Software Integer Division

Thomas L. Rodeheffer
Microsoft Research, Silicon Valley

August 26, 2008

Abstract

Early computers omitted instructions for integer multiplication and division, requiring these operations to be synthesized in software. Even some modern RISC and DSP architectures are deficient in the case of division. Therefore software methods for performing integer division continue to be of interest.

We consider typical architectures based on two's complement binary arithmetic and present various methods of performing single precision unsigned integer division in software. In addition to methods based on the standard test-subtract-shift approach, we present a method and variants based on a novel adaptation of the Newton-Raphson recurrence to the domain of unsigned integers.

1 Introduction

> There are many algorithms for *hardware* implementation of arithmetic operations which are very interesting but which appear to be inapplicable to computer programs...
>
> —Donald Knuth [8, page 244]

Early computers omitted instructions for integer multiplication and division, requiring these operations to be synthesized in software. Even some modern RISC and DSP architectures, for example, the ARM [15] and the TMS320C64 [16], are deficient in the case of division. Therefore software methods for performing integer division continue to be of interest.

Although there is much literature on hardware implementations of computer arithmetic and division in particular [2, 4, 9, 11, 12, 13, 14], methods that are useful in hardware are often completely impractical in software. Descriptions from the point of view of software or microcode typically present just the shift double word dividend method [13, pages 213–215] or occasionally the shift divisor method [18, page 38].

Methods for performing division can be classified roughly into methods based on digit recurrence (which retire a fixed number of quotient bits per iteration) and methods based on functional iteration (which double the number of retired quotient bits per iteration). Both types have practical software implementations, depending on the operations provided by the underlying hardware architecture.

We consider typical architectures based on two's complement binary arithmetic. Although one more bit of precision is required, unsigned integer division is essentially a simpler problem than signed division because it can ignore the case analysis required to deal with the signs of the dividend and the divisor. We consider methods for performing single precision unsigned integer division.

Let W be the number of bits in a machine word. Typically $W = 32$. Since the subscript W would otherwise appear on all definitions, we omit it for clarity. Let $U = \{0, \ldots, 2^W - 1\}$ be the set of unsigned integers and $U^+ = \{1, \ldots, 2^W - 1\}$ be the set of strictly positive unsigned integers. Given dividend $x \in U$ and divisor $y \in U^+$, the problem of unsigned integer division is to compute the quotient $q = \lfloor x/y \rfloor$ and remainder $r = x - q * y$. Note that $q, r \in U$ and $r < y$.

In the exposition to follow, some care is necessary to distinguish between abstract mathematical operations and the concrete operations implemented by a two's complement architecture. Equations written in the text refer to abstract operations, although named concrete operations

are used freely. When the text needs to refer to the concrete operations of addition and subtraction it uses ADD and SUB. Code examples are written in C and use + and - since ADD and SUB would be clumsy. All of the code examples have been extracted mechanically from a compiled and extensively tested program.

Section 2 describes operations that might be provided by a hardware architecture. Section 3 presents division methods based on digit recurrence. Section 4 presents division methods based on functional iteration. Section 5 concludes. Proofs appear in Apendix A.

2 Operation definitions

We assume the usual two's complement arithmetic operations in which overflow is ignored. Given $a, b \in U$ let

$$a \texttt{ ADD } b = (a + b) \bmod 2^W$$

$$a \texttt{ SUB } b = (a - b) \bmod 2^W$$

Comparison of unsigned integers is assumed to produce the correct answer in all cases. Note that, because of overflow, this is not the same thing as testing the high order bit of a SUB b.

We use the following operations to extract bits and to compute half. Note that computing half is simply a matter of shifting right by one bit position. Given $a \in U$ let

$$bit_k(a) = \left\lfloor \frac{a}{2^k} \right\rfloor \bmod 2$$

$$hibit(a) = \left\lfloor \frac{a}{2^{W-1}} \right\rfloor$$

$$half(a) = \lfloor a/2 \rfloor$$

Instructions to perform bit-wise logical operations are common. It turns out that some of these are occasionally useful even in numeric calculations. Given $a, b \in U$ let

$$band(a, b) = \sum_{k=0}^{W-1} bit_k(a) * bit_k(b) * 2^k$$

Instructions to perform arbitrary logical shifts are common. Note that we adopt the typical restriction that the shift amount must be less than the word size. Given $u, k \in U$ with $k < W$ let

$$lsl(u, k) = (u * 2^k) \bmod 2^W$$

$$lsr(u, k) = \left\lfloor \frac{u}{2^k} \right\rfloor$$

Some architectures provide an instruction to count the number of leading zeros in a non-zero value. A reasonably efficient software method can often be constructed based on binary search and lsr. Given $y \in U^+$ let

$$clz(y) = W - \lfloor \log_2(y) \rfloor - 1$$

Note that $k = \lfloor \log_2(y) \rfloor$ is the integer that satisfies

$$2^k \leq y < 2^{k+1}$$

$clz(y)$ is the number of leading zero bits in the representation of y. Observe that $0 \leq clz(y) < W$.

Architectures that lack an integer divide instruction sometimes provide an instruction to compute one quotient bit in a tight loop. One clever example is a peculiar test-subtract-shift operation. Given $a, b \in U$ let

$$tsubsh(a, b) = \begin{cases} lsl(a - b, 1) + 1 & \text{if } a \geq b \\ lsl(a, 1) & \text{otherwise} \end{cases}$$

Section 3.4 shows how $tsubsh$ can be useful.

Some architectures provide unsigned integer multiplication operations. Given $a, b \in U$ let

$$mul(a, b) = (a * b) \bmod 2^W$$

$$umulh(a, b) = \left\lfloor \frac{a * b}{2^W} \right\rfloor$$

$mul(a, b)$ is the low-order word of the product of a and b and $umulh(a, b)$ is the high-order word of the product of a and b.

Although it is difficult to reverse the order of bits in a word in software, this is an easy task for hardware. So some architectures provide an instruction for it. Given $a, b \in U$ let

$$bitrev(a) = \sum_{k=0}^{W-1} bit_k(a) * 2^{W-k-1}$$

Section 4.2 shows how $bitrev$ can be useful.

```
r = 0; q = x;
unsigned i = W;
do {
   // double word left shift (r,q)
   r = r + r + hibit(q);
   q = q + q;

   if (r >= y) { r = r−y; q = q+1; }
   i = i−1;
} while (i != 0);
```

Figure 1: Shift double word dividend method.

3 Digit recurrence methods

Digit recurrence methods retire a fixed number of quotient bits per iteration. Although versions that retire more than one quotient bit per iteration are popular in hardware, such as radix-4 SRT division [7, 10], the quotient digit selection and factor selection functions are inefficient to implement in software. Hence the only practical software digit recurrence methods are those that retire one quotient bit per iteration. Although these methods all employ the same basic test-subtract-shift approach, there are many variations.

3.1 Shift double word dividend

The shift double word dividend method pads the dividend out to a double word (r, q) and then performs W test-subtract-shift iterations. Each iteration determines one quotient bit by test-subtracting the divisor from r and then shifting the double word (r, q) one bit left. The quotient bits can cleverly be shifted into q as the dividend shifts left. At the end, q contains the quotient and r the remainder.

