Integer Division Using Reciprocals

Robert Alverson
Tera Computer Company
400 North 34th St., Suite 300
Seattle, WA 98103

Abstract

As logic density increases, more and more functionality is moving into hardware. Several years ago, it was uncommon to find more than minimal support in a processor for integer multiplication and division. Now, several processors have multipliers included within the central processing unit on one integrated circuit [8, 12]. Integer division, due to its iterative nature, benefits much less when implemented directly in hardware and is difficult to pipeline. By using a reciprocal approximation, integer division can be synthesized from a multiply followed by a shift. Without carefully selecting the reciprocal, however, the quotient obtained often suffers from off-by-one errors, requiring a correction step. This paper describes the design decisions we made when architecting integer division for a new 64 bit machine. The result is a fast and economical scheme for computing both unsigned and signed integer quotients that guarantees an exact answer without any correction. The reciprocal computation is fast enough, with one table lookup and five multiplies, that this scheme is competitive with a dedicated divider while requiring much less hardware specific to division.

1 Introduction

Not too long ago, the cost of integer multiplication was sufficiently high that compiler writers often found it useful to reduce the strength of a multiplication by a constant into a series of additions. Lately, such "optimizations" must be done more carefully, since several current microprocessors include fast integer multipliers.

Integer division has remained an enigma. If a processor has any support for integer division, it is usually in the form of a simple iterative divider producing a single quotient bit per clock. While the dynamic frequency of integer divides is typically small, the time weighted impact on path length can be significant. One investigation found nearly 10 percent of all cycles for a spreadsheet application were devoted to integer division[1]. To mitigate this effect, various solutions have been used. The straightforward iterative divider is area-efficient but not pipelineable, complicating the instruction issue logic in a pipelined processor. A second option is a non-iterative array divider, which is pipelineable but not area-efficient. Both solutions result in divide operations which take about four times as many cycles as multiplies, for 32 bit integers. This is fundamental, since the partial products in a multiply may be summed in parallel while the remainders for divide must be calculated serially. With 64 bit integers, the divide latency worsens to around six times longer than that for multiply.

Modern compilers have been effective at reducing the cost of integer division, as some researchers have devised methods for dividing by small constants using scaled reciprocals[3, 9, 10]. This approach is quite effective, since constant divisors are quite frequent in practice. However, computing the reciprocal at run-time has been too costly for integer division in general until now. This paper presents an integer division scheme which supports fast division by a constant, is still efficient for non-constant divisors, requires little extra hardware, and is pipelinable. The ideas presented reflect the design decisions we made when architecting integer division for the Tera Computer[2], a new 64 bit machine.

Section 2 refines the integer division problem and informally describes its solution using reciprocals. Section 3 proves that the outlined division scheme produces the correct result for unsigned division in all cases. These results are extended to the division of signed integers in section 4.

2 Reciprocal Division

Integer division has several forms. Unsigned division gives the positive quotient and positive remainder from the division of two positive numbers. Extended precision division is the same as unsigned division except the dividend is a double word rather than a single word. For division of signed integers, two ways of defining the desired result are common, round toward zero (chopped division) and round toward negative infinity (floored division). Thus, there are four common variants of integer division: unsigned, extended precision, signed chopped,

*This research was supported by the United States Defense Advanced Research Projects Agency under Contract MDA972-89-C-0002. The views and conclusions contained in this document are those of Tera Computer Company and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U. S. Government.

and signed floored.

unsigned	$q = \lfloor x/y \rfloor$
extended-precision	$q = \lfloor (x_h 2^w + x_l)/y \rfloor$
signed chopped	$q = \lfloor abs(x/y) \rfloor * sign(x/y)$
signed floored	$q = \lfloor x/y \rfloor$

In all thses cases, the remainder is $x - q*y$ (for extended-precision division, $x = x_h 2^w + x_l$). The remainder takes the sign of the dividend in signed chopped division, while the sign of the remainder matches the sign of the divisor in signed floored division. FORTRAN requires signed chopped division, while C leaves the choice to the implementor. In addition, integer division is implicitly used in calculating C pointer differences and converting binary values to decimal for output.

