but for completeness we will discuss it.

Since signed integer division satisfies $n \div (-d) = -(n \div d)$, it is adequate to generate code for $n \div |d|$ and follow it with an instruction to negate the quotient. (This does not give the correct result for $d = -2^{W-1}$, but for this and other negative powers of two, the code on page 3, followed by a negating instruction, may be used). It will not do to negate the dividend, because of the possibility that it is the maximum negative number.

It is possible, however, to avoid the negating instruction. The scheme is to compute

$$q = \left\lfloor \frac{mn}{2^p} \right\rfloor \quad \text{if } n \leq 0, \quad \text{and}$$
$$q = \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \quad \text{if } n > 0.$$

But adding one if $n > 0$ is awkward (because one cannot simply use the sign bit of n), so the code will instead add one if $q < 0$. This is equivalent because the multiplier m is negative (as will be seen).

The code to be generated is illustrated below for the case $d = -7$.

```
li    M,x'6DB6DB6D'     # Load magic number, -(2**34+5)/7 + 2**32.
mulhs q,M,n             # q = floor(M*n/2**32).
sub   q,q,n             # q = floor(M*n/2**32) - n.
srai  q,q,2             # q = floor(q/4).
sri   t,q,31            # Add 1 to q if
add   q,q,t             # q is negative (n is positive).

muli  t,q,-7            # Compute remainder from
sub   r,n,t             # r = n - q*(-7).
```

This code is the same as that for division by $+7$, except that it uses the negative of the multiplier for $+7$, and the *sri* of 31 must use q rather than n, as discussed above. The *subtract* will not overflow because the operands have the same sign. However, this scheme does not always work! Although the code above for $W = 32$, $d = -7$ is correct, the analogous alteration of the "divide by three" code to produce code to divide by minus three does not give the correct result for $W = 32$, $n = -2^{31}$.

Let us look at the situation more closely.

Given a word size $W \geq 3$ and a divisor d, $-2^{W-1} \leq d \leq -2$, we wish to find the least (in absolute value) integer m and integer p such that

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \qquad \text{for } -2^{W-1} \leq n \leq 0, \quad \text{and} \tag{15a}$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \quad \text{for } 1 \leq n < 2^{W-1}, \tag{15b}$$

with $-2^W \leq m \leq 0$ and $p \geq W$.

Proceeding similarly to the case of division by a positive divisor, let n_c be the most negative value of n such that $n_c = kd + 1$ for some integer k. n_c exists because one possibility is $n_c = d + 1$. It can be calculated from $n_c = \lfloor(-2^{W-1} - 1)/d\rfloor d + 1 = -2^{W-1} + rem(2^{W-1} + 1, d)$. n_c is one of the least $|d|$ admissible values of n, so

$$-2^{W-1} \le n_c \le -2^{W-1} - d - 1, \tag{16a}$$

and clearly

$$n_c \le d + 1. \tag{16b}$$

Since (15b) must hold for $n = -d$, and (15a) must hold for $n = n_c$, we obtain, analogous to (5),

$$\frac{2^p}{d}\frac{n_c - 1}{n_c} < m < \frac{2^p}{d}. \tag{17}$$

Since m is to be the greatest integer satisfying (17), it is the next integer less than $2^p/d$, i.e.

$$\boxed{m = \frac{2^p - d - rem(2^p, d)}{d}.} \tag{18}$$

Combining this with the left half of (17) and simplifying gives

$$\boxed{2^p > n_c(d + rem(2^p, d)).} \tag{19}$$

The proof that the algorithm suggested by (18) and (19) is feasible and that the product is correct is similar to that for a positive divisor, and will not be repeated. However, a difficulty arises in trying to prove that $-2^W \le m \le 0$. To prove this, consider separately the cases that d is the negative of a power of two, or some other number. For $d = -2^k$, it is easy to show that $n_c = -2^{W-1} + 1$, $p = W + k - 1$, and $m = -2^{W-1} - 1$ (which is within range). For d not of the form -2^k, it is straightforward to alter the earlier proof.

For which divisors is $m(-d) \ne -m(d)$?

By $m(d)$ we mean the multiplier corresponding to a divisor d. If $m(-d) = -m(d)$, code for division by a negative divisor can be generated by calculating the multiplier for $|d|$, negating it, and then generating code similar to that of the "divide by −7" case illustrated above.

By comparing (19) with (7) and (18) with (6), it can be seen that if the value of n_c for $-d$ is the negative of that for d, then $m(-d) = -m(d)$. Hence $m(-d) \ne -m(d)$ can occur only when the value of n_c calculated for the negative divisor is the maximum negative number, -2^{W-1}. Such divisors are the negatives of the factors of $2^{W-1} + 1$. These numbers are fairly rare, as illustrated by the factorings below (obtained from Scratchpad).

$$2^{15} + 1 = 3 \cdot 11 \cdot 331$$
$$2^{31} + 1 = 3 \cdot 715{,}827{,}883$$
$$2^{63} + 1 = 3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77{,}158{,}673{,}929$$

For *all* these factors, $m(-d) \neq m(d)$. Proof sketch: for $d > 0$ we have $n_c = 2^{W-1} - d$. Since $rem(2^{W-1}, d) = d - 1$, (7) is satisfied by $p = W - 1$ and hence also by $p = W$. However, for $d < 0$, we have $n_c = -2^{W-1}$ and $rem(2^{W-1}, d) = |d| - 1$. Hence (19) is not satisfied for $p = W - 1$ nor for $p = W$, so $p > W$.

Incorporation into a compiler

For a compiler to change division by a constant into a multiplication, it must compute the magic number M and the shift amount s, given a divisor d. The straightforward computation is to evaluate (7) or (19) for $p = W, W + 1, \ldots$ until it is satisfied. Then, m is calculated from (6) or (18). M is simply a reinterpretation of m as a signed integer, and $s = p - W$.