Figure 1 shows a code example. The shift double word dividend method requires W iterations independent of the values of x and y.

Note that the concrete operation q `ADD` q shifts q one bit position to the left, discarding the high-order bit. This is an instance in which the difference between the concrete operation q `ADD` q and the abstract mathematical operation $q + q$ can be exploited.

On architectures with a carry flag the double-word left-shift of (r, q) can typically be implemented in two instructions. The first instruction shifts q left while saving $hibit(q)$ in the carry and the second shifts r left while bringing in the carry.

The shift double word dividend method is frequently employed in hardware as well as software. It turns out that the logic required to compute r `SUB` y can evaluate $r \geq y$ with essentially no extra effort and so both of these can be computed simultaneously in the same logic unit. There are variations depending on whether the remainder is updated conditionally based on $r \geq y$ (as shown in the code example here), is updated unconditionally and then restored if $r \geq y$ turns out to have been false (the so-called "restoring version"), or updated unconditionally and then the future logic of the algorithm inverted if $r \geq y$ turns out to have been false (the so-called "non-restoring version"). Depending on the details of the architecture, software implementations of these variants may or may not be practical.

3.2 Shift divisor

The shift divisor method [18, page 38] is based on shifting the divisor rather than shifting the dividend. The idea is to shift the divisor left (by iterated doubling) until it exceeds the dividend and then compute each bit of the quotient using a step of divisor right-shift (halving) and test-subtraction.

However, there is a problem. Because of the difference between abstract and concrete operations, a naive implementation fails for large dividends. When $x \geq 2^{W-1}$ there will be some divisors ($y = 2$, for example) for which the required left-shifted value would equal or exceed 2^W and thus cannot be represented as an unsigned integer. Fortunately this can be fixed without much trouble.

The fix is to stop doubling the divisor y when $x_0 < y + y$, where x_0 is the original value of x, and then adjust the quotient loop so that the test-subtraction occurs before halving the divisor. But since we cannot depend on $y + y = y$ `ADD` y, the doubling phase proceeds carefully as follows.

We start out by evaluating $x \geq y$. If this is false, we are done with doubling. Otherwise, we know that $x - y = x$ `SUB` y, so we decrease x by y, which results in a state with the invariant $x = x_0 - y$. At this point, if $x \geq y$ we know that $x_0 \geq y + y$ and therefore $y + y = y$ `ADD` y. Given

```
r = x; q = 0;
unsigned y0 = y; // original divisor
// divisor doubling phase
if (x >= y) {
   x = x−y;
   while (x >= y) { x = x−y; y = y+y; }
}
// quotient computing phase
for (;;) {
   if (r >= y) { r = r−y; q = q+1; }
   if (y == y0) break;
   q = q + q;
   y = half(y);
}
```

Figure 2: Shift divisor method.

```
r = x; q = 0;
if (y <= r) {
   unsigned i = clz(y) − clz(r);
   y = lsl(y,i);
   // quotient computing phase
   for (;;) {
      if (r >= y) { r = r−y; q = q+1; }
      if (i == 0) break;
      i = i − 1;
      q = q + q;
      y = half(y);
   }
}
```

Figure 3: Improved shift divisor method.

this, decreasing x by y and then doubling y preserves the invariant. We iterate until the condition $x \geq y$ is false. When any of the evaluations of $x \geq y$ comes up false, it must be the case that $x_0 < y + y$, and so the doubling phase is done. Figure 2 shows a code example.

Observe that once $x \geq y$ comes up false in the divisor doubling phase, the value of x is never used thereafter. In many architectures, decreasing x by y evaluates $x \geq y$ as a side effect and so these two operations can be combined into a single instruction, resulting in a tight loop.

Compared with the shift double word dividend method, the code for the shift divisor method is longer and runs slower in the worst case, since there are iterations doubling the divisor to scale it up and then iterations halving the divisor to compute the quotient bits. However, the number of iterations depends on the logarithm of the quotient, which is typically much less than W, so in the typical case the shift divisor method runs faster than the shift double word dividend method.

3.3 Improved shift divisor

It can be observed that what the doubling phase of the shift divisor method achieves is to align the leading 1 bit of the divisor with the leading 1 bit of the dividend. The count-leading-zeros and logical-shift-left operations can be exploited to attain this alignment directly. Figure 3 shows a code example.

Care must be taken that the arguments to the clz and lsl operations fall within their defined ranges and so produce intelligible results. Assuming $0 < y$, the check for $y \leq r$ guarantees $0 < r$ and so we can conclude

$$clz(r) \leq clz(y)$$

$$0 \leq clz(y) - clz(r) < W$$

$$clz(y) - clz(r) = clz(y) \; \texttt{SUB} \; clz(r)$$

Consequently the operation $lsl(y, i)$ produces exactly the aligned divisor we need.

Architectures that provide a count-leading-zeros operation typically define a result for $clz(0)$ that can be checked easily. In such a case, it might be advantageous to recast the argument checking.

Replacing the divisor doubling loop with count-leading-zeros and logical-shift-left operations reduces the instruction count and makes this method roughly identical to the shift double word dividend method in both code size and worst case execution time. The improved shift divisor method also has the advantage of considerably better typical execution time.

3.4 Align divisor shift dividend

The align divisor shift dividend method is based on aligning the divisor under the dividend and then performing a number of test-subtract iterations which shift the dividend left. As in the shift divisor method, the number of iterations is $k + 1$, where k is the number of bit positions the

divisor must be shifted left in order to align it under the dividend. The quotient bits can cleverly be saved in the rightmost positions of the dividend as it shifts left on each test-subtract iteration.

The $tsubsh$ operation defined in Section 2 performs the entire iteration. This operation is attractive for hardware implementation because only two registers are required, one containing the aligned divisor (read only) and one containing the dividend (read and written). The SUBC instruction on the TMS320C64 performs exactly this operation and details of how to exploit it appear in an application note [3].

Unfortunately, the align divisor shift dividend method has a problem with large dividends. It is possible for the dividend to exceed $2^{W-1} - 1$ but still be less than the aligned divisor. The naive implementation performs a left shift, but this drops the high order bit of the dividend.

The problem can be fixed, but the fix is not very pleasant. Basically, the first test-subtract iteration has to be handled as a special case, saving its quotient bit in a separate location and not shifting the (possibly subtracted) dividend. Then, if $k \neq 0$, additional iterations are required. In this case, we know that the divisor must have been shifted left by at least one bit during alignment, so we can safely shift the divisor right by one bit. This produces the same alignment as if the dividend had been shifted one bit left during the special first iteration. Then we proceed with k regular test-subtract-shift-dividend iterations. At the end, the first quotient bit has to be combined with the quotient bits resulting from the k regular iterations.

Figure 4 shows a code example. The complexities of handling the special first iteration and then extracting the quotient and remainder at the end are mitigated by the tight inner loop based on the $tsubsh$ operation.

```
r = x; q = 0;
if (y <= r) {
  unsigned k = clz(y) − clz(r);
  y = lsl(y,k); // align y
  if (r >= y) { r = r−y; q = lsl(1,k); } // special first iter
  if (k != 0) {
    y = half(y);
    unsigned i = k;
    do { r = tsubsh(r,y); i = i−1; } while (i!=0); // k iters
    q = q + r; // combine with first cycle quotient bit
    r = lsr(r,k); // extract remainder
    q = q − lsl(r,k); // leave just quotient
  }
}
```

Figure 4: Align divisor shift dividend method.