The basic strategy for reciprocal division is to find values a and sh such that $x/y = x*a/2^{sh}$. For the moment, consider only the unsigned case $0 \leq x < 2^w, 0 < y < 2^w$, where w is the wordlength (our target machine has a 64 bit wordlength). Given a shift sh, the obvious choice for the scaled reciprocal is $a = round(2^{sh}/y)$. Thus, reciprocal division has the dubious advantage of computing a single division using a division, a multiply, and a shift. Fortunately, the reciprocal need not be calculated using division, but may be computed using a Newton-Raphson iterative approximation [13].

To guarantee quick convergence, the reciprocal iteration must start with an accurate initial guess. The standard solution, which works well for floating point reciprocals, is to use a table lookup[4]. The table index is taken from the k most significant mantissa bits after removing the leading one (for a normalized binary floating point number). The easiest way to look up an initial guess for the reciprocal of an integer divisor is to normalize the integer by converting it to float and applying the same lookup method.

While it is possible to convert the initial reciprocal estimate back to an integer immediately, we chose to perform the intermediate iteration steps with floating point. Since we use reciprocals for floating point divide as well, this choice allows the instruction that updates the reciprocal to be used by both integer and floating point divide.

The other solution is to use an integer iteration. While this option is conceptually simpler, the floating point iteration is adequate and reduces the overall architectural impact of divide support. Because the standard 53 bit floating point mantissa does not provide enough precision for 64 bit division, the result eventually must be converted to an integer.

When the reciprocal is converted to an integer, the shift sh must be selected. Essentially, the shift sh must be large enough for the final shift to truncate away the rounding error in the reciprocal. This guarantees that the quotient is either correct or low by one. Consequently, the remainder must be calculated and the quotient conditionally adjusted by one. Alternately, the shift sh can be made even larger to always yield a correct quotient. Of course, a larger sh leads to a larger scaled reciprocal, with which it is harder to compute. Thus the shift sh should be as small as possible while guaranteeing the desired level of accuracy.

Consider the division $(2^w - 3)/(2^w - 2)$. Here, a shift sh of $2w$ is necessary to compute the correct quotient, even though a shift of w yields an answer within one.

$$\begin{aligned} q &= \lfloor (2^w - 3) * \lceil 2^{sh}/(2^w - 2) \rceil / 2^{sh} \rfloor \\ &= 2^{w-sh} - 1 \text{ if } 2 \leq sh < w \\ &= 1 \text{ if } w \leq sh < 2w \\ &= 0 \text{ if } 2w \leq sh \end{aligned}$$

For this division to be handled without exception, the shift of $2w$ must yield the scaled reciprocal $2^w + 3$. To represent this quantity requires $w + 1$ bits, which is a challenge. For machines with 32 bit words, the reciprocal could be expressed using double precison IEEE floating point numbers[6]. With our wordsize of 64 bits, a double-extended format with 65 significand bits is needed to be useful. This width is more than the de facto standard for double-extended (64 bits in the 8087[7]). In addition, we have no other reason to support a double-extended format for our machine.

With the above constraints in mind, we present our integer division procedure. Each line represents one machine instruction. All float variable names start with f; the others are integers. First, we convert the integer divisor y to float and compute an initial reciprocal approximation which is correct to 17 bits. If our machine had floating point divide in hardware, we could use that to compute the initial reciprocal. Like the IBM RS/6000[11], we use reciprocals for floating point division. The shift sh is computed by counting zeros at the left of the divisor (powers of two are handled as special cases).

```
fy ← double(y);
fa ← approx(1/fy);
sh ← w + ceil(log2(y));
```

