CHAPTER 10

INTEGER DIVISION BY CONSTANTS

Insert this material at the end of page 201, just before the poem on page 202.

10–17 Methods Not Using Multiply High

In this section we consider some methods for dividing by constants that do not use the *multiply high* instruction, or a multiplication instruction that gives a double-word result. We show how to change division by a constant into a sequence of *shift* and *add* instructions, or *shift*, *add*, and *multiply* for more compact code.

Unsigned Division

For these methods, unsigned division is simpler than signed division, so we deal with unsigned division first. One method is to use the techniques given that use the *multiply high* instruction, but use the code shown in Figure 8-2 on page 132 to do the *multiply high* operation. Figure 10–5 shows how this works out for the case of (unsigned) division by 3. This is a combination of the code on page 178 and

```
unsigned divu3(unsigned n) {
   unsigned n0, n1, w0, w1, w2, t, q;

   n0 = n & 0xFFFF;
   n1 = n >> 16;
   w0 = n0*0xAAAB;
   t  = n1*0xAAAB + (w0 >> 16);
   w1 = t & 0xFFFF;
   w2 = t >> 16;
   w1 = n0*0xAAAA + w1;
   q = n1*0xAAAA + w2 + (w1 >> 16);
   return q >> 1;
}
```

FIGURE 10–5. Unsigned divide by 3 using simulated *multiply high unsigned*.

Figure 8-2 with "`int`" changed to "`unsigned`." The code is 15 instructions, including four multiplications. The multiplications are by large constants and would take quite a few instructions if converted to *shift*'s and *add*'s. Very similar code can be devised for the signed case. This method is not particularly good and won't be discussed further.

Another method [GLS1] is to compute in advance the reciprocal of the divisor, and multiply the dividend by that with a series of *shift right* and *add* instructions. This gives an approximation to the quotient. It is merely an approximation because the reciprocal of the divisor (which we assume is not an exact power of two) is not expressed exactly in 32 bits, and also because each *shift right* discards bits of the dividend. Next the remainder with respect to the approximate quotient is computed, and that is divided by the divisor to form a correction, which is added to the approximate quotient, giving the exact quotient. The remainder is generally small compared to the divisor (a few multiples thereof), so there is often a simple way to compute the correction without using a *divide* instruction.

To illustrate this method, consider dividing by 3, i.e., computing $\lfloor n/3 \rfloor$ where $0 \le n < 2^{32}$. The reciprocal of 3, in binary, is approximately

$$0.0101\ 0101\ 0101\ 0101\ 0101\ 0101\ 0101\ 0101.$$

To compute the approximate product of that and n, we could use

$$\boldsymbol{q} \leftarrow (\boldsymbol{n} \overset{u}{\gg} 2) + (\boldsymbol{n} \overset{u}{\gg} 4) + (\boldsymbol{n} \overset{u}{\gg} 6) + \ldots + (\boldsymbol{n} \overset{u}{\gg} 30) \tag{32}$$

(29 instructions; the last 1 in the reciprocal is ignored because it would add the term $\boldsymbol{n} \overset{u}{\gg} 32$, which is obviously 0). However, the simple repeating pattern of 1's and 0's in the reciprocal permits a method that is both faster (9 instructions) and more accurate:

$$\begin{aligned}
\boldsymbol{q} &\leftarrow (\boldsymbol{n} \overset{u}{\gg} 2) + (\boldsymbol{n} \overset{u}{\gg} 4) \\
\boldsymbol{q} &\leftarrow \boldsymbol{q} + (\boldsymbol{q} \overset{u}{\gg} 4) \\
\boldsymbol{q} &\leftarrow \boldsymbol{q} + (\boldsymbol{q} \overset{u}{\gg} 8) \\
\boldsymbol{q} &\leftarrow \boldsymbol{q} + (\boldsymbol{q} \overset{u}{\gg} 16)
\end{aligned} \tag{33}$$

To compare these methods for their accuracy, consider the bits that are shifted out by each term of (32), if $\boldsymbol{n}$ is all 1-bits. The first term shifts out two 1-bits, the next four 1-bits, and so on. Each of these contributes an error of almost 1 in the least significant bit. Since there are 16 terms (counting the term we ignored), the shifts contribute an error of almost 16. There is an additional error due to the fact that the reciprocal is truncated to 32 bits; it turns out that the maximum total error is 16.

For procedure (33), each right shift also contributes an error of almost 1 in the least significant bit. But there are only five shift operations. They contribute an error of almost 5, and there is a further error due to the fact that the reciprocal is truncated to 32 bits; it turns out that the maximum total error is 5.

After computing the estimated quotient $\boldsymbol{q}$, the remainder $\boldsymbol{r}$ is computed from

$$\boldsymbol{r} \leftarrow \boldsymbol{n} - \boldsymbol{q} * 3.$$

The remainder cannot be negative, because $\boldsymbol{q}$ is never larger than the exact quotient. We need to know how large $\boldsymbol{r}$ can be, to devise the simplest possible method for computing $\boldsymbol{r} \overset{u}{\div} 3$. In general, for a divisor d and an estimated quotient q too low by k, the remainder will range from $k * d$ to $k * d + d - 1$. (The upper limit is conservative, it may not actually be attained.) Thus, using (33), for which $\boldsymbol{q}$ is too low by at most 5, we expect the remainder to be at most 5*3 + 2 = 17. Experiment reveals that it is actually at most 15. Thus for the correction we must compute (exactly)

$$\boldsymbol{r} \overset{u}{\div} 3, \quad \text{for } 0 \le \boldsymbol{r} \le 15.$$

Since $\boldsymbol{r}$ is small compared to the largest value that a register can hold, this can be approximated by multiplying $\boldsymbol{r}$ by some approximation to 1/3 of the form a/b where b is a power of 2. This is easy to compute because the division is simply a shift. The value of a/b must be slightly larger than 1/3, so that after shifting the result will agree with truncated division. A sequence of such approximations is:

1/2, 2/4, 3/8, 6/16, 11/32, 22/64, 43/128, 86/256, 171/512, 342/1024, ….