4 Functional iteration methods

Functional iteration methods compute a recurrence that converges to a useful value. They have the advantage that fewer iterations are required, especially for larger word sizes, because the recurrence converges quadratically, doubling the number of retired quotient bits each iteration. They have the disadvantage that each iteration requires multiplication. Fortunately, architectures that lack division often provide a high-speed multiplication operation.

There are two basic functional iteration methods. Divisor reciprocation uses the Newton-Raphson recurrence to compute the reciprocal of the divisor which is then multiplied by the dividend. Goldschmidt's Algorithm [5] multiplies both dividend and divisor by a series of factors that cause the divisor to converge to 1.

The principal difficulty in using functional iteration methods is arranging for enough precision in intermediate calculations so that the final result has sufficient accuracy. Hardware floating point implementations often use a large number of extra bits of precision, in some cases even doubling the width of the data path [12]. Such an approach is painful to emulate in software.

An adaptation of divisor reciprocation for software unsigned integer division is presented below. This method uses only single precision unsigned integer operations as defined in Section 2. Previous work has used floating point operations [1] or considered only the special case of division by a constant [6].

4.1 Divisor reciprocation

One way of dividing real numbers is to multiply the dividend by the reciprocal of the divisor. Given a real divisor $Y > 0$ and an initial approximation Z_0 that satisfies

$$0 < Z_0 < 2/Y$$

the Newton-Raphson recurrence

$$Z_{i+1} = Z_i * (2 - Z_i * Y)$$

converges to the reciprocal of Y. The convergence is amazingly fast, with the error falling quadratically on each iteration.

An analogous technique can be used to implement unsigned integer division. The value 2^W serves as an analogue of the unit value. An unsigned integer z such that the product $y * z$ approximates 2^W serves as an analogue of the reciprocal of y. And an analogue of the Newton-Raphson recurrence can be used to compute an approximation of the reciprocal.

Care is required to guarantee that the correct result is obtained in all cases. We arrange for all intermediate results to fall within the permitted range of unsigned integers, so that the concrete operations can be used directly. The main insight is to use lower bound approximations. This makes all the details work out in a very direct manner.

Newton-Raphson iteration is often used to implement fixed-point division in hardware. However we are not aware of any treatment of this idea for unsigned integer division. The following sections develop and evaluate the method in some detail.

4.1.1 Definitions

Given $y \in U^+$ observe that there exist one or more $z \in U^+$ such that $y * z < 2^W$. We define $inv(y)$ as the largest such z. The value $inv(y)$ serves as an analogue of the reciprocal of y. It follows that

$$2^W - y \leq y * inv(y) < 2^W$$

For any dividend $x \in U$ and quotient $q = \lfloor x/y \rfloor$ it follows that

$$q - 1 \leq \left\lfloor \frac{x * inv(y)}{2^W} \right\rfloor \leq q$$

Observe that $\lfloor a * b/2^W \rfloor = umulh(a, b)$ when $a, b \in U$. Multiplying the dividend x by $inv(y)$ does not quite get us the quotient, but only a close lower bound of the quotient. We can get to the actual quotient by computing the remainder and performing a few test-subtract iterations.

If we multiply the dividend by $inv(y)$, at most one test-subtract iteration will be required. However, it is almost as good to multiply by a close lower bound of $inv(y)$. We will still get a close lower bound of the quotient, just perhaps not as close as if we had used the actual value of $inv(y)$. Several test-subtract iterations may be required to get to the actual quotient.

So the problem becomes how to obtain a close lower bound of $inv(y)$. This can be accomplished by an analogue of the Newton-Raphson recurrence. For the following exposition, it is convenient to define several sets of lower bounds of $inv(y)$. Let

$$LB_y = \{z \in U^+ : 1 \leq y * z < 2^W\}$$

$$DLB_y = \{z \in U^+ : 2^{W-1} \leq y * z < 2^W\}$$

$$TYLB_y = \{z \in U^+ : 2^W - 2 * y \leq y * z < 2^W\}$$

We say that LB_y is the set of all lower bounds, DLB_y of *decent* lower bounds, and $TYLB_y$ of *two-y* lower bounds.

4.1.2 The UNR recurrence

UNR stands for Unsigned integer Newton-Raphson.

Theorem 1 *Given $y \in U^+$, $z_i \in LB_y$, and the UNR recurrence*

$$z_{i+1} = z_i + \left\lfloor \frac{z_i * (2^W - y * z_i)}{2^W} \right\rfloor$$

it follows that $z_i \leq z_{i+1}$ and $z_{i+1} \in LB_y$.

It may be observed that the UNR recurrence bears a lot of similarity to the Newton-Raphson recurrence. However, the variables have been pushed around so that the calculation maps directly onto the concrete operations. It turns out that

$$z_{i+1} = z_i \texttt{ ADD } umulh(z_i, mul(0 \texttt{ SUB } y, z_i))$$

This can be verified by checking the ranges of the argument and result values of each of the concrete operations. In particular, since $y, z_i, y * z_i \in U^+$ we have

$$\begin{aligned} mul(0 \texttt{ SUB } y, z_i) &= ((0 \texttt{ SUB } y) * z_i) \bmod 2^W \\ &= (-y * z_i) \bmod 2^W \\ &= 2^W - y * z_i \end{aligned}$$

4.1.3 Final value of the UNR recurrence

Starting from an initial $z_0 \in LB_y$ the UNR recurrence computes a non-decreasing sequence $z_0, z_1, z_2, \ldots$ of integers where $z_i \leq inv(y)$ for all i. Consequently the recurrence must eventually reach a final value. There must be some smallest $n \geq 0$ such that $z_n = z_m$ for all $m \geq n$. We say that the recurrence reaches its final value in n iterations.

Of course, we would like the final value to be close to $inv(v)$. This is accomplished by starting the recurrence with a decent lower bound as discussed next.

4.1.4 Starting with a decent lower bound

Theorem 2 *Given $z_0 \in DLB_y$ and computing z_1, z_2, and so on using the UNR recurrence, it follows that the final value $z_n \in TYLB_y$.*

Observe from the definition of $TYLB_y$ that the final value z_n is within 1 of $inv(y)$, since at most one more multiple of y will fit before 2^W.

It may be observed that the condition $z_0 \in DLB_y$ is an exact analogue to the condition on the initial approximation for Newton-Raphson iteration.

Of course, in order to start the recurrence with a decent lower bound, we need to be able to compute one. Fortunately, that is not hard. Given $y \in U^+$ take $k = \lfloor \log_2(y) \rfloor$. Observe that $2^k \leq y < 2^{k+1}$ and $0 \leq W - k - 1 < W$. Take $z_0 = 2^{W-k-1}$. Clearly $z_0 \in U^+$. Substitution shows that

$$2^{W-1} \leq y * z_0 < 2^W$$

Therefore $z_0 \in DLB_y$. Observe that $W - k - 1$ is computed by the count-leading-zeros operation. Hence

$$z_0 = 2^{W-k-1} = 2^{clz(y)} = lsl(1, clz(y))$$

4.1.5 Using a two-y lower bound

Theorem 3 *Given $z \in TYLB_y$, $x \in U$, and $q = \lfloor x/y \rfloor$ it follows that*

$$q - 2 \leq \left\lfloor \frac{x * z}{2^W} \right\rfloor \leq q$$

```
// decent start
unsigned z = lsl(1,clz(y));
// z recurrence
unsigned my = 0−y;
for (;;) {
  unsigned zd = umulh(z,mul(my,z));
  if (zd == 0) break;
  z = z + zd;
}
// q estimate
q = umulh(x,z); r = x − mul(y,q);
// q refinement
if (r >= y) { r = r − y; q = q + 1;
  if (r >= y) { r = r − y; q = q + 1; }
}
```

Figure 5: UNR recurrence method.