The scheme described below handles positive and negative d with only a little extra code, and it avoids double precision arithmetic.

Recall that n_c is given by

$$n_c = \begin{cases} 2^{W-1} - rem(2^{W-1}, d) - 1, & \text{if } d > 0, \\ -2^{W-1} + rem(2^{W-1} + 1, d), & \text{if } d < 0, \end{cases}$$

Hence $|n_c|$ can be computed from:

$$t = 2^{W-1} + \begin{cases} 0, & \text{if } d > 0, \\ 1, & \text{if } d < 0, \end{cases}$$
$$|n_c| = t - 1 - rem(t, |d|).$$

The remainder must be evaluated using unsigned division, because of the magnitude of the arguments. We have written $rem(t, |d|)$ rather than the equivalent $rem(t, d)$, to emphasize that the program must deal with two positive (and unsigned) arguments.

From (7) and (19), p can be calculated from

$$2^p > |n_c|(|d| - rem(2^p, |d|)). \tag{20}$$

and then $|m|$ can be calculated from (c.f. (6) and (18)):

$$|m| = \frac{2^p + |d| - rem(2^p, |d|)}{|d|}. \tag{21}$$

Direct evaluation of $rem(2^p, |d|)$ in (20) requires "long division" (dividing a $2W$-bit dividend by a W-bit divisor, giving a W-bit result), and in fact it must be *unsigned* long division. However, there is a way to solve (20), and in fact to do all the calculations, that avoids long division and can easily be implemented in a conventional HLL using only W-bit arithmetic. However, we do need unsigned division and unsigned comparisons.

We can calculate $rem(2^p, |d|)$ incrementally, by initializing two variables q and r to the quotient and remainder of 2^p divided by $|d|$ with $p = W - 1$, and then updating q and r as p increases.

As the search progresses, i.e. when p is incremented by one, q and r are updated from (see Theorem 4(a)):

```
q = 2*q;
r = 2*r;
if (r >= abs(d)) {
   q = q + 1;
   r = r - abs(d);}
```

The left half of inequality (5) and the right half of (17), together with the bounds proved for m, imply that $q = \lfloor 2^p/|d| \rfloor < 2^W$, so q is representable as a W-bit unsigned number. Also, $0 \leq r < |d|$, so r is representable as a W-bit signed or unsigned number. [Caution: The intermediate result $2r$ can exceed $2^{W-1} - 1$, so r should be unsigned and the comparison above should also be unsigned.]

Next, calculate $\delta = |d| - r$. Both terms of the subtraction are representable as W-bit unsigned integers, and the result is also $(1 \leq \delta \leq |d|)$, so there is no difficulty here.

To avoid the long multiplication of (20), rewrite it as

$$\frac{2^p}{|n_c|} > \delta.$$

The quantity $2^p/|n_c|$ is representable as a W-bit unsigned integer (similarly to (8), from (20) it can be shown that $2^p \leq 2|n_c| \cdot |d|$ and, for $d = -2^{W-1}$, $n_c = -2^{W-1} + 1$ and $p = 2W - 2$, so that $2^p/|n_c| = 2^{2W-2}/(2^{W-1} - 1) < 2^W$ for $W \geq 3$), and it is easily calculated incrementally (as p increases) in the same manner as for $rem(2^p, |d|)$. The comparison should be unsigned, for the case $2^p/|n_c| \geq 2^{W-1}$ (which can occur, for large d).

To compute m, we need not evaluate (21) directly (which would require long division). Observe that

$$\frac{2^p + |d| - rem(2^p, |d|)}{|d|} = \left\lfloor \frac{2^p}{|d|} \right\rfloor + 1 = q + 1.$$

The loop closure test $2^p/|n_c| > \delta$ is awkward to evaluate. The quantity $2^p/|n_c|$ is available only in the form of a quotient q_1 and a remainder r_1. $2^p/|n_c|$ may or may not be an integer (it is an integer only for $d = 2^{W-2} + 1$ and a few negative values of d). The test $2^p/|n_c| \leq \delta$ may be coded as

$$q_1 < \delta \ \mid \ (q_1 = \delta \ \& \ r_1 = 0).$$

The complete procedure for computing m and s from d is shown below, coded in the C programming language, for $W = 32$. There are a few places where overflow can occur, but the correct result is obtained if overflow is ignored.

To use the results of this program, the compiler should generate the *li* and *mulhs* instructions, generate the *add* if $d > 0$ and $m < 0$, or the *subtract* if $d < 0$ and $m > 0$, and generate the *srai* if $s > 0$. Then, the *sri* and final *add* must be generated.

For $W = 32$, handling a negative divisor may be avoided by simply returning a precomputed result for $d = 3$ and $d = 715{,}827{,}883$, and using $m(-d) = -m(d)$ for other negative divisors. However, that program would not be significantly shorter, if at all, than the one given below.

```
struct {int m;                  /* Magic number */
        int s;                  /* and shift amount */
       } mag;                   /* are returned here. */

magic(d)
int d;                          /* Must have 2 <= d <= 2**31-1 */
                                /* or   -2**31 <= d <= -2. */
{
   int p;
   unsigned int ad, anc, delta, q1, r1, q2, r2, t;
   unsigned int two31 = 2147483648;  /* Constant, 2**31. */

   ad = abs(d);
   t = two31 + ((unsigned int)d >> 31);
   anc = t - 1 - t%ad;          /* Absolute value of nc. */
   p = 31;                      /* Initialize p. */
   q1 = two31/anc;              /* Initialize 2**p/|nc|. */
   r1 = two31 - q1*anc;         /* Initialize rem(2**p, |nc|). */
   q2 = two31/ad;               /* Initialize q2 = 2**p/|d|. */
   r2 = two31 - q2*ad;          /* Initialize r2 = rem(2**p, |d|). */
loop:
      p = p + 1;
      q1 = 2*q1;                /* Update quotient and */
      r1 = 2*r1;                /* remainder of 2**p/|nc|. */
      if (r1 >= anc) {          /* (Must be an unsigned */
         q1 = q1 + 1;           /* comparison here). */
         r1 = r1 - anc;
         }
      q2 = 2*q2;                /* Update quotient and */
      r2 = 2*r2;                /* remainder of 2**p/|d|. */
      if (r2 >= ad) {           /* (Must be an unsigned */
         q2 = q2 + 1;           /* comparison here). */
         r2 = r2 - ad;
         }
      delta = ad - r2;
      if (q1 < delta || (q1 == delta && r1 == 0)) goto loop;

   mag.m = q2 + 1;              /* Give multiplier and */
   if (d < 0) mag.m = -mag.m;
   mag.s = p - 32;              /* shift amount to caller. */
}
```

Figure 1. Computing the magic number for signed division

Theorem 7. The least multiplier m is odd if p is not forced to equal W.