Usually the smaller fractions in the sequence are easier to compute, so we choose the smallest one that works; in the case at hand this is 11/32. Therefore the final, exact, quotient is given by

$$\boldsymbol{q} \leftarrow \boldsymbol{q} + (11 * \boldsymbol{r} \overset{u}{\gg} 5).$$

The solution involves two multiplications by small numbers (3 and 11); these can be changed to *shift*'s and *add*'s.

Figure 10–6 shows the entire solution in C. As shown, it consists of 14 instructions, including two multiplications. If the multiplications are changed to *shift*'s and *add*'s, it amounts to 18 elementary instructions. However, if it is desired to avoid the multiplications, then either alternative `return` statement shown gives a solution in 17 elementary instructions. Alternative 2 has just a little instruction-level parallelism, but in truth, this method generally has very little of that.

A more accurate estimate of the quotient can be obtained by changing the first executable line to

```
q = (n >> 1) + (n >> 3);
```

(which makes `q` too large by a factor of 2, but it has one more bit of accuracy), and then inserting just before the assignment to `r`,

```
q = q >> 1;
```

```
unsigned divu3(unsigned n) {
   unsigned q, r;

   q = (n >> 2) + (n >> 4);      // q = n*0.0101 (approx).
   q = q + (q >> 4);             // q = n*0.01010101.
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*3;                  // 0 <= r <= 15.
   return q + (11*r >> 5);       // Returning q + r/3.
// return q + (5*(r + 1) >> 4);          // Alternative 1.
// return q + ((r + 5 + (r << 2)) >> 4);// Alternative 2.
}
```

FIGURE 10–6. Unsigned divide by 3.

With this variation, the remainder is at most 9. However, there does not seem to be any better code for calculating $r \overset{u}{\div} 3$ with r limited to 9, than there is for r limited to 15 (four elementary instructions in either case). Thus using the idea would cost one instruction. This possibility is mentioned because it *does* give a code improvement for most divisors.

Figure 10–7 shows two variations of this method for dividing by 5. The

```
unsigned divu5a(unsigned n) {
   unsigned q, r;

   q = (n >> 3) + (n >> 4);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*5;
   return q + (13*r >> 6);
}
```

```
unsigned divu5b(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*5;
   return q + (7*r >> 5);
// return q + (r>4) + (r>9);
}
```

FIGURE 10–7. Unsigned divide by 5.

reciprocal of 5, in binary, is

$$0.0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011.$$

As in the case of division by 3, the simple repeating pattern of 1's and 0's allows a fairly efficient and accurate computing of the quotient estimate. The estimate of the quotient computed by the code on the left can be off by at most 5, and it turns out that the remainder is at most 25. The code on the right retains two additional bits of accuracy in computing the quotient estimate, which is off by at most 2. The remainder in this case is at most 10. The smaller maximum remainder permits

approximating 1/5 by 7/32 rather than 13/64, which gives a slightly more efficient program if the multiplications are done by *shift*'s and *add*'s. The instruction counts are, for the code on the left: 14 instructions including two multiplications, or 18 elementary instructions; for the code on the right: 15 instructions including two multiplications, or 17 elementary instructions. The alternative code in the `return` statement is useful only if your machine has comparison predicate instructions. It doesn't reduce the instruction count, but merely has a little instruction-level parallelism.

For division by 6, the divide by 3 code can be used, followed by a *shift right* of 1. However, the extra instruction can be saved by doing the computation directly, using the binary approximation

$$4/6 \approx 0.1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010.$$

The code is shown in Figure 10–8. The version on the left multiplies by an approximation to 1/6 and then corrects with a multiplication by 11/64. The version on the right takes advantage of the fact that by multiplying by an approximation to 4/6, the quotient estimate is off by only 1 at most. This permits simpler code for the correction; it simply adds 1 to `q` if `r` ≥ 6. The code in the second `return` statement is appropriate if the machine has the comparison predicate instructions. Function `divu6b` is 15 instructions, including one *multiply*, as shown, or 17 elementary instructions if the multiplication by 6 is changed to *shift*'s and *add*'s.

```
unsigned divu6a(unsigned n) {
   unsigned q, r;

   q = (n >> 3) + (n >> 5);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*6;
   return q + (11*r >> 6);
}
```

```
unsigned divu6b(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*6;
   return q + ((r + 2) >> 3);
// return q + (r > 5);
}
```

FIGURE 10–8. Unsigned divide by 6.

For larger divisors, usually it seems to be best to use an approximation to $1/d$ that is shifted left so that its most significant bit is 1. It seems that the quotient is then off by at most 1 usually (possibly always, this writer does not know), which permits efficient code for the correction step. Figure 10–9 shows code for dividing by 7 and 9, using the binary approximations

$$4/7 \approx 0.1001\ 0010\ 0100\ 1001\ 0010\ 0100\ 1001\ 0010, \quad \text{and}$$
$$8/9 \approx 0.1110\ 0011\ 1000\ 1110\ 0011\ 1000\ 1110\ 0011.$$

If the multiplications by 7 and 9 are expanded into *shift*'s and *add*'s, these functions take 16 and 15 elementary instructions, respectively.

```
unsigned divu7(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 4);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 2;
   r = n - q*7;
   return q + ((r + 1) >> 3);
// return q + (r > 6);
}
```

```
unsigned divu9(unsigned n) {
   unsigned q, r;

   q = n - (n >> 3);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*9;
   return q + ((r + 7) >> 4);
// return q + (r > 8);
}
```

FIGURE 10–9. Unsigned divide by 7 and 9.