We can use $z \in TYLB_y$ to compute a quotient lower bound $\hat{q}$ and corresponding remainder $\hat{r}$.

$$\hat{q} = \left\lfloor \frac{x * z}{2^W} \right\rfloor = umulh(x, z)$$

$$\hat{r} = x - y * \hat{q} = x \texttt{ SUB } mul(y, \hat{q})$$

By checking the ranges, it is easy to verify that the concrete operations compute the desired results. From Theorem 3 it follows that at most two test subtractions of y from $\hat{r}$ are needed to arrive at the final quotient and remainder.

4.1.6 Implementation

Putting it all together gives the UNR recurrence method for computing unsigned integer division shown in Figure 5. Each iteration of the UNR recurrence requires one mul and one $umulh$. However, convergence is very fast, with the number of accurate significant bits doubling on each iteration. For $W = 32$, the recurrence never takes more than six iterations to reach the final value and then one additional iteration to establish that the recurrence makes no further improvement. In almost all cases, fewer than six iterations are required. The next section provides detailed performance measurements.

Although the code of the UNR recurrence method is longer and more complicated than for the improved shift divisor method, due to the small iteration count it typically

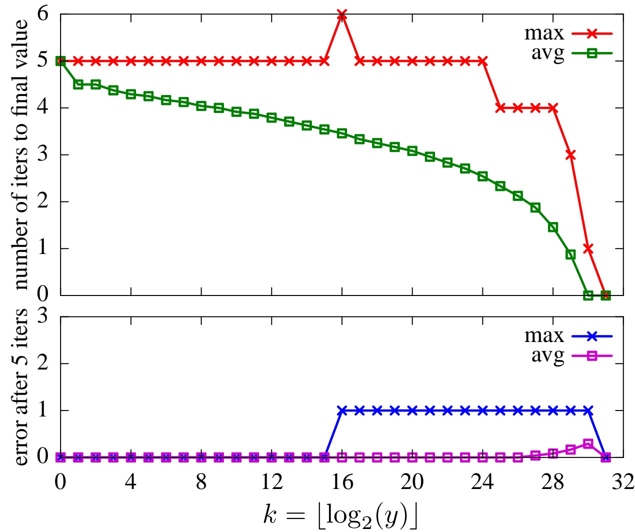

Figure 6: Performance for W=32 of the UNR recurrence.

```
// decent start
unsigned z = lsl(1,clz(y));
// z recurrence, 5 iterations
unsigned my = 0−y;
z = z + umulh(z,mul(my,z));
z = z + umulh(z,mul(my,z));
z = z + umulh(z,mul(my,z));
z = z + umulh(z,mul(my,z));
z = z + umulh(z,mul(my,z));
// q estimate
q = umulh(x,z); r = x − mul(y,q);
// q refinement
if (r >= y) { r = r − y; q = q + 1;
   if (r >= y) { r = r − y; q = q + 1; }
}
```

Figure 7: UNR recurrence method for W=32.

runs faster, assuming that the mul and $umulh$ operations are reasonably fast.

4.1.7 Performance for W=32

To investigate the performance of the UNR recurrence for the special case of $W = 32$, the method was instrumented to record two statistics: the number of iterations required to reach the final value of the recurrence (not counting the extra iteration that merely establishes no further improvement) and the error between the value of the recurrence after five iterations and the actual value of $inv(y)$.

The instrumented method was run once for every divisor $y \in \{1, \ldots, 2^{32} - 1\}$. The divisors were grouped into buckets based on $k = \lfloor \log_2(y) \rfloor$ and the maximum and average of each statistic calculated for each bucket. The results are plotted in Figure 6.

The average number of iterations decreases from 5.0 to 0.0 as k increases from 0 to 31. This can be explained by considering that $\log_2(y) + \log_2(inv(y)) \approx W$, so fewer and fewer bits are left for $inv(y)$ as k increases. With fewer bits to determine, the recurrence reaches the final value sooner.

Depending on the relative costs of computing the recurrence and controlling the loop, it might be more efficient always to run a fixed number of iterations, especially if $y < 2^{16}$ is common. Observe that no more than five iterations are required to reach the final value for any divisor except for an anomaly at $k = 16$.

It can be seen that the maximum error never exceeds 1 after five iterations. Therefore, although the recurrence may not have reached the final value in all cases, there is no harm to correctness if we stop calculating at this point. In fact, the average error after five iterations is quite low. This means that for the majority of divisors, the current value after five iterations is in fact $inv(y)$ and at most one test-subtract will be required to reach the actual quotient.

Since five iterations are always sufficient for $W = 32$, a specialized method can be produced by unrolling the UNR recurrence five times and totally eliminating the loop overhead. Figure 7 shows the result.

4.2 Bit reverse UNR recurrence

It turns out amazingly enough that $bitrev$ and $band$ can be used in place of lsl and clz to compute the decent lower bound initial estimate for the UNR method. In particular, given $y \in U^+$

$$band(bitrev(y), 0 \,\texttt{SUB}\, bitrev(y)) = lsl(1, clz(y))$$

Observe [17] that $band(a, 0 \,\texttt{SUB}\, a)$ isolates the lowest order 1 bit in a. Taking $a = bitrev(y)$ causes this to be the highest order 1 bit in y shifted to the correct place for $lsl(1, clz(y))$. Figure 8 shows the UNR recurrence

```
// decent start
unsigned yr = bitrev(y);
unsigned z = band(yr,0−yr);
// z recurrence
unsigned my = 0−y;
for (;;) {
  unsigned zd = umulh(z,mul(my,z));
  if (zd == 0) break;
  z = z + zd;
}
// q estimate
q = umulh(x,z); r = x − mul(y,q);
// q refinement
if (r >= y) { r = r − y; q = q + 1;
  if (r >= y) { r = r − y; q = q + 1; }
}
```

Figure 8: Bit reverse UNR recurrence method.

method using this approach for computing the initial estimate.

4.3 Leading 9-bit table UNR recurrence

The UNR recurrence requires a decent lower bound initial estimate $z_0 \in DLB_y$ in order to guarantee that the final value $z_n \in TYLB_y$, which enables the quotient estimate to be improved to the actual quotient in no more than two test-subtract iterations. The basic UNR recurrence method uses the initial estimate

$$z_0 = lsl(1, clz(y))$$

which for some divisors is just barely good enough. Providing a better initial estimate will enable the recurrence to reach the final value sooner. Since the recurrence roughly doubles the number of accurate bits on each iteration, providing eight significant bits should allow us to skip the first three iterations.