Proof. Assume that equations (2) are satisfied with least (not forced) integer p, and m even. Then clearly m could be divided by two and p could be decreased by one, and equations (2) would still be satisfied. This contradicts the assumption that p is minimal.

Uniqueness

The magic number for a given divisor is sometimes unique (e.g., for $W = 32$, $d = 7$), but often it is not. In fact, experiment indicates that it is usually not unique. For example, for $W = 32$, $d = 6$, there are four magic numbers:

$$\begin{array}{lrll} M = & 715{,}827{,}883 & ((2^{32}+2)/6), & s = 0 \\ M = & 1{,}431{,}655{,}766 & ((2^{32}+2)/3), & s = 1 \\ M = & -\,1{,}431{,}655{,}765 & ((2^{33}+1)/3 - 2^{32}), & s = 2 \\ M = & -\,1{,}431{,}655{,}764 & ((2^{33}+4)/3 - 2^{32}), & s = 2. \end{array}$$

However, there is the following uniqueness property:

Theorem 8. For a given divisor d, there is only one multiplier m having the minimal value of p, if p is not forced to equal W.

Proof. First consider the case $d > 0$. The difference between the upper and lower limits of inequality (5) is $2^p/dn_c$. We have already proved (8) that if p is minimal, then $2^p/dn_c \le 2$. Therefore there can be at most two values of m satisfying (5). Let m be the smaller of these values, given by (6); then $m + 1$ is the other.

Let p_0 be the least value of p for which $m + 1$ satisfies the right half of (5) (p_0 is not forced to equal W). Then

$$\frac{2^{p_0} + d - rem(2^{p_0}, d)}{d} + 1 < \frac{2^{p_0}}{d}\,\frac{n_c + 1}{n_c}.$$

This simplifies to:

$$2^{p_0} > n_c(2d - rem(2^{p_0}, d)).$$

Dividing by 2:

$$2^{p_0 - 1} > n_c(d - \frac{1}{2}\,rem(2^{p_0}, d)).$$

Since $rem(2^{p_0}, d) \le 2rem(2^{p_0 - 1}, d)$ (Theorem 4(a)),

$$2^{p_0 - 1} > n_c(d - rem(2^{p_0 - 1}, d)),$$

contradicting the assumption that p_0 is minimal.

The proof for $d < 0$ is similar and will not be given.

The divisors with the best programs

The program for $d = 3$ and $W = 32$ is particularly short, because there is no *add* or *srai* after the *mulhs*. What other divisors have this short program?

We consider only positive divisors. We wish to find integers m and p that satisfy equations (2) on page 8, and for which $p = W$ and $0 \le m < 2^{W-1}$. Since any integers m and p that satisfy equations (2) must also satisfy (5), it suffices to find those divisors d for which (5) has a solution with $p = W$ and $0 \le m < 2^{W-1}$. All solutions of (5) with $p = W$ are given by

$$m = \frac{2^W + kd - rem(2^W, d)}{d}, \quad k = 1, 2, 3, \dots.$$

Combining this with the right half of (5) and simplifying gives

$$rem(2^W, d) > kd - \frac{2^W}{n_c}. \tag{22}$$

The weakest restriction on $rem(2^W, d)$ is with $k = 1$ and n_c at its minimal value of 2^{W-2} (see page 10). Hence we must have

$$rem(2^W, d) > d - 4,$$

i.e. d divides $2^W + 1$, $2^W + 2$, or $2^W + 3$.

Now let us see which of these factors actually have optimal programs.

If d divides $2^W + 1$, then $rem(2^W, d) = d - 1$. Then a solution of (7) is $p = W$, because the inequality becomes

$$2^W > n_c(d - (d-1)) = n_c$$

which is obviously true, because $n_c < 2^{W-1}$. Then in the calculation of m we have

$$m = \frac{2^W + d - (d-1)}{d} = \frac{2^W + 1}{d}$$

which is less than 2^{W-1} for $d \ge 3$ ($d \ne 2$ because d divides $2^W + 1$). Hence all the factors of $2^W + 1$ have optimal programs.

Similarly, if d divides $2^W + 2$, then $rem(2^W, d) = d - 2$. Again, a solution of (7) is $p = W$, because the inequality becomes

$$2^W > n_c(d - (d-2)) = 2n_c$$

which is obviously true. Then in the calculation of m we have

$$m = \frac{2^W + d - (d-2)}{d} = \frac{2^W + 2}{d}$$

which exceeds $2^{W-1}-1$ for $d=2$, but which is less than or equal to $2^{W-1}-1$ for $W \geq 3, d \geq 3$ (the case $W=3$ and $d=3$ does not occur, because 3 is not a factor of $2^3+2=10$). Hence all factors of 2^W+2, except for 2 and the cofactor of 2, have optimal programs. (The cofactor of 2 is $(2^W+2)/2$, which is not representable as a W-bit signed integer).

If d divides 2^W+3, the following argument shows that d does not have an optimal program. Since $rem(2^W, d) = d-3$, inequality (22) implies that we must have

$$n_c < \frac{2^W}{kd-d+3}$$

for some $k = 1, 2, 3, \ldots$. The weakest restriction is with $k=1$, so we must have $n_c < 2^W/3$.