Figures 10–10 and 10–11 show code for dividing by 10, 11, 12, and 13. These are based on the binary approximations:

$$8/10 \approx 0.1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100,$$
$$8/11 \approx 0.1011\ 1010\ 0010\ 1110\ 1000\ 1011\ 1010\ 0010,$$
$$8/12 \approx 0.1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010, \quad \text{and}$$
$$8/13 \approx 0.1001\ 1101\ 1000\ 1001\ 1101\ 1000\ 1001\ 1101.$$

If the multiplications are expanded into *shift*'s and *add*'s, these functions take 17, 20, 17, and 20 elementary instructions, respectively.

```
unsigned divu10(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*10;
   return q + ((r + 6) >> 4);
// return q + (r > 9);
}
```

```
unsigned divu11(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 2) -
      (n >> 5) + (n >> 7);
   q = q + (q >> 10);
   q = q + (q >> 20);
   q = q >> 3;
   r = n - q*11;
   return q + ((r + 5) >> 4);
// return q + (r > 10);
```

FIGURE 10–10. Unsigned divide by 10 and 11.

```
unsigned divu12(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*12;
   return q + ((r + 4) >> 4);
// return q + (r > 11);
}
```

```
unsigned divu13(unsigned n) {
   unsigned q, r;

   q = (n>>1) + (n>>4);
   q = q + (q>>4) + (q>>5);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*13;
   return q + ((r + 3) >> 4);
// return q + (r > 12);
}
```

FIGURE 10–11. Unsigned divide by 12 and 13.

The case of dividing by 13 is instructive because it shows how you must look for repeating strings in the binary expansion of the reciprocal of the divisor. The first assignment sets `q` equal to `n`∗0.1001. The second assignment to `q` adds `n`∗0.00001001 and `n`∗0.000001001. At this point `q` is (approximately) equal to `n`∗0.100111011. The third assignment to `q` adds in repetitions of this pattern. It sometimes helps to use subtraction, as in the case of `divu9` above. However, you must use care with subtraction because it may cause the quotient estimate to be too large, in which case the remainder is negative, and the method breaks down. It is quite complicated to get optimal code, and we don't have a general cook-book method that you can put in a compiler to handle any divisor.

The examples above are able to economize on instructions because the reciprocals have simple repeating patterns, and because the multiplication in the computation of the remainder `r` is by a small constant, which can be done with only a few *shift*'s and *add*'s. One might wonder how successful this method is for larger divisors. To roughly assess this, Figures 10–12 and 10–13 show code for dividing by 100 and by 1000 (decimal). The relevant reciprocals are

$$64/100 \approx 0.1010\ 0011\ 1101\ 0111\ 0000\ 1010\ 0011\ 1101 \quad \text{and}$$
$$512/1000 \approx 0.1000\ 0011\ 0001\ 0010\ 0110\ 1110\ 1001\ 0111.$$

If the multiplications are expanded into *shift*'s and *add*'s, these functions take 25 and 23 elementary instructions, respectively.

```
unsigned divu100(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 3) + (n >> 6) - (n >> 10) +
       (n >> 12) + (n >> 13) - (n >> 16);
   q = q + (q >> 20);
   q = q >> 6;
   r = n - q*100;
   return q + ((r + 28) >> 7);
// return q + (r > 99);
}
```

FIGURE 10–12. Unsigned divide by 100.

```
unsigned divu1000(unsigned n) {
   unsigned q, r, t;

   t = (n >> 7) + (n >> 8) + (n >> 12);
   q = (n >> 1) + t + (n >> 15) + (t >> 11) + (t >> 14);
   q = q >> 9;
   r = n - q*1000;
   return q + ((r + 24) >> 10);
// return q + (r > 999);
}
```

FIGURE 10–13. Unsigned divide by 1000.

In the case of dividing by 1000, the least significant 8 bits of the reciprocal estimate are nearly ignored. The code of Figure 10–13 replaces the binary 1001 0111 with 0100 0000, and still the quotient estimate is within 1 of the true quotient. Thus it appears that although large divisors might have very little repetition in the binary representation of the reciprocal estimate, at least some bits can be ignored, which helps hold down the number of *shift*'s and *add*'s required to compute the quotient estimate.

This section has shown in a somewhat imprecise way how unsigned division by a constant can be reduced to a sequence of, typically, about 20 elementary instructions. It is nontrivial to get an algorithm that generates these code sequences that is suitable for incorporation into a compiler because of three difficulties in getting optimal code:

1. it is necessary to search the reciprocal estimate bit string for repeating patterns,
2. negative terms (as in `divu10` and `divu100`) can be used sometimes, but the error analysis required to determine just when they can be used is difficult, and
3. sometimes some of the least significant bits of the reciprocal estimate can be ignored (how many?)

Another difficulty for some target machines is that there are many variations on the code examples given that have more instructions but that would execute faster on a machine with multiple shift and add units.

The code of Figures 10–5 through 10–13 has been tested for all 2^{32} values of the dividends.

Signed Division

The methods given above can be made to apply to signed division. The *right shift*'s in computing the quotient estimate become *signed right shift*'s, which are equivalent to "floor division" by powers of 2. Thus the quotient estimate is too low (algebraically), so the remainder is nonnegative, as in the unsigned case.