4.3.1 Leading 9-bit table

We could get an more accurate initial estimate by means of a lookup table using the leading bits of y. Note that the leading bit of any non-zero value is, by definition, 1. So the leading 9 bits of y as well as the leading 9 bits of $inv(y)$ are values in the range $2^8, \ldots, 2^9 - 1$. Since this range comes up a lot, let

$$L_9 = \{2^8, \ldots, 2^9 - 1\}$$

The idea is to feed the leading 9 bits of y through a table to produce the leading 9 bits of a decent lower bound of $inv(y)$. The domain and range of this table is to be L_9.

Depending on the architecture, this might or might not be a useful software approach. Note that the table needs only to contain 256 entries of 8-bit values. This does not seem out of the question as a hardware acceleration.

The problem becomes how to prescale y to a table index in L_9, how to determine the contents of the table that maps from L_9 to L_9, and finally how to postscale the table value from L_9 to a decent lower bound of $inv(y)$. The prescaling and postscaling are fairly straightforward. The big question is how to determine the contents of the table.

Approximation error makes it tricky. Taking the leading 9 bits from $y \geq 2^{10}$ ignores some low order bits of y. In this case, the table index represents a range of values with the same $clz(y)$ and the result from the table will have to produce an estimate z that works as a lower bound on $inv(y)$ for all of them. One approach would be to account for this in the prescaling by rounding up, but that requires additional overhead on every division. A better approach is to account for this in determining the contents of the table.

The following formulation assumes $y \in U^+$, $i \in L_9$, and $W \geq 17$. For prescaling, let

$$ty(y) = lsr(lsl(y, clz(y)), W - 9)$$

For the contents of the table, let

$$t[i] = \begin{cases} lsr(inv(i+1), W - 17) & \text{if } i < 2^9 - 1 \\ 2^8 & \text{otherwise} \end{cases}$$

For postscaling, let

$$zt(i, y) = lsr(lsl(i, W - 9), W - clz(y) - 1)$$

Stringing them all together, let

$$zty(y) = zt(t[ty(y)], y)$$

Theorem 4 *Given $y \in U^+$ it follows that $ty(y) \in L_9$.*

Theorem 5 *Given $i \in L_9$ it follows that $t[i] \in L_9$.*

Theorem 6 *Given $y \in U^+$ it follows that $zty(y) \in DLB_y$.*

```
unsigned unrt [256];

void calc_unrt () {
   unsigned ty = 256;
   for (; ty < 512−1; ty++) {
      unsigned t = lsr(inv(ty + 1),W−17);
      unrt[ty − 256] = t − 256;
   }
   unrt[ty − 256] = 256 − 256;
}
```

Figure 9: Method to compute the leading 9-bit table.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	254	252	250	248	246	244	242	240	238	236	234	233	231	229	227	225
16	224	222	220	218	217	215	213	212	210	208	207	205	203	202	200	199
32	197	195	194	192	191	189	188	186	185	183	182	180	179	178	176	175
48	173	172	170	169	168	166	165	164	162	161	160	158	157	156	154	153
64	152	151	149	148	147	146	144	143	142	141	139	138	137	136	135	134
80	132	131	130	129	128	127	126	125	123	122	121	120	119	118	117	116
96	115	114	113	112	111	110	109	108	107	106	105	104	103	102	101	100
112	99	98	97	96	95	94	93	92	91	90	89	88	88	87	86	85
128	84	83	82	81	80	80	79	78	77	76	75	74	74	73	72	71
144	70	70	69	68	67	66	66	65	64	63	62	62	61	60	59	59
160	58	57	56	56	55	54	53	53	52	51	50	50	49	48	48	47
176	46	46	45	44	43	43	42	41	41	40	39	39	38	37	37	36
192	35	35	34	33	33	32	32	31	30	30	29	28	28	27	27	26
208	25	25	24	24	23	22	22	21	21	20	19	19	18	18	17	17
224	16	15	15	14	14	13	13	12	12	11	10	10	9	9	8	8
240	7	7	6	6	5	5	4	4	3	3	2	2	1	1	0	0

Table 1: Contents of the leading 9-bit table.

4.3.2 Table implementation

As mentioned before, we do not need to waste resources for the first leading bit, since it is always 1. We only implement the lower 8 bits of the table index and value, and translate during prescaling and postscaling. Figure 9 shows a method of computing the leading 9-bit table. Table 1 shows the contents of the table.

Figure 10 shows a method of calculating $inv(y)$. Once a two-y lower bound is obtained using the UNR recurrence, the estimate z is inched upward as far as possible. Note that we must have both $2^W - y * z > y$ and $2^W - z > 1$ in order to be able to increase the estimate z by 1.

```
unsigned inv (unsigned y) {
   // decent start
   unsigned z = lsl(1,clz(y));
   // z recurrence
   unsigned my = 0−y;
   for (;;) {
      unsigned zd = umulh(z,mul(my,z));
      if (zd == 0) break;
      z = z + zd;
   }
   // inch z upward if possible
   while (mul(my,z) > y && 0−z > 1) { z = z+1; }
   return z;
}
```

Figure 10: Method to calculate $inv(y)$.

```
   // table lookup start
   unsigned k = clz(y);
   unsigned ty = lsr( lsl(y,k), W−9 ); // prescaling
   unsigned t = unrt[ ty − 256 ] + 256; // table lookup
   unsigned z = lsr( lsl(t,W−9), W−k−1 ); // postscaling
   // z recurrence
   unsigned my = 0−y;
   for (;;) {
      unsigned zd = umulh(z,mul(my,z));
      if (zd == 0) break;
      z = z + zd;
   }
   // q estimate
   q = umulh(x,z); r = x − mul(y,q);
   // q refinement
   if (r >= y) { r = r − y; q = q + 1;
      if (r >= y) { r = r − y; q = q + 1; }
   }
```

Figure 11: Leading 9-bit table UNR recurrence method.

4.3.3 Implementation

Using a leading 9-bit table to improve the initial estimate produces the UNR recurrence method for computing unsigned integer division shown in Figure 11. The prescaling and postscaling operations are a little painful in software.

It is not clear that a totally software implementation of the leading 9-bit table method would be more efficient

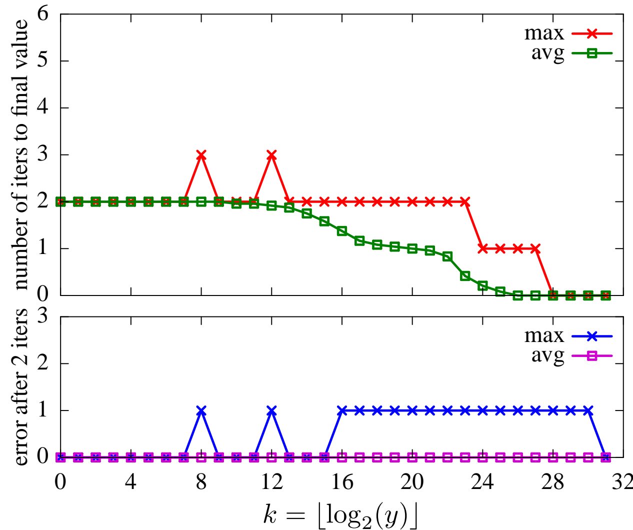

Figure 12: Performance for W=32 of the leading 9-bit table UNR recurrence.

```
// table lookup start
unsigned k = clz(y);
unsigned ty = lsr( lsl(y,k), W−9 ); // prescaling
unsigned t = unrt[ ty − 256 ] + 256; // table lookup
unsigned z = lsr( lsl(t,W−9), W−k−1 ); // postscaling
// z recurrence, 2 iterations
unsigned my = 0−y;
z = z + umulh(z,mul(my,z));
z = z + umulh(z,mul(my,z));
// q estimate
q = umulh(x,z); r = x − mul(y,q);
// q refinement
if (r >= y) { r = r − y; q = q + 1;
  if (r >= y) { r = r − y; q = q + 1; }
}
```

Figure 13: Leading 9-bit table UNR recurrence method for W=32.

than the three iterations of the UNR recurrence that it replaces. However, the code example can be used to see what additional hardware operations would be most useful.