From (4a), $n_c \geq 2^{W-1} - d$, or $d \geq 2^{W-1} - n_c$. Hence it is necessary that

$$d > 2^{W-1} - \frac{2^W}{3} = \frac{2^W}{6}.$$

Also, since 2, 3, and 4 do not divide 2^W+3, the smallest possible factor of 2^W+3 is 5. Hence the largest possible factor is $(2^W+3)/5$. Thus if d divides 2^W+3 and d has an optimal program, it is necessary that

$$\frac{2^W}{6} < d \leq \frac{2^W+3}{5}.$$

Taking reciprocals of this with respect to 2^W+3 shows that the cofactor of d, $(2^W+3)/d$, has the limits

$$5 \leq \frac{2^W+3}{d} < \frac{(2^W+3)\cdot 6}{2^W} = 6 + \frac{18}{2^W}.$$

For $W \geq 5$, this implies that the only possible cofactors are 5 and 6. For $W < 5$, it is easily verified that there are no factors of 2^W+3. Since 6 cannot be a factor of 2^W+3, the only possibility is 5. Therefore the only possible factor of 2^W+3 that might have an optimal program is $(2^W+3)/5$.

For $d = (2^W+3)/5$,

$$n_c = \left\lfloor \frac{2^{W-1}}{(2^W+3)/5} \right\rfloor \left(\frac{2^W+3}{5} \right) - 1.$$

For $W \geq 4$,

$$2 < \frac{2^{W-1}}{(2^W+3)/5} < 2.5,$$

so

$$n_c = 2\left(\frac{2^W+3}{5} \right) - 1.$$

This exceeds $2^W/3$, so $d = (2^W + 3)/5$ does not have an optimal program. Since for $W < 4$ there are no factors of $2^W + 3$, we conclude that no factors of $2^W + 3$ have optimal programs.

In summary, all the factors of $2^W + 1$ and of $2^W + 2$, except for 2 and $(2^W + 2)/2$, have optimal programs, and no other numbers do. Furthermore, the above proof shows that algorithm *magic* (Figure 1 on page 20) always produces the optimal program when it exists.

Let us consider the specific cases $W = 16$, 32, and 64. The relevant factorizations are shown below.

$$2^{16} + 1 = 65537 \text{ (prime)} \qquad 2^{32} + 1 = 641 \cdot 6{,}700{,}417$$
$$2^{16} + 2 = 2 \cdot 3^2 \cdot 11 \cdot 331 \qquad 2^{32} + 2 = 2 \cdot 3 \cdot 715{,}827{,}883$$

$$2^{64} + 1 = 274{,}177 \cdot 67{,}280{,}421{,}310{,}721$$
$$2^{64} + 2 = 2 \cdot 3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77{,}158{,}673{,}929$$

Hence we have the results that for $W = 16$, there are 20 divisors that have optimal programs. The ones less than 100 are 3, 6, 9, 11, 18, 22, 33, 66, and 99.

For $W = 32$, there are six such divisors: 3, 6, 641, 6,700,417, 715,827,883, and 1,431,655,766.

For $W = 64$, there are 126 such divisors. The ones less than 100 are 3, 6, 9, 18, 19, 27, 38, 43, 54, 57, and 86.

Unsigned division

Unsigned division by a power of two is of course implemented by a single *shift right logical* instruction.

Unsigned divide by three

For a non-power of two, let us first consider unsigned division by three on a 32-bit machine. Since the dividend n can now be as large as $2^{32} - 1$, the multiplier $(2^{32} + 2)/3$ is inadequate, because the error term $2n/3 \cdot 2^{32}$ (see "divide by three" example above) can exceed 1/3. However, the multiplier $(2^{33} + 1)/3$ is adequate. The code is:

```
li    M,x'AAAAAAAB'     # Load magic number, (2**33+1)/3.
mulhu q,M,n             # q = floor(M*n/2**32).
sri   q,q,1

muli  t,q,3             # Compute remainder from
sub   r,n,t             # r = n - q*3.
```

An instruction for long unsigned multiply is required, which we show above as *mulhu.*

To see that the code is correct, observe that it computes

$$q = \left\lfloor \frac{2^{33}+1}{3} \frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{n}{3 \cdot 2^{33}} \right\rfloor .$$

For $0 \le n < 2^{32}$, $0 \le n/3 \cdot 2^{33} < 1/3$, so by Theorem 3, $q = \lfloor n/3 \rfloor$.

In computing the remainder, the *muli* can overflow if we regard the operands as signed numbers, but it does not overflow if we regard them and the result as unsigned. Also the *subtract* cannot overflow, because the result is in the range 0 to 2, so the remainder is correct.

Unsigned divide by seven

For unsigned division by seven on a 32-bit machine, the multipliers $(2^{32} + 3)/7$, $(2^{33} + 6)/7$, and $(2^{34} + 5)/7$ are all inadequate because they give too large an error term. The multiplier $(2^{35} + 3)/7$ is acceptable, but it's too large to represent in a 32-bit unsigned word. We can multiply by this large number by multiplying by $(2^{35} + 3)/7 - 2^{32}$ and then correcting the product by inserting an *add.* The code is:

```
li    M,x'24924925'     # Load magic number, (2**35+3)/7 - 2**32.
mulhu q,M,n             # q = floor(M*n/2**32).
add   q,q,n             # Can overflow (sets carry).
srxi  q,q,3             # Shift right with carry bit.

muli  t,q,7             # Compute remainder from
sub   r,n,t             # r = n - q*7.
```

Here we have a problem: the *add* can overflow. To allow for this, we have invented the new instruction *shift right extended immediate* (*srxi*), which treats the carry from the add and the 32 bits of register q as a single 33-bit quantity, and shifts it right with zero-fill. On the Motorola 68000 family, this can be done with two instructions: *rotate with extend* right one position, followed by a logical right shift of three (*roxr* actually uses the X bit, but the *add* sets that the same as the carry bit). On most machines, it will take more. For example, on the IBM RISC System/6000, it takes three instructions: clear rightmost three bits of q, add carry to q, and rotate right three positions.