The code most naturally computes the floor division result, so we need a correction to make it compute the conventional truncated toward 0 result. This can be done with three computational instructions by adding $d - 1$ to the dividend if the dividend is negative. For example, if the divisor is 6, the code begins with (the *shift* here is a *signed shift*)

```
n = n + (n>>31 & 5);
```

Other than this, the code is very similar to that of the unsigned case. The number of elementary operations required is usually three more than in the corresponding unsigned division function. Several examples are given below. All have been exhaustively tested.

```
int divs3(int n) {
   int q, r;

   n = n + (n>>31 & 2);           // Add 2 if n < 0.
   q = (n >> 2) + (n >> 4);       // q = n*0.0101 (approx).
   q = q + (q >> 4);              // q = n*0.01010101.
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*3;                   // 0 <= r <= 14.
   return q + (11*r >> 5);        // Returning q + r/3.
// return q + (5*(r + 1) >> 4);          // Alternative 1.
// return q + ((r + 5 + (r << 2)) >> 4);// Alternative 2.
}
```

FIGURE 10–14. Signed divide by 3.

```
int divs5(int n) {
   int q, r;

   n = n + (n>>31 & 4);
   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*5;
   return q + (7*r >> 5);
// return q + (r>4) + (r>9);
}
```

```
int divs6(int n) {
   int q, r;

   n = n + (n>>31 & 5);
   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*6;
   return q + ((r + 2) >> 3);
// return q + (r > 5);
}
```

FIGURE 10–15. Signed divide by 5 and 6.

```
int divs7(int n) {
   int q, r;

   n = n + (n>>31 & 6);
   q = (n >> 1) + (n >> 4);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 2;
   r = n - q*7;
   return q + ((r + 1) >> 3);
// return q + (r > 6);
}
```

```
int divs9(int n) {
   int q, r;

   n = n + (n>>31 & 8);
   q = (n >> 1) + (n >> 2) +
       (n >> 3);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*9;
   return q + ((r + 7) >> 4);
// return q + (r > 8);
}
```

FIGURE 10–16. Signed divide by 7 and 9.

```
int divs10(int n) {
   int q, r;

   n = n + (n>>31 & 9);
   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*10;
   return q + ((r + 6) >> 4);
// return q + (r > 9);
}
```

```
int divs11(int n) {
   int q, r;

   n = n + (n>>31 & 10);
   q = (n >> 1) + (n >> 2) -
       (n >> 5) + (n >> 7);
   q = q + (q >> 10);
   q = q + (q >> 20);
   q = q >> 3;
   r = n - q*11;
   return q + ((r + 5) >> 4);
// return q + (r > 10);
}
```

FIGURE 10–17. Signed divide by 10 and 11.

```
int divs12(int n) {
   int q, r;

   n = n + (n>>31 & 11);
   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*12;
   return q + ((r + 4) >> 4);
// return q + (r > 11);
}
```

```
int divs13(int n) {
   int q, r;

   n = n + (n>>31 & 12);
   q = (n>>1) + (n>>4);
   q = q + (q>>4) + (q>>5);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*13;
   return q + ((r + 3) >> 4);
// return q + (r > 12);
}
```

FIGURE 10–18. Signed divide by 12 and 13.

```
int divs100(int n) {
   int q, r;

   n = n + (n>>31 & 99);
   q = (n >> 1) + (n >> 3) + (n >> 6) - (n >> 10) +
       (n >> 12) + (n >> 13) - (n >> 16);
   q = q + (q >> 20);
   q = q >> 6;
   r = n - q*100;
   return q + ((r + 28) >> 7);
// return q + (r > 99);
}
```

FIGURE 10–19. Signed divide by 100.

```
int divs1000(int n) {
   int q, r, t;

   n = n + (n>>31 & 999);
   t = (n >> 7) + (n >> 8) + (n >> 12);
   q = (n >> 1) + t + (n >> 15) + (t >> 11) + (t >> 14) +
       (n >> 26) + (t >> 21);
   q = q >> 9;
   r = n - q*1000;
   return q + ((r + 24) >> 10);
// return q + (r > 999);
}
```

FIGURE 10–20. Signed divide by 1000.

10–18 Remainder by Summing Digits

This section addresses the problem of computing the remainder of division by a constant without computing the quotient. The methods of this section apply only to divisors of the form $2^k \pm 1$, for k an integer greater than or equal to 2, and in most cases the code resorts to a table lookup (an indexed *load* instruction) after a fairly short calculation.

We will make frequent use of the following elementary property of congruences:

THEOREM C. *If* $a \equiv b \pmod{m}$ *and* $c \equiv d \pmod{m}$, *then*

$$\begin{aligned} a + c &\equiv b + d \pmod{m} \quad \text{and} \\ ac &\equiv bd \pmod{m}. \end{aligned}$$

The unsigned case is simpler and is dealt with first.

Unsigned Remainder

For a divisor of 3, multiplying the trivial congruence $1 \equiv 1 \pmod 3$ repeatedly by the congruence $2 \equiv -1 \pmod 3$, we conclude by Theorem C that

$$2^k \equiv \begin{cases} 1 \pmod 3, & k \text{ even}, \\ -1 \pmod 3, & k \text{ odd}. \end{cases}$$

Thus a number n written in binary as $\ldots b_3 b_2 b_1 b_0$ satisfies

$$n = \ldots + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0 \equiv \ldots - b_3 + b_2 - b_1 + b_0 \pmod 3,$$

which is derived by using Theorem C repeatedly. Thus we can alternately add and subtract the bits in the binary representation of the number to obtain a smaller number that has the same remainder upon division by 3. If the sum is negative you must add a multiple of 3, to make it nonnegative. The process can then be repeated until the result is in the range 0 to 2.

The same trick works for finding the remainder after dividing a decimal number by 11.