For example, if there were a hardware table lookup operation that used bits $W-2, \ldots, W-9$ as the table index and produced a result that had 1 in bit $W-1$, the table value in bits $W-2, \ldots, W-9$, and zeros in the remaining bits, that operation would be quite useful, even though the software would still have to first shift left by $clz(y)$ and then afterwards shift right by $W - clz(y) - 1$.

4.3.4 Performance for W=32

To investigate the performance of the leading 9-bit table UNR recurrence for the special case of $W = 32$, an experiment similar to the one in Section 4.1.7 was performed except that the error was recorded after two iterations. The results are plotted in Figure 12.

As expected, using the leading 9-bit table generally reduces the number of iterations by three as compared with the basic UNR recurrence. Observe that no more than two iterations are required to reach the final value for any divisor except for anomalous cases of $k = 8$ and $k = 12$.

However, it can be seen that the maximum error never exceeds 1 after two iterations. Therefore, although the recurrence may not have reached the final value in all cases, there is no harm to correctness if we stop calculating at this point. The average error is so close to zero that it cannot be seen on the plot. In fact, for no value of k is the average error after two iterations in excess of 0.02. This means that for the vast majority of divisors, the final value of the recurrence is in fact $inv(y)$.

Since two iterations are always sufficient for $W = 32$, a specialized method can be produced by unrolling the UNR recurrence twice and totally eliminating the loop overhead. Figure 13 shows the result.

4.4 Small divisor table UNR recurrence

Since it seems likely that small divisors are common, a good software approach might be to create a table to store the exact value of $inv(y)$ for $y < 2^K$, for some parameter K. This makes division by small divisors very fast, requiring one $umulh$ to get the quotient estimate, one mul to compute the corresponding remainder, and then at most one test-subtract step to arrive at the actual quotient.

We also have to consider larger divisors. At the cost of doubling the table size, we can store lower bound estimates of $inv(y)$ for $2^K \leq y < 2^{K+1}$. These lower bound estimates are used to provide the initial estimate z_0 for the UNR recurrence in a way similar to that employed in the

```
void calc_unrtk () {
  unsigned i = 0;
  unrtk[i] = 0; i = i+1;
  for (; i < lsl(1,K); i++) {
    unrtk[i] = inv(i);
  }
  for (; i < lsl(2,K)−1; i++) {
    unsigned z = inv(i+1);
    unrtk[i] = lsl(z,clz(z));
  }
  unrtk[i] = lsl(1,clz(1));
}
```

Figure 14: Method to compute the small divisor table.

leading 9-bit table method in Section 4.3. To facilitate the software implementation, we store the lower bound estimates with their leading bit aligned to the highest order bit position.

Figure 14 shows a code example of how to compute the small divisor table. The first half of the table contains exact values of $inv(y)$ for the small divisors and the second half of the table contains lower bound estimates. Note that although the table is for small divisors, it is not necessarily a small table. Depending on the size and speed of memory, the parameter K can be adjusted to scale the table.

Figure 15 gives a code example of how to perform unsigned integer division using the small divisor table. When $y < 2^K$ the first half of the table is accessed, getting the exact value of $inv(y)$. Otherwise, the second half of the table is accessed, getting a lower bound initial estimate z_0 for the UNR recurrence.

As with previous variations, the performance of the small divisor table UNR recurrence was investigated for the special case of $W = 32$. Given the objective of unrolling the recurrence loop to a fixed number of iterations, it is not necessary to iterate the recurrence until it reaches its final value. Rather, what is important is that the recurrence arrive at an estimate of $inv(y)$ that has an error of at most 1 and, for some divisors, the recurrence may attain such an error before it reaches its final value. Consequently, the small divisor table UNR recurrence method was instrumented to determine how many iterations of the recurrence are needed to attain the desired error. For small divisors $y < 2^K$ the number of iterations is zero.

```
unsigned k = clz(y);
if (k > W−K−1) {
  // z = inv(y) table lookup
  unsigned z = unrtk[y];
  // q estimate
  q = umulh(x,z); r = x − mul(y,q);
  // q refinement
  if (r >= y) { r = r − y; q = q + 1; }
}
else {
  // table lookup start
  unsigned ty = lsr( lsl(y,k), W−K−1 ); // prescaling
  unsigned t = unrtk[ ty ]; // table lookup
  unsigned z = lsr( t, W−k−1 ); // postscaling
  // z recurrence
  unsigned my = 0−y;
  for (;;) {
    unsigned zd = umulh(z,mul(my,z));
    if (zd == 0) break;
    z = z + zd;
  }
  // q estimate
  q = umulh(x,z); r = x − mul(y,q);
  // q refinement
  if (r >= y) { r = r − y; q = q + 1;
    if (r >= y) { r = r − y; q = q + 1; }
  }
}
```

Figure 15: Small divisor table UNR recurrence method.

The instrumented method was run once for every divisor $y \in \{1, \ldots, 2^{32} - 1\}$. The divisors were grouped into buckets based on $k = \lfloor \log_2(y) \rfloor$ and the maximum number of iterations calculated for each bucket. This experiment was repeated for each of the parameter values $K \in \{1, \ldots, 16\}$. The results are plotted in Figure 16.

Looking at the plot, it can be seen that in general more iterations are required when $k = \lfloor \log_2(y) \rfloor$ has an intermediate value and fewer iterations are required when k takes an extreme value. This is because the small values of k are handled by looking up the exact value of $inv(y)$ and the larger values of k require fewer iterations since $inv(y)$ is limited to fewer bits.

Table 2 gives the smallest K and table size for a given maximum number of iterations required to reach error at most 1 for any divisor y when using the small divisor ta-

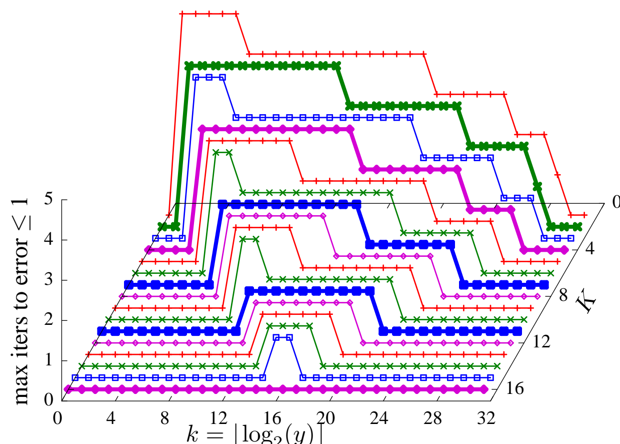

Figure 16: Performance for W=32 of the small divisor table UNR recurrence, for $K \in \{1, \ldots, 16\}$.

max iters	K	table size
4	2	32 B
3	4	128 B
2	7	1024 B
1	11	16 KB
0	16	512 KB

Table 2: Smallest K and corresponding size of the small divisor table for a given maximum number of iterations of the UNR recurrence for W=32.

ble. The curves for these values of K are drawn with thicker lines in Figure 16 for emphasis.