Next, we iterate to improve the accuracy of the reciprocal. This iteration mirrors the float reciprocal iteration, except we use the integer divisor to maximize the precision of the error term e. As a final step, the reciprocal is multiplied by 2^{sh} and rounded to ceiling.

```
fe ← nearest(1.0 - fa*y);
fa ← nearest(fa + fa*fe);
fe ← nearest(1.0 - fa*y);
a ← ceil((fa + fa*fe)*2^sh);
```

Unless the unrounded result of a reciprocal iteration has at least $2w$ correct bits, the reciprocal may still need a correction for proper rounding. For example, consider $y = 2^w - 1$. On conversion to float, this number rounds to 2^w. The float reciprocal will come to 2^{-w} and the scaled reciprocal is initially calculated as $2^w + 1$. Since the properly rounded answer is $2^w + 2$, a correction must be made.

This correction could be accomplished by another reciprocal iteration. Instead, we detect the cases where

the reciprocal is low by one with a trial division of y/y, which should give one. If the result is zero, then increment the reciprocal. While this correction is an annoyance, its cost may be amortized over the iterations of a loop or eliminated during compilation. The cost of a correction to the quotient, however, must be paid with each divide.

```
q ← floor(y*a/2^sh);
a ← a + (1-q);
```

Finally, do the division. The dividend x is multiplied by the scaled reciprocal a and shifted right by the amount sh. This multiply-shift logic can be shared with the logic needed to implement a floating point multiply add with a single round[5].

```
q ← floor(x*a/2^sh);
```

Notably, the compiler need only load a and sh for division by a constant, leaving only one operation to be computed at run-time. In the general case, only five more multiplies are needed to compute the scaled reciprocal. The only logic specifically needed for implementing integer division is the initial reciprocal lookup table. This same table can be used for computing floating point reciprocals, so its cost is also amortized over both integer and floating point divides.

3 Unsigned Division

While the preceding informal discussion identified some pitfalls in performing reciprocal division, it did not attempt to prove that we avoid them in our procedure. Here, we formally show that our division procedure determines the correct quotient when applied to unsigned integers. Let the wordlength be w bits. For a given divisor $y \neq 0$, let $sh = w + \lceil log_2 y \rceil$. Then $a = \lceil 2^{sh}/y \rceil$ implies $2^w \leq a < 2^{w+1}$. The ceiling of the reciprocal is the most useful rounding since the error introduced is truncated away when the final quotient is converted to an integer. The ceiling never rounds up to 2^{w+1}, since $2^{sh}/y$ is never more than $2^{w+1} - 1$:

$$\begin{aligned}
\lceil log_2 y \rceil &< log_2 y + 1 \\
2^{\lceil log_2 y \rceil} &< 2y \\
2^{\lceil log_2 y \rceil} &\leq 2y - 1 \\
2^{sh} &\leq 2^w(2y - 1) \\
\lceil 2^{sh}/y \rceil &\leq \lceil 2^w(2y-1)/y \rceil \\
a &\leq 2^{w+1} - \lfloor 2^w/y \rfloor \\
a &\leq 2^{w+1} - 1 \qquad \text{(as long as } y \leq 2^w\text{)}
\end{aligned}$$

Now the quotient can be computed by multiplying by a and dividing by 2^{sh}:

$$\begin{aligned}
q &= \lfloor x/y \rfloor \\
&= \lfloor (x/2^{sh}) * (2^{sh}/y) \rfloor \\
&\approx \lfloor (x/2^{sh}) * a \rfloor \\
&= \lfloor (x * a)/2^{sh} \rfloor \\
&= Q
\end{aligned}$$