With *srxi* implemented somehow, the code above computes:

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{35}+3}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) / 2^3 \right\rfloor = \left\lfloor \frac{n}{7} + \frac{3n}{7 \cdot 2^{35}} \right\rfloor.$$

For $0 \le n < 2^{32}$, $0 \le 3n/7 \cdot 2^{35} < 1/7$, so by Theorem 3, $q = \lfloor n/7 \rfloor$.

Unsigned division by divisors ≥ 1

Given a word size $W \geq 1$ and a divisor d, $1 \leq d < 2^W$, we wish to find the least integer m and integer p such that

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \quad \text{for } 0 \leq n < 2^W, \tag{23}$$

with $0 \leq m < 2^{W+1}$ and $p \geq W$.

In the unsigned case, the magic number M is given by

$$M = \begin{cases} m, & \text{if } 0 \leq m < 2^W, \\ m - 2^W, & \text{if } 2^W \leq m < 2^{W+1}. \end{cases}$$

Since (23) must hold for $n = d$, $\lfloor md/2^p \rfloor = 1$, or

$$\frac{md}{2^p} \geq 1. \tag{24}$$

As in the signed case, let n_c be the largest value of n such that $rem(n_c, d) = d - 1$. It can be calculated from $n_c = \lfloor 2^W/d \rfloor d - 1 = 2^W - rem(2^W, d) - 1$. Then

$$2^W - d \leq n_c \leq 2^W - 1, \tag{25a}$$

and

$$n_c \geq d - 1. \tag{25b}$$

These imply that $n_c \geq 2^{W-1}$.

Since (23) must hold for $n = n_c$,

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d},$$

or

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}.$$

Combining this with (24):

$$\frac{2^p}{d} \leq m < \frac{2^p}{d} \frac{n_c + 1}{n_c}. \tag{26}$$

Since m is to be the least integer satisfying (26), it is the next integer greater than or equal to $2^p/d$, i.e.

$$\boxed{m = \frac{2^p + d - 1 - rem(2^p - 1, d)}{d}.} \tag{27}$$

Combining this with the right half of (26) and simplifying gives:

$$\boxed{2^p > n_c(d - 1 - rem(2^p - 1, d)).} \tag{28}$$

The algorithm (unsigned)

Thus the algorithm is to find by trial and error the least $p \geq W$ satisfying (28). Then m is calculated from (27). This is the smallest possible value of m satisfying (23) with $p \geq W$. As in the signed case, if (28) is true for some value of p then it is true for all larger values of p. The proof is essentially the same as that of Theorem 6, except Theorem 4(b) is used instead of Theorem 4(a).

Proof that the algorithm is feasible (unsigned)

We must show that (28) always has a solution and that $0 \leq m < 2^{W+1}$.

Since for any nonnegative integer x there is a power of two greater than x and less than or equal to $2x + 1$, from (28),

$$n_c(d - 1 - rem(2^p - 1, d)) \; < \; 2^p \; \leq \; 2n_c(d - 1 - rem(2^p - 1, d)) + 1.$$

Since $0 \leq rem(2^p - 1, d) \leq d - 1$,

$$1 \leq 2^p \leq 2n_c(d - 1) + 1. \tag{29}$$

Since n_c, $d \leq 2^W - 1$, this becomes

$$1 \leq 2^p \leq 2(2^W - 1)(2^W - 2) + 1,$$

or

$$0 \leq p \leq 2W. \tag{30}$$

Thus (28) always has a solution.

If p is not forced to equal W, then from (26) and (29):

$$\begin{aligned} \frac{1}{d} \leq m &< \frac{2n_c(d-1)+1}{d}\frac{n_c+1}{n_c}, \\ 1 \leq m &< \frac{2d - 2 + 1/n_c}{d}(n_c + 1), \\ 1 \leq m &< 2(n_c + 1) \leq 2^{W+1}. \end{aligned}$$

If p is forced to equal W, then from (26),

$$\frac{2^W}{d} \leq m < \frac{2^W}{d}\frac{n_c + 1}{n_c}.$$

Since $1 \leq d \leq 2^W - 1$ and $n_c \geq 2^{W-1}$,

$$\frac{2^W}{2^W - 1} \leq m < \frac{2^W}{1} \frac{2^{W-1} + 1}{2^{W-1}},$$
$$2 \leq m \leq 2^W + 1.$$

Hence in either case m is within limits for the code schema illustrated by the "unsigned divide by seven" example.

Proof that the product is correct (unsigned)

We must show that if p and m are calculated from (28) and (27), then (23) is satisfied.

Equation (27) and inequality (28) are easily seen to imply (26). (26) is nearly the same as (5), and the remainder of the proof is nearly identical to that for signed division with $n \geq 0$ (page 11).

Incorporation into a compiler (unsigned)

There is a difficulty in implementing an algorithm based on direct evaluation of the expressions used in this proof. Although $p \leq 2W$, which is proved above, the case $p = 2W$ can occur (e.g., for $d = 2^W - 2$ with $W \geq 4$). When $p = 2W$, it is difficult to calculate m, because the dividend in (27) does not fit in a $2W$-bit word.

However, it can be implemented by the "incremental division and remainder" technique of algorithm *magic*. The algorithm is given below, for $W = 32$. It passes back an indicator a, that tells whether or not to generate an *add* instruction. (In the case of signed division, the caller recognizes this by m and d having opposite signs).