With $K = 16$ no iterations are needed at all. The initial estimate obtained from the table has an error of at most 1. Setting $K = 16$ results in the table containing 2^{17} unsigned integers, which occupies 512 KB (W=32). For some applications this may be a reasonable table size.

Based on a choice of K, a specialized method can be produced for $W = 32$ by unrolling the recurrence loop the required number of times, or, in the case of $K = 16$, deleting the recurrence loop entirely.

5 Conclusion

We presented various methods of performing single precision unsigned integer division in software. Several variations of the standard test-subtract-shift approach were elucidated, including one in which a single operation performs the entire iteration. Large dividends present a difficulty to naive implementations of some of the variations.

Another method and variants that were elucidated are based on a novel adaptation of the Newton-Raphson recurrence to the domain of unsigned integers. The primary insight is to employ lower bound approximations so that all calculated results fall within the range of unsigned integers. The recurrence requires one mul and one $umulh$ operation per iteration. By using a table of initial approximations, the number of iterations can be reduced significantly. A range of table sizes were investigated.

Acknowledgements

Thanks to John Davis and Niklaus Wirth for comments on an earlier version of this paper.

References

[1] R. Alverson. Integer division using reciprocals. In *Proceedings of the Tenth Symposium on Computer Arithmetic*, pages 186–190. IEEE Computer Society Press, 1991.

[2] J. J. F. Cavanagh. *Digital Computer Arithmetic: Design and Implementation*. McGraw-Hill, Inc., 1984.

[3] Y.-T. Cheng. TMS320C6000 integer division. Application Report SPRA707, Texas Instruments, October 2000. `http://focus.ti.com.cn/cn/lit/an/spra707/spra707.pdf`.

[4] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufman, 2004.

[5] R. E. Goldschmidt. Applications of division by convergence. Master's thesis, Dept. of Electrical Engineering, MIT, June 1964.

[6] T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 61–72, New York, NY, USA, 1994. ACM.

[7] D. L. Harris, S. F. Oberman, and M. A. Horowitz. SRT division architectures and implementations. In *Proceeding of the 13th IEEE Symposium on Computer Arithmetic*, pages 18–25. IEEE Computer Society Press, 1997.

[8] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1969.

[9] I. Koren. *Computer Arithmetic Algorithms (2nd Edition)*. A K Peters, 2001.

[10] H. Nikmehr, B. Phillips, and C.-C. Lim. A fast radix-4 floating-point divider with quotient digit selection by comparison multiples. *Comput. J.*, 50(1):81–92, 2007.

[11] S. F. Oberman and M. J. Flynn. An analysis of division algorithms and implementations. Technical Report CSL-TR-95-675, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, July 1995. `ftp://reports.stanford.edu/pub/cstr/reports/csl/tr/95/675/CSL-TR-95-675.pdf`.

[12] S. F. Oberman and M. J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, 1997.

[13] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.

[14] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface, Second Edition*, chapter 4.7 Division, pages 265–274. Morgan Kaufmann, 1997.

[15] D. Seal, editor. *ARM Architecture Reference Manual (2nd Edition)*. Addison-Wesley, January 2001.

[16] Texas Instruments. *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*, February 2008. Literature Number SPRU732G. `http://focus.ti.com/lit/ug/spru732g/spru732g.pdf`.

[17] H. S. Warren, Jr. Functions realizable with word-parallel logical and two's-complement addition instructions. *Commun. ACM*, 20(6):439–441, 1977.

[18] N. Wirth. *Systematic Programming: An Introduction*. Prentice-Hall, 1973.

A Proofs

Theorem 1 *Given $y \in U^+$, $z_i \in LB_y$, and the UNR recurrence*

$$z_{i+1} = z_i + \left\lfloor \frac{z_i * (2^W - y * z_i)}{2^W} \right\rfloor$$

it follows that $z_i \leq z_{i+1}$ and $z_{i+1} \in LB_y$.

Proof Clearly z_{i+1} is an integer. Since $z_i \in LB_y$ we have $1 \leq y * z_i < 2^W$. Take $\epsilon = 2^W - y * z_i$. Observe that $0 < \epsilon < 2^W$. Hence

$$\begin{aligned} z_{i+1} &= z_i + \left\lfloor \frac{z_i * (2^W - y * z_i)}{2^W} \right\rfloor \\ &= z_i + \left\lfloor \frac{z_i * \epsilon}{2^W} \right\rfloor \\ &\geq z_i \end{aligned}$$

Since $1 \leq y * z_i$ it follows that $1 \leq y * z_{i+1}$. We also have

$$\begin{aligned} z_{i+1} &= z_i + \left\lfloor \frac{z_i * \epsilon}{2^W} \right\rfloor \\ &\leq z_i + \frac{z_i * \epsilon}{2^W} \\ &= \frac{z_i * (2^W + \epsilon)}{2^W} \end{aligned}$$

Multiplying by y gives us

$$\begin{aligned} y * z_{i+1} &\leq \frac{y * z_i * (2^W + \epsilon)}{2^W} \\ &= \frac{(2^W - \epsilon) * (2^W + \epsilon)}{2^W} \\ &= \frac{(2^W)^2 - \epsilon^2}{2^W} \\ &= 2^W - \epsilon^2 / 2^W \\ &< 2^W \end{aligned}$$

Since $1 \leq y$ we have $z_{i+1} < 2^W$. Therefore $z_{i+1} \in LB_y$. **QED**

Theorem 2 *Given $z_0 \in DLB_y$ and computing z_1, z_2, and so on using the UNR recurrence, it follows that the final value $z_n \in TYLB_y$.*

Proof Since z_n is the final value, $z_n = z_{n+1}$ and hence

$$\left\lfloor \frac{z_n * (2^W - y * z_n)}{2^W} \right\rfloor = 0$$

Therefore

$$z_n * (2^W - y * z_n) < 2^W$$

Multiplying by $y/2^{W-1}$ gives us

$$\frac{y * z_n * (2^W - y * z_n)}{2^{W-1}} < 2 * y$$

We started with $2^{W-1} \leq y * z_0$ since $z_0 \in DLB_y$. The UNR recurrence has $z_i \leq z_{i+1}$ for all i. So it must be the case that $2^{W-1} \leq y * z_n$. Therefore

$$\begin{aligned} 2^W - y * z_n &= \frac{2^{W-1} * (2^W - y * z_n)}{2^{W-1}} \\ &\leq \frac{y * z_n * (2^W - y * z_n)}{2^{W-1}} \\ &< 2 * y \end{aligned}$$

Hence $2^W - 2 * y < y * z_n$. The UNR recurrence guarantees $y * z_n < 2^W$. **QED**

It may be observed that the final value z_n is slightly better than a two-y lower bound, since it satisfies strict inequality rather than just the non-strict inequality required of a two-y lower bound. However, this point seems to be of no consequence, other than slightly improving the probability of getting a closer lower bound.