For what range of values of the dividend x is the quotient q guaranteed to be correct? The boundary cases are $x = q * y$ and $x = q * y + y - 1$. In the former case, a small negative error in the scaled reciprocal a might make the quotient one too small. In the latter, a small positive error in a may cause the quotient to be one too large. Since a is the ceiling of the true value and both x and y are positive, the rounding of a can only cause the unrounded product $x * a$ to be too large. To show this error never propagates to the quotient, compare the computed quotient Q with the true quotient $q = \lfloor x/y \rfloor$:

$$\begin{aligned}
&\text{Let } r = a * y - 2^{sh} \\
&\quad x = q * y + s \\
&\quad 0 \leq s < y
\end{aligned}$$

$$\begin{aligned}
Q &= \lfloor x * a/2^{sh} \rfloor \\
&= \lfloor (q * y + s) * a/2^{sh} \rfloor \\
&= \lfloor (q * (a * y) + a * s)/2^{sh} \rfloor \\
&= \lfloor (q * (2^{sh} + r) + a * s)/2^{sh} \rfloor \\
&= q + \lfloor q * r + a * s)/2^{sh} \rfloor
\end{aligned}$$

For Q to equal q, $q * r + a * s$ must be less than 2^{sh}. While $q * r + a * s$ also must not be less than zero, it is clear that $Q \geq q$ from the rounding of the scaled reciprocal. As long as there is a bound on the size of the dividend, we can choose a scaled reciprocal which guarantees a correct quotient. The first step is to bound the term $q * r$, taking advantage of the fact that r is the remainder of 2^{sh} divided by y:

$$\begin{aligned}
r &\leq y - 1 \\
q * r &\leq q * (y - 1) \\
q * (y - 1) &= x - s - q \\
q * r &\leq x - q - s
\end{aligned}$$

When the dividend x is a maximum 2^w, q must be at least one since y cannot be greater than a maximum x. Thus, the difference $x - q$ cannot be as large as 2^w.

$$\begin{aligned}
x - q &< 2^w \\
q * r &< 2^w - s \\
q * r + a * s &< 2^w + (a - 1) * s
\end{aligned}$$

Similarly, the term $a * s$ can be bounded by recalling that s is the remainder of x/y and thus is between zero and $y - 1$.

$$\begin{aligned}
s &\leq y - 1 \\
q * r + a * s &< 2^w + (a - 1)(y - 1) \\
q * r + a * s &< 2^w + a * y - a - y + 1 \\
r &\leq y - 1 \\
a * y &\leq 2^{sh} + y - 1 \\
q * r + a * s &< 2^{sh} + 2^w - a
\end{aligned}$$

Finally, the lower bound on the scaled reciprocal a leaves the desired relation which guarantees the computed quotient is correct.

$$\begin{aligned}
a &\geq 2^w \\
q * r + a * s &< 2^{sh} \\
Q &= q
\end{aligned}$$

Hence, Q is correct if $x \leq 2^w$. While the dividend x is never 2^w for unsigned division with wordlength w, the stronger result will prove useful for signed division where two's complement allows the value -2^{w-1}.

We have shown that a scaled reciprocal can be used to compute a quotient of positive integers exactly. To achieve this accuracy, we need a reciprocal with one more bit than will fit in a word. Like a floating point mantissa, the scaled reciprocal has a leading one which need not be explicitly represented. With this improvement, the scaled reciprocal nicely fits in one machine word. To maintain the hidden bit, the shift sh is constrained to yield a scaled reciprocal $2^w \leq a < 2^{w+1}$, even though other values of a and sh might work just as well.

4 Signed Division

In the realm of signed integers, the two most common definitions of division round the infinitely precise quotient towards zero or towards negative infinity. Rounding towards zero (chopped rounding) results in a division algorithm in which the operands are separated into sign and magnitude parts. Except for the extra cases where the dividend or divisor is -2^{w-1}, chopped division is exactly the same as unsigned division with a wordlength of $w-1$ and one sign bit. However, the previous proof already allowed for a dividend of 2^{w-1}, so there is no mathematical problem. The hardware still must allow w bits for the magnitude of the dividend in this case. Thus, the signed integer chopped division of x by y is simply performed:

$$\begin{aligned} a &= \lceil abs(2^{sh}/y) \rceil * sign(y) \\ q &= \lfloor abs(x*a)/2^{sh} \rfloor * sign(x*a) \end{aligned}$$

Note that the scaled reciprocal computation effectively rounds away from zero, a round mode that is not part of the IEEE floating point standard.