Thus, if the machine has the *population count* instruction, a function that computes the remainder modulo 3 of an unsigned number `n` might begin with

```
n = pop(n & 0x55555555) - pop(n & 0xAAAAAAAA);
```

However, this can be simplified by using the following surprising identity discovered by Paolo Bonzini [PB]:

$$\text{pop}(\boldsymbol{x} \,\&\, \overline{\boldsymbol{m}}) - \text{pop}(\boldsymbol{x} \,\&\, \boldsymbol{m}) = \text{pop}(\boldsymbol{x} \oplus \boldsymbol{m}) - \text{pop}(\boldsymbol{m}). \tag{34}$$

Proof:

$$\begin{aligned}
&\text{pop}(\boldsymbol{x} \,\&\, \bar{\boldsymbol{m}}) - \text{pop}(\boldsymbol{x} \,\&\, \boldsymbol{m}) \\
&\quad = \text{pop}(\boldsymbol{x} \,\&\, \bar{\boldsymbol{m}}) - (32 - \text{pop}(\overline{\boldsymbol{x} \,\&\, \boldsymbol{m}})) && \text{pop}(\boldsymbol{a}) = 32 - \text{pop}(\bar{\boldsymbol{a}}). \\
&\quad = \text{pop}(\boldsymbol{x} \,\&\, \bar{\boldsymbol{m}}) + \text{pop}(\bar{\boldsymbol{x}} \mid \bar{\boldsymbol{m}}) - 32 && \text{DeMorgan.} \\
&\quad = \text{pop}(\boldsymbol{x} \,\&\, \bar{\boldsymbol{m}}) + \text{pop}(\bar{\boldsymbol{x}} \,\&\, \boldsymbol{m}) + \text{pop}(\bar{\boldsymbol{m}}) - 32 && \text{pop}(\boldsymbol{a} \mid \boldsymbol{b}) = \\
& && \text{pop}(\boldsymbol{a} \,\&\, \bar{\boldsymbol{b}}) + \text{pop}(\boldsymbol{b}). \\
&\quad = \text{pop}((\boldsymbol{x} \,\&\, \bar{\boldsymbol{m}}) \mid (\bar{\boldsymbol{x}} \,\&\, \boldsymbol{m})) + \text{pop}(\bar{\boldsymbol{m}}) - 32 && \text{Disjoint.} \\
&\quad = \text{pop}(\boldsymbol{x} \oplus \boldsymbol{m}) - \text{pop}(\boldsymbol{m})
\end{aligned}$$

Since the references to 32 (the word size) cancel out, the result holds for any word size. Another way to prove (34) is to observe that it holds for $\boldsymbol{x} = \boldsymbol{0}$, and if a 0-bit in $\boldsymbol{x}$ is changed to a 1 where $\boldsymbol{m}$ is 1, then both sides of (34) decrease by 1, and if a 0-bit of $\boldsymbol{x}$ is changed to a 1 where $\boldsymbol{m}$ is 0, then both sides of (34) increase by 1.

Applying (34) to the line of C code above gives

```
n = pop(n ^ 0xAAAAAAAA) - 16;
```

We want to apply this transformation again, until `n` is in the range 0 to 2, if possible. But it is best to avoid producing a negative value of `n`, because the sign bit would not be treated properly on the next round. A negative value can be avoided by adding a sufficiently large multiple of 3 to `n`. Bonzini's code, shown in Figure 10–21, increases the constant by 39. This is larger than necessary to make `n` nonnegative, but it causes `n` to range from –3 to 2 (rather than –3 to 3) after the second round of reduction. This simplifies the code on the `return` statement, which is adding 3 if `n` is negative. The function executes in 11 instructions, counting two to load the large constant.

```
int remu3(unsigned n) {
   n = pop(n ^ 0xAAAAAAAA) + 23;     // Now 23 <= n <= 55.
   n = pop(n ^ 0x2A) - 3;            // Now -3 <= n <= 2.
   return n + (((int)n >> 31) & 3);
}
```

FIGURE 10–21. Unsigned remainder modulo 3, using *population count*.

Figure 10–22 shows a variation that executes in four instructions plus a simple table lookup operation (e.g., and indexed *load byte* instruction).

```
int remu3(unsigned n) {

   static char table[33] = {2, 0,1,2, 0,1,2, 0,1,2,
          0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
          0,1,2, 0,1};

   n = pop(n ^ 0xAAAAAAAA);
   return table[n];
}
```

FIGURE 10–22. Unsigned remainder modulo 3, using *population count* and a table lookup.

To avoid the *population count* instruction, notice that because $4 \equiv 1 \pmod 3$, $4^k \equiv 1 \pmod 3$. A binary number can be viewed as a base 4 number by taking its bits in pairs and interpreting the bits 00 to 11 as a base 4 digit ranging from 0 to 3. The pairs of bits can be summed using the code of Figure 5–2 on page 66, omitting the first executable line (overflow does not occur in the additions). The final sum ranges from 0 to 48, and a table lookup can be used to reduce this to the range 0 to 2. The resulting function is 16 elementary instructions plus an indexed *load*.