Theorem 3 *Given $z \in TYLB_y$, $x \in U$, and $q = \lfloor x/y \rfloor$ it follows that*

$$q - 2 \leq \left\lfloor \frac{x * z}{2^W} \right\rfloor \leq q$$

Proof We prove the inequalities one at a time. First, since $2^W > x \geq q * y$ and $y * z \geq 2^W - 2 * y$, we have

$$\begin{aligned} \frac{x * z}{2^W} &\geq \frac{q * y * z}{2^W} \\ &\geq \frac{q * (2^W - 2 * y)}{2^W} \\ &= q - \frac{2 * q * y}{2^W} \\ &\geq q - \frac{2 * x}{2^W} \\ &> q - \frac{2 * 2^W}{2^W} \\ &= q - 2 \end{aligned}$$

Therefore

$$\left\lfloor \frac{x * z}{2^W} \right\rfloor \geq q - 2$$

Second, since $x < (q + 1) * y$ and $y * z < 2^W$, we have

$$\begin{aligned} \frac{x * z}{2^W} &< \frac{(q + 1) * y * z}{2^W} \\ &< \frac{(q + 1) * 2^W}{2^W} \\ &= q + 1 \end{aligned}$$

Therefore

$$\left\lfloor \frac{x * z}{2^W} \right\rfloor \leq q$$

QED

Theorem 4 *Given $y \in U^+$ it follows that $ty(y) \in L_9$.*

Proof The computation $lsl(y, clz(y))$ aligns the leading 1 bit of y in the leftmost position, that is, at 2^{W-1}. Then $lsr(\ldots, W - 9)$ shifts the value right causing the leading 1 bit to move from 2^{W-1} to 2^8. **QED**

Theorem 5 *Given $i \in L_9$ it follows that $t[i] \in L_9$.*

Proof If $i = 2^9 - 1$ then $t[i] = 2^8$. Otherwise $2^8 \leq i < 2^9 - 1$. In this case observe that

$$2^{W-8} * (i + 1) > 2^{W-8} * 2^8 = 2^W$$

So $inv(i+1) < 2^{W-8}$. Hence

$$\begin{aligned} t[i] &= lsr(inv(i+1), W-17) \\ &= \left\lfloor \frac{inv(i+1)}{2^{W-17}} \right\rfloor \\ &\leq \frac{inv(i+1)}{2^{W-17}} \\ &< \frac{2^{W-8}}{2^{W-17}} \\ &= 2^9 \end{aligned}$$

Going the other way, observe that

$$2^{W-9} * (i+1) < 2^{W-9} * 2^9 = 2^W$$

So $inv(i+1) \geq 2^{W-9}$. Hence

$$\begin{aligned} t[i] &= lsr(inv(i+1), W-17) \\ &= \left\lfloor \frac{inv(i+1)}{2^{W-17}} \right\rfloor \\ &\geq \left\lfloor \frac{2^{W-9}}{2^{W-17}} \right\rfloor \\ &= \lfloor 2^8 \rfloor \\ &= 2^8 \end{aligned}$$

Therefore $2^8 \leq t[i] < 2^9$. **QED**

Lemma 1 *Given $i \in L_9$ and $y \in U^+$ it follows that $2^{W-1} \leq y * zt(i,y)$.*

Proof Since $i \in L_9$ we have $2^8 \leq i < 2^9$. The computation $lsl(i, W-9)$ aligns the leading 1 bit of i in the leftmost position, that is, at 2^{W-1}. Then $lsr(\ldots, W - clz(y) - 1)$ shifts the value right causing the leading 1 bit to move from 2^{W-1} to $2^{clz(y)}$. This gives us

$$zt(i,y) \geq 2^{clz(y)}$$

Recall that $y \geq 2^{W-clz(y)-1}$. Hence

$$\begin{aligned} 2^{W-1} &= 2^{W-clz(y)-1} * 2^{clz(y)} \\ &\leq y * zt(i,y) \end{aligned}$$

QED

Lemma 2 *Given $y \in U^+$ it follows that*

$$zty(y) \leq t[ty(y)] * 2^{clz(y)-8}$$

Proof Let $j = t[ty(y)]$. From Theorem 4 and Theorem 5 it follows that $j \in L_9$. Hence $j * 2^{W-9} < 2^W$ and therefore $lsl(j, W-9) = j * 2^{W-9}$. Consequently

$$\begin{aligned} zty(y) &= zt(t[ty(y)], y) \\ &= zt(j,y) \\ &= lsr(lsl(j, W-9), W - clz(y) - 1) \\ &= lsr(j * 2^{W-9}, W - clz(y) - 1) \\ &= \left\lfloor \frac{j * 2^{W-9}}{2^{W-clz(y)-1}} \right\rfloor \\ &= \left\lfloor j * 2^{clz(y)-8} \right\rfloor \\ &\leq j * 2^{clz(y)-8} \\ &= t[ty(y)] * 2^{clz(y)-8} \end{aligned}$$

QED

Theorem 6 *Given $y \in U^+$ it follows that $zty(y) \in DLB_y$.*

Proof From Theorem 4, Theorem 5, and Lemma 1 we have

$$2^{W-1} \leq y * zty(y)$$

Next we will show $y * zty(y) < 2^W$. Recall that

$$2^{W-clz(y)-1} \leq y < 2^{W-clz(y)}$$

Let $i = ty(y)$. There are two cases.
First case. Suppose $i = 2^9 - 1$. Then we have $t[ty(y)] = 2^8$ and Lemma 2 gives us

$$zty(y) \leq 2^8 * 2^{clz(y)-8} = 2^{clz(y)}$$

Hence

$$y * zty(y) \leq y * 2^{clz(y)} < 2^{W-clz(y)} * 2^{clz(y)} = 2^W$$

This completes the first case.
Second case. Otherwise $i < 2^9 - 1$. We have

$$\begin{aligned} i &= ty(y) \\ &= lsr(lsl(y, clz(y)), W-9) \\ &= \left\lfloor \frac{y * 2^{clz(y)}}{2^{W-9}} \right\rfloor \\ &> \frac{y * 2^{clz(y)}}{2^{W-9}} - 1 \end{aligned}$$

Hence

$$i + 1 > \frac{y * 2^{clz(y)}}{2^{W-9}}$$

Therefore

$$\begin{aligned}
2^W &> (i+1) * inv(i+1) \\
&> \frac{y * 2^{clz(y)}}{2^{W-9}} * inv(i+1) \\
&= y * 2^{clz(y)-8} * \frac{inv(i+1)}{2^{W-17}} \\
&\geq y * 2^{clz(y)-8} * \left\lfloor \frac{inv(i+1)}{2^{W-17}} \right\rfloor \\
&= y * 2^{clz(y)-8} * t[i] \\
&= y * 2^{clz(y)-8} * t[ty(y)] \\
&\geq y * zty(y)
\end{aligned}$$

The last step comes from Lemma 2. This completes the second case. Therefore $y * zty(y) < 2^W$. Since both inequalities are proved, we have $zty(y) \in DLB_y$. **QED**