For rounding towards negative infinity (floored rounding), the handling depends on the sign of the quotient. For a non-negative quotient, floored division may be computed exactly as chopped division, since the results will always match. Complication arises when the signs of the dividend and divisor do not match. To compute the quotient as

$$q = \lfloor x * a/2^{sh} \rfloor$$

the scaled product $x*a/2^{sh}$ must not be less than x/y. When the quotient is negative, this constraint requires a to be rounded towards zero. As it stands, we must round a away from zero when the signs of x and y match and toward zero when their signs are different. Rounding the reciprocal depending on the sign of the dividend x is inconvenient and reduces the performance improvement possible when the divisor is constant. Fortunately, there is another solution. We may use the same reciprocal as with chopped division by biasing the product so that any error is one-sided, allowing the exact quotient to be calculated without any correction. While the same reciprocal is computed as always, the quotient calculation requires the bias b:

$$\begin{aligned} a &= \lceil abs(2^{sh}/y) \rceil * sign(y) \\ b &= \text{if } x*a \geq 0 \text{ then } 0 \text{ else } 2^{w-1}-1 \\ q &= \lfloor (x*a+b)/2^{sh} \rfloor \end{aligned}$$

The bias must be large enough to cancel out negative error when the product of the dividend and the scaled reciprocal is negative. This error is most significant when the dividend is an exact multiple of the divisor: $x = qy$. In this case, the slightest negative error will cause the quotient to be low by one. To get the correct quotient from the floor, its argument must not be less than the true quotient q:

$$\begin{aligned} (x*a+b)/2^{sh} &\geq q \\ x*a+b &\geq q*2^{sh} \\ b &\geq q*2^{sh} - x*a \\ b &\geq q*2^{sh} - q*y*a \\ b &\geq q*(2^{sh} - a*y) \\ b &\geq -q*r \qquad \text{(By definition of } r\text{)} \end{aligned}$$

Note that r is zero when y is a power of two. Consequently, the bias is constrained by divisors which are not powers of two. Since $x = -2^{w-1}$ can only be exactly divided by powers of two, the minimum dividend to consider is $-2^{w-1}+1$.

$$\begin{aligned} r &\leq y-1 \\ -q*r &\leq -q*y+q \\ -q*r &\leq -x+q \\ -x &\leq 2^{w-1}-1 \\ q &\leq -1 \\ -q*r &\leq 2^{w-1}-2 \\ b &\geq 2^{w-1}-2 \qquad \text{(For a correct quotient)} \end{aligned}$$

As an example, consider dividing $(-2^{w-1}+1)/(2^{w-1}-1)$. We select the shift $sh = 2w-2$, yielding the scaled reciprocal $a = 2^{w-1}+2$. This requires a bias $b \geq 2^{w-1}-2$ to get the correct quotient of negative one.

The bias must not be so large that a correct answer is incremented to become too large (i.e. smaller in magnitude). When the quotient is negative, the biased product $x*a+b$ is independent of the signs of x and y. Thus, only the case of negative x and positive y must be considered. The answer is most sensitive to the bias when $x = q*y+y-1$. As before, the critical constraint is to compute a value which rounds to the correct quotient:

$$\begin{aligned} (x*a+b)/2^{sh} &< q+1 \\ x*a+b &< (q+1)*2^{sh} \\ b &< (q+1)*2^{sh} - x*a \\ b &< (q+1)*2^{sh} - ((q+1)*y-1)*a \\ b &< (q+1)*(2^{sh} - a*y) + a \\ b &< -r*(q+1)+a \end{aligned}$$