There is a similar but slightly better way. As a first step, `n` can be reduced to a smaller number that is in the same congruence class modulo 3 with

```
n = (n >> 16) + (n & 0xFFFF);
```

This splits the number into two 16-bit portions, which are added together. The contribution modulo 3 of the left 16 bits of `n` is not altered by shifting them right 16 positions, because the shifted number, multiplied by 2^{16}, is the original number, and $2^{16} \equiv 1 \pmod 3$. More generally, $2^k \equiv 1 \pmod 3$ if k is even. This is used repeatedly (five times) in the code shown in Figure 10–23. This code is 19 instructions. The instruction count can be reduced by cutting off the digit summing earlier and using an in-memory table lookup, as illustrated in Figure 10–24 (nine instructions plus an indexed *load*). The instruction count can be reduced to six (plus an indexed *load*) by using a table of size 0x2FE = 766 bytes.

```
int remu3(unsigned n) {
   n = (n >> 16) + (n & 0xFFFF);        // Max 0x1FFFE.
   n = (n >>  8) + (n & 0x00FF);        // Max 0x2FD.
   n = (n >>  4) + (n & 0x000F);        // Max 0x3D.
   n = (n >>  2) + (n & 0x0003);        // Max 0x11.
   n = (n >>  2) + (n & 0x0003);        // Max 0x6.
   return (0x0924 >> (n << 1)) & 3;
}
```

FIGURE 10–23. Unsigned remainder modulo 3, digit summing and an in-register lookup.

```
int remu3(unsigned n) {
   static char table[62] = {0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1};

   n = (n >> 16) + (n & 0xFFFF);        // Max 0x1FFFE.
   n = (n >>  8) + (n & 0x00FF);        // Max 0x2FD.
   n = (n >>  4) + (n & 0x000F);        // Max 0x3D.
   return table[n];
}
```

FIGURE 10–24. Unsigned remainder modulo 3, digit summing and an in-memory lookup.

To compute the unsigned remainder modulo 5, the code of Figure 10–25 uses the relations $16^k \equiv 1 \pmod 5$ and $4 \equiv -1 \pmod 5$. It is 21 elementary instructions, assuming the multiplication by 3 is expanded into a *shift* and *add*.

```
int remu5(unsigned n) {
   n = (n >> 16) + (n & 0xFFFF);          // Max 0x1FFFE.
   n = (n >>  8) + (n & 0x00FF);          // Max 0x2FD.
   n = (n >>  4) + (n & 0x000F);          // Max 0x3D.
   n = (n>>4) - ((n>>2) & 3) + (n & 3);   // -3 to 6.
   return (01043210432 >> 3*(n + 3)) & 7; // Octal const.
}
```

FIGURE 10–25. Unsigned remainder modulo 5, digit summing method.

The instruction count can be reduced by using a table, similarly to what is done in Figure 10–24. In fact, the code is identical except the table is:

```
static char table[62] = {0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1};
```

For the unsigned remainder modulo 7, the code of Figure 10–26 uses the relation $8^k \equiv 1 \pmod 7$ (nine elementary instructions plus an indexed *load*).

As a final example, the code of Figure 10–27 computes the remainder of unsigned division by 9. It is based on the relation $8 \equiv -1 \pmod 9$. As shown, it is nine elementary instructions plus an indexed load. The elementary instruction count can be reduced to six by using a table of size 831 (decimal).

```
int remu7(unsigned n) {

   static char table[75] = {0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
             0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
             0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
             0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4};

   n = (n >> 15) + (n & 0x7FFF);         // Max 0x27FFE.
   n = (n >>  9) + (n & 0x001FF);        // Max 0x33D.
   n = (n >>  6) + (n & 0x0003F);        // Max 0x4A.
   return table[n];
}
```

FIGURE 10–26. Unsigned remainder modulo 7, digit summing method.

```
int remu9(unsigned n) {

   int r;
   static char table[75] = {0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2};

   r = (n & 0x7FFF) - (n >> 15);      // FFFE0001 to 7FFF.
   r = (r & 0x01FF) - (r >>  9);      // FFFFFFC1 to 2FF.
   r = (r & 0x003F) + (r >>  6);      // 0 to 4A.
   return table[r];
}
```

FIGURE 10–27. Unsigned remainder modulo 9, digit summing method.

Signed Remainder

The digit summing method can be adapted to compute the remainder resulting from signed division. However, there seems to be no better way than to add a few steps to correct the result of the method as applied to unsigned division. Two corrections are necessary: (1) correct for a different interpretation of the sign bit, and (2) add or subtract a multiple of the divisor d to get the result in the range 0 to $-(d-1)$.

For division by 3, the unsigned remainder code interprets the sign bit of the dividend $\boldsymbol{n}$ as contributing 2 to the remainder (because $2^{31} \bmod 3 = 2$). But for the remainder of signed division the sign bit contributes only 1 (because $(-2^{31}) \bmod 3 = 1$). Therefore we can use the code for unsigned remainder and correct its result by subtracting 1. Then the result must be put in the range 0 to –2. That is, the result of the unsigned remainder code must be mapped as follows:

$$(0, 1, 2) \Rightarrow (-1, 0, 1) \Rightarrow (-1, 0, -2).$$

This adjustment can be done fairly efficiently by subtracting 1 from the unsigned remainder if it is 0 or 1, and subtracting 4 if it is 2 (when the dividend is negative). The code must not alter the dividend `n` because it is needed in this last step.

This procedure can easily be applied to any of the functions given for the unsigned remainder modulo 3. For example, applying it to Figure 10–24 on page 15 gives the function shown in Figure 10–28. It is 13 elementary instructions plus an indexed *load*. The instruction count can be reduced by using a larger table.

```
int rems3(int n) {
   unsigned r;
   static char table[62] = {0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1};

   r = n;
   r = (r >> 16) + (r & 0xFFFF);        // Max 0x1FFFE.
   r = (r >>  8) + (r & 0x00FF);        // Max 0x2FD.
   r = (r >>  4) + (r & 0x000F);        // Max 0x3D.
   r = table[r];
   return r - (((unsigned)n >> 31) << (r & 2));
}
```

FIGURE 10–28. Signed remainder modulo 3, digit summing method.