Some key points in understanding this algorithm are

- Unsigned overflow can occur at several places and should be ignored.
- $n_c = 2^W - rem(2^W, d) - 1 = (2^W - 1) - rem(2^W - d, d)$.
- The quotient and remainder of dividing 2^p by n_c cannot be updated in the same way as is done in algorithm *magic*, because here the quantity $2r_1$ can overflow. Hence the algorithm has the test "`if (r1 >= nc - r1)`" whereas "`if (2*r1 >= nc)`" would be more natural. A similar remark applies to computing the quotient and remainder of $2^p - 1$ divided by d.
- $0 \leq \delta \leq d - 1$, so δ is representable as a 32-bit unsigned number.
- $m = (2^p + d - 1 - rem(2^p - 1, d))/d = \lfloor (2^p - 1)/d \rfloor + 1 = q_2 + 1$.
- The subtraction of 2^W when the multiplier m exceeds $2^W - 1$ is not explicit in the program; it occurs if the computation of q_2 overflows..
- The "add" indicator, *magu.a*, cannot be set by a straightforward comparison of m to 2^{32}, or of q_2 to $2^{32} - 1$, because of overflow. Instead, the program tests q_2 before overflow can occur. If q_2 ever gets as large as $2^{32} - 1$, so that m will be greater than or equal to 2^{32}, then *magu.a* is set equal to one. If q_2 stays below $2^{32} - 1$, then *magu.a* is left at its initial value of zero.
- Inequality (28) is equivalent to $2^p/n_c > \delta$.
- The loop test needs the condition "$p < 64$" because without it, overflow of q_1 would cause the program to loop too many times, giving incorrect results.

```
struct {unsigned int M;         /* Magic number, */
        int a;                  /* "add" indicator, */
        int s;                  /* and shift amount */
       } magu;                  /* are returned here. */

magicu(d)
unsigned int d;                 /* Must have 1 <= d <= 2**32-1. */
{
   int p;
   unsigned int nc, q1, r1, q2, r2, delta;

   magu.a = 0;                  /* Initialize "add" indicator. */
   nc = -1 - (-d)%d;
   p = 31;                      /* Initialize p. */
   q1 = 0x80000000/nc;          /* Initialize 2**p/nc. */
   r1 = 0x80000000 - q1*nc;     /* Initialize rem(2**p, nc). */
   q2 = 0x7fffffff/d;           /* Initialize q2 = (2**p - 1)/d. */
   r2 = 0x7fffffff - q2*d;      /* Initialize r2 = rem(2**p - 1, d). */
loop:
      p = p + 1;
      if (r1 >= nc - r1) {      /* Update quotient and */
         q1 = 2*q1 + 1;         /* remainder of 2**p/nc. */
         r1 = 2*r1 - nc;}
      else {
         q1 = 2*q1;
         r1 = 2*r1;}
      if (r2 + 1 >= d - r2) {   /* Update quotient and */
                                /* remainder of (2**p - 1)/d. */
         if (q2 >= 0x7fffffff) magu.a = 1;
         q2 = 2*q2 + 1;
         r2 = 2*r2 + 1 - d;}
      else {
         if (q2 >= 0x80000000) magu.a = 1;
         q2 = 2*q2;
         r2 = 2*r2 + 1;}
      delta = d - 1 - r2;
      if (p < 64 && (q1 < delta || (q1==delta && r1==0))) goto loop;

   magu.M = q2 + 1;             /* Give magic number and */
   magu.s = p - 32;             /* shift amount to caller */
                                /* (magu.a was set above). */
}
```

Figure 2. Computing the magic number for unsigned division

Miscellaneous topics (unsigned)

Theorem 7u. The least multiplier m is odd if p is not forced to equal W.

Theorem 8u. For a given divisor d, there is only one multiplier m having the minimal value of p, if p is not forced to equal W.

The proofs of these theorems follow very closely the corresponding proofs for signed division.

The divisors with the best programs (unsigned)

For unsigned division, to find the divisors (if any) with optimal programs of two instructions to obtain the quotient (*li, mulhu*), an analysis may be done similar to that of the signed case (see page 22). The result is that such divisors are the factors of 2^W or $2^W + 1$, except for $d = 1$. For the common word sizes, this leaves very few nontrivial divisors that have optimal programs for unsigned division. For $W = 16$, there are none. For $W = 32$, there are only two: 641 and 6,700,417. For $W = 64$, again there are only two: 274,177 and 67,280,421,310,721.

The case $d = 2^k$, $k = 1, 2, \ldots$, deserves special mention. In this case, algorithm *magicu* produces $p = W$ (forced), $m = 2^{32-k}$. This is the minimal value of m, but it is not the minimal value of M. Better code results if $p = W + k$ is used, if sufficient simplifications are done. Then, $m = 2^W$, $M = 0$, $a = 1$, and $s = k$. The generated code involves a multiplication by zero and can be simplified to a single *shift right k* instruction. As a practical matter, divisors that are a power of two would probably be special-cased without using *magicu*. [This phenomenon does not occur for signed division, because for signed division m cannot be a power of two. Proof: For $d > 0$, inequalities (4b) and (5) imply that $d - 1 < 2^p/m < d$. Therefore $2^p/m > 1$ and $2^p/m$ cannot be an integer. For $d < 0$, the result follows similarly from (16b) and (17).]

For unsigned division, the code when $m \geq 2^W$ is considerably worse than that when $m < 2^W$, if *srxi* is hard to implement. Hence it is of interest to have some idea of how often the large multipliers arise. For $W = 32$, among the numbers less than 100, there are 31 "bad" divisors: 1, 7, 14, 19, 21, 27, 28, 31, 35, 37, 38, 39, 42, 45, 53, 54, 55, 56, 57, 62, 63, 70, 73, 74, 76, 78, 84, 90, 91, 95, and 97.

Using signed in place of unsigned multiply, and the reverse

If your machine does not have *mulhu*, but it does have *mulhs* (or signed long multiplication), there is a trick (M&S) that might make our method of doing unsigned division by a constant still useful.