If b is less than the minimum value of $-r*(q+1)+a$, then correctness is assured. To find the minimum value, consider each component. Recall that we assume x is

negative and y is positive, yielding a negative quotient after floored rounding:

$$\begin{aligned} q+1 &\leq 0 \\ r &\geq 0 \\ -r*(q+1) &\geq 0 \\ a &\geq 2^{w-1} \\ b &< 2^{w-1} \end{aligned}$$

We see that the bound on the scaled reciprocal a leads to a sufficient condition on b for the correctness of the quotient. In fact, this upper limit for b is also necessary, since the division $-1/1$ will not give the correct answer if $b \geq 2^{w-1}$. Thus, the bias of $b = 2^{w-1} - 1$ is correct for all cases with a negative quotient.

Unsigned integer division readily generalizes to signed integer chopped division. While floored division does entail extra complication, our carefully chosen bias allows the extra complexity to be hidden from the compiler, so that constant divisors may be converted to scaled reciprocals without regard to the rounding desired for the integer quotient.

5 Conclusion

While reciprocal division has been used for floating point for many years, until now the ideas have only been applied to integer arithmetic when the divisor is constant. The division scheme described here is fast in the general case, requiring only six multiplies and a table lookup. Our reciprocal division supports unsigned integer, signed chopped, and signed floored division. The real strength of our method is division by a constant, which takes only a single multiply and shift, one operation on our machine. The analysis we presented shows that the computed quotient is always exact — no adjustment or correction is necessary. There is very little extra hardware required, primarily rounding logic and an initial reciprocal lookup table. Thus, we believe this design strikes a balance between hardware and software support for 64 bit integer division.

Further research is needed to determine how best to implement extended precision division using reciprocals. Besides generating the correct quotient in the normal case, extended precision division may result in overflow if the quotient is larger than will fit in one word. A simple change to the unsigned division code sequence seems to allow computing the extended-precision division $(x_h 2^w + x_l)/y$ correct to $w-2$ bits, although overflow is not detected. We are currently investigating the best way to increase the accuracy to guarantee the full w bits.

Acknowledgements

The author is indebted to Burton Smith, who wanted to do integer division just a little bit better. Thanks also to Gail Alverson and Jon Bertoni for their comments on an early version of this paper.

References

[1] Thomas L. Adams and Richard E. Zimmerman. An Analysis of 8086 Instruction Set Usage in MS DOS Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 152–160, April 1989.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[3] 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.

[4] D. L. Fowler and J. E. Smith. An Accurate, High Speed Implementation of Division by Reciprocal Approximation. In *Proceedings of the Ninth Symposium on Computer Arithmetic*, pages 60–67, September 1989.

[5] E. Hokenek, R. K. Montoye, and P. W. Cook. Second-Generation RISC Floating Point with Multiply-Add Fused. *IEEE Journal of Solid-State Circuits*, 25(5):1207–1213, October 1990.

[6] *IEEE Standard for Binary Floating Point Arithmetic.* IEEE, New York, NY, 1985.

[7] *Microprocessor and Peripheral Handbook, Volume I - Microporcessor.* Intel Corporation, Santa Clara, CA, 1989.

[8] Gerry Kane. *MIPS R2000 RISC Architecture.* Pentice Hall, Englewood Cliffs, NJ, 1987.

[9] S-Y. R. Li. Fast Constant Division Routines. *IEEE Transactions on Computers*, C-34(9):866–869, September 1985.

[10] Daniel J. Magenheimer, Liz Peters, Karl Pettis, and Dan Zuras. 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*, pages 90–99, April 1987.

[11] P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, January 1990.

[12] Brett Olsson, Robert Montoye, Peter Markstein, and MyHong NguyenPhu. RISC System/6000 Floating-Point Unit. In Mamata Misra, editor, *IBM RISC System/6000 Technology*, pages 34–42. IBM Corporation, Austin, TX, 1990.

[13] Shlomo Waser and Michael J. Flynn. *Introduction to Arithmetic for Digital Systems Designers.* Holt, Rinehart and Winston, New York, NY, 1982.