Figures 10–29 to 10–31 show similar code for computing the signed remainder of division by 5, 7, and 9. All the functions consist of 15 elementary operations plus an indexed *load*. They use signed right shifts and the final adjustment consists of subtracting the modulus if the dividend is negative and the remainder is nonzero. The number of instructions can be reduced by using larger tables.

```
int rems5(int n) {
   int r;
   static char table[62] = {2,3,4, 0,1,2,3,4, 0,1,2,3,4,
             0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
             0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
             0,1,2,3,4, 0,1,2,3};

   r = (n >> 16) + (n & 0xFFFF);   // FFFF8000 to 17FFE.
   r = (r >>  8) + (r & 0x00FF);   // FFFFFF80 to 27D.
   r = (r >>  4) + (r & 0x000F);   // -8 to 53 (decimal).
   r = table[r + 8];
   return r - (((int)(n & -r) >> 31) & 5);
}
```

FIGURE 10–29. Signed remainder modulo 5, digit summing method.

```
int rems7(int n) {
   int r;
   static char table[75] =   {5,6, 0,1,2,3,4,5,6,
     0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
     0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
     0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2};

   r = (n >> 15) + (n & 0x7FFF);   // FFFF0000 to 17FFE.
   r = (r >>  9) + (r & 0x001FF);  // FFFFFF80 to 2BD.
   r = (r >>  6) + (r & 0x0003F);  // -2 to 72 (decimal).
   r = table[r + 2];
   return r - (((int)(n & -r) >> 31) & 7);
}
```

FIGURE 10–30. Signed remainder modulo 7, digit summing method.

```
int rems9(int n) {
   int r;
   static char table[75] = {7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0};

   r = (n & 0x7FFF) - (n >> 15);   // FFFF7001 to 17FFF.
   r = (r & 0x01FF) - (r >>  9);   // FFFFFF41 to 0x27F.
   r = (r & 0x003F) + (r >>  6);   // -2 to 72 (decimal).
   r = table[r + 2];
   return r - (((int)(n & -r) >> 31) & 9);
}
```

FIGURE 10–31. Signed remainder modulo 9, digit summing method.

10–19 Remainder by Multiplication and Shifting Right

The method described in this section applies in principle to all integer divisors greater than 2, but as a practical matter only to fairly small divisors and to divisors of the form $2^k - 1$. As in the preceding section, in most cases the code resorts to a table lookup after a fairly short calculation.

Unsigned Remainder

This section uses the mathematical (not computer algebra) notation $a \bmod b$, where a and b are integers and $b > 0$, to denote the integer x, $0 \le x < b$, that satisfies $x \equiv a \pmod{b}$.

To compute $n \bmod 3$, observe that

$$n \bmod 3 = \left\lfloor \frac{4}{3} n \right\rfloor \bmod 4. \tag{35}$$

Proof: Let $n = 3k + \delta$, where δ and k are integers and $0 \le \delta \le 2$. Then

$$\left\lfloor \frac{4}{3}(3k + \delta) \right\rfloor \bmod 4 = \left\lfloor 4k + \frac{4\delta}{3} \right\rfloor \bmod 4 = \left\lfloor \frac{4\delta}{3} \right\rfloor \bmod 4.$$

The value of the last expression is clearly 0, 1, or 2 for $\delta = 0$, 1, or 2 respectively. This allows changing the problem of computing the remainder modulo 3 to one of computing the remainder modulo 4, which is of course much easier on a binary computer.

Relations like (35) do not hold for all moduli, but similar relations do hold if the modulus is of the form $2^k - 1$ for k an integer greater than 1. For example, it is easy to show that

$$n \bmod 7 = \left\lfloor \frac{8}{7} n \right\rfloor \bmod 8.$$

For numbers not of the form $2^k - 1$, there is no such simple relation, but there is a certain uniqueness property that can be used to compute the remainder for other divisors. For example, if the divisor is 10 (decimal), consider the expression

$$\left\lfloor \frac{16}{10} n \right\rfloor \bmod 16. \tag{36}$$

Let $n = 10k + \delta$ where $0 \le \delta \le 9$. Then

$$\left\lfloor \frac{16}{10} n \right\rfloor \bmod 16 = \left\lfloor \frac{16}{10}(10k + \delta) \right\rfloor \bmod 16 = \left\lfloor \frac{16\delta}{10} \right\rfloor \bmod 16.$$

For $\delta = 0, 1, 2, 3, 4, 5, 6, 7, 8$, and 9, the last expression takes on the values 0, 1, 3, 4, 6, 8, 9, 11, 12, and 14 respectively. The latter numbers are all distinct. Thus if we can find a reasonably easy way to compute (36), we can translate 0 to 0, 1 to 1, 3 to 2, 4 to 3, and so on, to obtain the remainder of division by 10. This will generally require a translation table of size equal to the next power of 2 greater than the divisor, so the method is practical only for fairly small divisors (and for divisors of the form $2^k - 1$, for which table lookup is not required).

The code to be shown was derived by using a little of the above theory and a lot of trial and error.

Consider the remainder of unsigned division by 3. Following (35), we wish to compute the rightmost two bits of the integer part of $4n/3$. This can be done approximately by multiplying by $\lfloor 2^{32}/3 \rfloor$ and then dividing by 2^{30} by using a *shift right* instruction. When the multiplication by $\lfloor 2^{32}/3 \rfloor$ is done (using the *multiply* instruction that gives the low-order 32 bits of the product), high-order bits will be lost. But that doesn't matter, and in fact it's helpful, because we want the result modulo 4. Therefore, because $\lfloor 2^{32}/3 \rfloor$ = 0x55555555, a possible plan is to compute

$$\boldsymbol{r} \leftarrow (\mathbf{0x55555555} * \boldsymbol{n}) \overset{u}{\gg} 30.$$

Experiment indicates that this works for $\boldsymbol{n}$ in the range 0 to $2^{30} + 2$. Almost works, I should say; if $\boldsymbol{n}$ is nonzero and a multiple of 3, it gives the result 3. Thus it must be followed by a translation step that translates (0, 1, 2, 3) to (0, 1, 2, 0) respectively.