To get unsigned multiplication from signed, let x and y denote the two unsigned W-bit numbers being multiplied. Then the machine, when it does signed multiplication, interprets x correctly if its most significant bit x_0 is zero, but it interprets it as $x - 2^W$ if the most significant bit is one. Operand y is interpreted similarly, so the machine forms the product

$$(x - 2^W x_0)(y - 2^W y_0) = xy - 2^W(x_0 y + y_0 x) + 2^{2W} x_0 y_0.$$

To get the desired result xy, we must add to this the quantity $2^W(x_0y + y_0x) - 2^{2W}x_0y_0$. Since the maximum value of xy is less than 2^{2W}, we can perform the additions modulo 2^{2W}, which means that we can safely ignore the last term.

By implementing the term x_0y with *shift right algebraic 31* and *and*, *mulhu* can be implemented with *mulhs* and six additional simple, branch-free, instructions. But a further simplification is possible. Since the compiler can test the most significant bit of the magic number, it can generate code such as the following for the operation "*mulhu q,m,n*," where *t* denotes a temporary register.

```
      m0 = 0                m0 = 1
    mulhs q,m,n           mulhs q,m,n
    srai  t,n,31          srai  t,n,31
    and   t,t,m           and   t,t,m
    add   q,q,t           add   t,t,n
                          add   q,q,t
```

Accounting for the other instructions used with *mulhu*, this uses a total of six to eight instructions to obtain the quotient of unsigned division by a constant, on a machine that does not have unsigned multiply.

This trick may be inverted, to get *mulhs* in terms of *mulhu*. The code is the same as that above except the *mulhs* is changed to *mulhu* and the final *add* in each column is changed to *subtract*.

A simpler algorithm (unsigned)

Dropping the requirement that the magic number be minimal yields a simpler algorithm. In place of (28) we can use

$$2^p \geq 2^W(d - 1 - rem(2^p - 1, d)). \tag{31}$$

and then use (27) to compute m, as before.

It should be clear that this algorithm is formally correct (i.e. that the value of m computed does satisfy equation (23)), because its only difference from the previous algorithm is to compute a value of p that, for some values of d, is unnecessarily large. It can be proved that the value of m computed from (31) and (27) is less than 2^{W+1}. We omit the proof and simply give the algorithm below.

Alverson gives a much simpler algorithm, discussed on page 35, but it gives somewhat large values for m. The point of algorithm *magicu2* below is that it nearly always gives the minimal value for m when $d \leq 2^{W-1}$. For $W = 32$, the smallest divisor for which *magicu2* does not give the minimal multiplier is d = 102,807, for which *magicu* calculates m = 2,737,896,999, and *magicu2* calculates m = 5,475,793,997.

There is an analog of *magicu2* for signed division by positive divisors, but it does not work out very well for signed division by arbitrary divisors.

```
struct {unsigned int M;        /* Magic number, */
        int a;                 /* "add" indicator, */
        int s;                 /* and shift amount */
       } magu;                 /* are returned here. */

magicu2(d)
unsigned int d;                /* Must have 1 <= d <= 2**32-1. */
{
   int p;
   unsigned int p32, q, r, delta;

   magu.a = 0;                 /* Initialize "add" indicator. */
   p = 31;                     /* Initialize p. */
   q = 0x7fffffff/d;           /* Initialize q = (2**p - 1)/d. */
   r = 0x7fffffff - q*d;       /* Initialize r = rem(2**p - 1, d). */
loop:
      p = p + 1;
      if (p == 32) p32 = 1;    /* Set p32 = 2**(p-32). */
      else p32 = 2*p32;
      if (r + 1 >= d - r) {    /* Update quotient and */
                               /* remainder of (2**p - 1)/d. */
         if (q >= 0x7fffffff) magu.a = 1;
         q = 2*q + 1;
         r = 2*r + 1 - d;}
      else {
         if (q >= 0x80000000) magu.a = 1;
         q = 2*q;
         r = 2*r + 1;}
      delta = d - 1 - r;
      if (p < 64 && p32 < delta) goto loop;

   magu.M = q + 1;             /* Give magic number and */
   magu.s = p - 32;            /* shift amount to caller */
                               /* (magu.a was set above). */
}
```

Figure 3. Simplified algorithm for computing the magic number, unsigned division

Related Work

The method described here has been in use in the AIX XL family of compilers for the IBM RISC System/6000 since their introduction in 1990. Only signed division is turned into multiplication, and only for a few small divisors and their multiples by a power of two. The magic number and shift amount are determined by table lookup, because the compiler developers (including the present writer) did not then know that this method works for all divisors, and that the program for computing the magic number is fairly simple. The lookup argument is the divisor reduced by dividing it by 2^k where k is the number of trailing zeros in the divisor. The shift amount found in the table is then increased by k.

This procedure, incidently, does not always give *minimal* magic numbers. The smallest positive divisor for which it fails in this respect for $W = 32$ is $d = 334{,}972$, for which it computes $m = 3{,}361{,}176{,}179$ and $s = 18$. However, the minimal magic number for $d = 334{,}972$ is $m = 840{,}294{,}045$, with $s = 16$. The procedure also fails to give the minimal magic number for $d = -6$. In both these cases, output code quality is affected.

Alverson (Alv) is the first known to the author to state that the method described here works with complete accuracy for all divisors. Using our notation, his method for unsigned integer division by d is to set the shift amount $p = W + \lceil \log_2 d \rceil$, and the multiplier $m = \lceil 2^p/d \rceil$, and then do the division by $n \div d = \lfloor mn/2^p \rfloor$ (i.e. multiply and shift right). He proves that the multiplier m is less than 2^{W+1}, and that the method gets the exact quotient for all n expressible in W bits.

Alverson's method is a simpler variation of ours in that it doesn't require trial and error to determine p, and is thus more suitable for building in hardware, which is his primary interest. However, his multiplier m is always greater than or equal to 2^W, and thus for the software application always gives the code illustrated by the "divide by seven" example (i.e. always has the *add* and *srxi* instructions). Since most small divisors can be handled with a multiplier less than 2^W, it seems worthwhile to look for these cases.

For signed division, Alverson suggests finding the multiplier for $|d|$ and a wordlength of $W - 1$ (then $2^{W-1} \leq m < 2^W$), multiplying the dividend by it, and negating the result if the operands have opposite signs. (The multiplier must be such that it gives the correct result when the dividend is 2^{W-1}, the absolute value of the maximum negative number). It seems possible that this suggestion might give better code than what has been given here in the case that the multiplier $m \geq 2^W$. Applying it to signed division by seven gives the following code, where we have used the relation $-x = \bar{x} + 1$ to avoid a branch:

```
abs    an,n
li     M,x'92492493'     # Load magic number, (2**34+5)/7.
mulhu  q,M,an            # q = floor(M*an/2**32).
sri    q,q,2
srai   t,n,31            # These three instructions
xor    q,q,t             # negate q if n is
sub    q,q,t             # negative.
```

This is not quite as good as the code we gave for signed division by seven (six instructions), but it would be useful on a machine that has *abs* and *mulhu* but not *mulhs*.

Magenheimer *et al* (MPPZ) consider doing integer division by a constant by means of

$$\frac{n}{d} = \left\lfloor \frac{an+b}{z} \right\rfloor,$$

where z is a power of two, $a = \lfloor z/d \rfloor$, and $b = a + rem(z, d) - 1$. This is close to our method; we choose a different value for a and have $b = 0$. Thus our method would be expected to usually result in fewer instructions, particularly since the addition in $an + b$ must be done in double precision (it adds a $2W$-bit number to a number that is $W + 1$ or fewer bits in length). However, this method might yield better code on some machines if the multiplier of our method is greater than or equal to 2^W (so that we need the *add* and the *srxi*), and the multiplier of the (MPPZ) method is less than 2^W. This occurs for $d = 7$. For this case, both methods require six instructions on the IBM RISC System/6000.

(MPPZ) considers doing the multiply by a sequence of shifts and adds, and they point out that the periodicity sometimes found in the binary representation of the multiplier is helpful here. However, these instructions must operate on $2W$-bit operands, and hence this transformation would be useful only on machines that have a very slow or nonexistent multiply instruction for producing the high-order part of the product.

Grappel (Gra) gives a method for doing unsigned division by a constant that is the same as the (MPPZ) method with z fixed at 2^W. This method suffers from accuracy problems; it works for many small divisors but only a small percentage of large ones. For $W = 16$, among the first 20 integers it works for d = 2, 3, 4, 5, 6, 8, 10, 12, 14, 15, 16, 17, and 20.

References

(Alv) Alverson, Robert. Integer Division Using Reciprocals. In *Proceedings IEEE 10th Symposium on Computer Arithmetic*, June 26-28, 1991, Grenoble, France, pages 186-190.

(GGS) Gregoire, Dennis G., Groves, Randall D., and Schmookler, Martin S. *Single Cycle Merge/Logic Unit*, US Patent No. 4,903,228, February 20, 1990.

(Gra) Grappel, Robert D. Optimizing Integer Division by a Constant Divisor. Dr. Dobb's Journal, February 1991, pages 80-84.

(Hop) Hopkins, Martin E., informal communication.

(MPPZ) Magenheimer, Daniel J., Peters, Liz, Pettis, Kari, and Zuras, Dan. Integer Multiplication and Division on the HP Precision Architecture. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1987, pages 90-99.

(M&S) Markstein, Peter W., and Stephenson, Christopher J., *Multiplying Unsigned Integers in a Two's-Complement Computer*, IBM Research Report RC 5402 (May 6, 1975). They attribute this result (actually the inverse of it) to Burks, Goldstine, and von Neumann, 1946.

Appendix A Sample Magic Numbers

Table 1. Some magic numbers for W = 32

	signed		unsigned		
d	M (hex)	s	M (hex)	a	s
-5	99999999	1			
-3	55555555	1			
-2^k	7FFFFFFF	$k-1$			
1	-	-	0	1	0
2^k	80000001	$k-1$	2^{32-k}	0	0
3	55555556	0	AAAAAAAB	0	1
5	66666667	1	CCCCCCCD	0	2
6	2AAAAAAB	0	AAAAAAAB	0	2
7	92492493	2	24924925	1	3
9	38E38E39	1	38E38E39	0	1
10	66666667	2	CCCCCCCD	0	3
11	2E8BA2E9	1	BA2E8BA3	0	3
12	2AAAAAAB	1	AAAAAAAB	0	3
25	51EB851F	3	51EB851F	0	3
125	10624DD3	3	10624DD3	0	3

Table 2. Some magic numbers for W = 64

	signed		unsigned		
d	M (hex)	s	M (hex)	a	s
-5	9999999999999999	1			
-3	5555555555555555	1			
-2^k	7FFFFFFFFFFFFFFF	$k-1$			
1	-	-	0	1	0
2^k	8000000000000001	$k-1$	2^{64-k}	0	0
3	5555555555555556	0	AAAAAAAAAAAAAAAB	0	1
5	6666666666666667	1	CCCCCCCCCCCCCCCD	0	2
6	2AAAAAAAAAAAAAAB	0	AAAAAAAAAAAAAAAB	0	2
7	4924924924924925	1	2492492492492493	1	3
9	1C71C71C71C71C72	0	E38E38E38E38E38F	0	3
10	6666666666666667	2	CCCCCCCCCCCCCCCD	0	3
11	2E8BA2E8BA2E8BA3	1	2E8BA2E8BA2E8BA3	0	1
12	2AAAAAAAAAAAAAAB	1	AAAAAAAAAAAAAAAB	0	3
25	A3D70A3D70A3D70B	4	47AE147AE147AE15	1	5
125	20C49BA5E353F7CF	4	0624DD2F1A9FBE77	1	7