To extend the range of applicability, the multiplication must be done more accurately. Two more bits of accuracy suffices (that is, multiplying by **0x55555555.4**). The following calculation, followed by the translation step, works for all $\boldsymbol{n}$ representable as an unsigned 32-bit integer:

$$\boldsymbol{r} \leftarrow (\mathbf{0x55555555} * \boldsymbol{n} + (\boldsymbol{n} \overset{u}{\gg} 2)) \overset{u}{\gg} 30.$$

It is of course possible to give a formal proof of this, but the algebra is quite lengthy and error-prone.

The translation step can be done in three or four instructions on most machines, but there is a way to avoid it at a cost of two instructions. The above expression for computing $\boldsymbol{r}$ estimates low. If you estimate slightly high, the result is always 0, 1, or 2. This gives the C function shown below (eight instructions, including a *multiply*).

```
int remu3(unsigned n) {
   return (0x55555555*n + (n >> 1) - (n >> 3)) >> 30;
}
```

FIGURE 10–32. Unsigned remainder modulo 3, multiplication method.

The multiplication can be expanded, giving the following 13-instruction function that uses only *shift*'s and *add*'s.

```
int remu3(unsigned n) {
   unsigned r;

   r = n + (n << 2);
   r = r + (r << 4);
   r = r + (r << 8);
   r = r + (r << 16);
   r = r + (n >> 1);
   r = r - (n >> 3);
   return r >> 30;
}
```

FIGURE 10–33. Unsigned remainder modulo 3, multiplication (expanded) method.

The remainder of unsigned division by 5 may be computed very similarly to the remainder of division by 3. Let $n = 5k + r$ with $0 \le r \le 4$. Then $(8/5)n \bmod 8 = (8/5)(5k + r) \bmod 8 = (8/5)r \bmod 8$. For $r = 0, 1, 2, 3,$ and 4, this takes on the values 0, 1, 3, 4, and 6 respectively. Since $\lfloor 2^{32}/5 \rfloor$ = 0x33333333, this leads to the function shown in Figure 10–34 (11 instructions, including a *multiply*). The last step (code on the `return` statement) is mapping (0, 1, 3, 4, 6, 7) to (0, 1, 2, 3, 4, 0) respectively, using an in-register method rather than an indexed *load* from memory. By also mapping 2 to 2 and 5 to 4, the precision required in the multiplication by $2^{32}/5$ is reduced to using just the term `n >> 3` to approximate the missing part of the multiplier (hexadecimal 0.333…). If the "accuracy" term `n >> 3` is omitted, the code still works for `n` ranging from 0 to 0x60000004.

```
int remu5(unsigned n) {
   n = (0x33333333*n + (n >> 3)) >> 29;
   return (0x04432210 >> (n << 2)) & 7;
}
```

FIGURE 10–34. Unsigned remainder modulo 5, multiplication method.

The code for computing the unsigned remainder modulo 7 is similar, but the mapping step is simpler; it is necessary only to convert 7 to 0. One way to code it is shown in Figure 10–35 (11 instructions, including a *multiply*). If the accuracy term `n >> 4` is omitted, the code still works for `n` up to 0x40000006. With both accuracy terms omitted, it works for `n` up to 0x08000006.

```
int remu7(unsigned n) {
   n = (0x24924924*n + (n >> 1) + (n >> 4)) >> 29;
   return n & ((int)(n - 7) >> 31);
}
```

FIGURE 10–35. Unsigned remainder modulo 7, multiplication method.

Code for computing the unsigned remainder modulo 9 is shown in Figure 10–36. It is six instructions, including a *multiply*, plus an indexed *load*. If the accuracy term `n >> 1` is omitted and the multiplier is changed to 0x1C71C71D, the function works for `n` up to 0x1999999E.

```
int remu9(unsigned n) {
   static char table[16] = {0, 1, 1, 2, 2, 3, 3, 4,
                            5, 5, 6, 6, 7, 7, 8, 8};

   n = (0x1C71C71C*n + (n >> 1)) >> 28;
   return table[n];
}
```

FIGURE 10–36. Unsigned remainder modulo 9, multiplication method.

Figure 10–37 shows a way to compute the unsigned remainder modulo 10. It is eight instructions, including a *multiply*, plus an indexed *load* instruction. If the accuracy term `n >> 3` is omitted, the code works for `n` up to 0x40000004. If both accuracy terms are omitted, it works for `n` up to 0x0AAAAAAD.

```
int remu10(unsigned n) {
   static char table[16] = {0, 1, 2, 2, 3, 3, 4, 5,
                            5, 6, 7, 7, 8, 8, 9, 0};

   n = (0x19999999*n + (n >> 1) + (n >> 3)) >> 28;
   return table[n];
}
```

FIGURE 10–37. Unsigned remainder modulo 10, multiplication method.

As a final example, consider the computation of the remainder modulo 63. This function is used by the population count program at the top of page 68. Joe Keane [Keane] has come up with the rather mysterious code shown in Figure 10–38. It is 12 elementary instructions on the basic RISC.

The “multiply and shift right” method leads to the code shown in Figure 10–39. This is 11 instructions on the basic RISC, one being a multiply. This would not be as fast as Keane’s method unless the machine has a very fast multiply and the load of the constant 0x04104104 can move out of a loop. If the multiplication is expanded into shifts and adds, it becomes 15 elementary instructions.

```
int remu63(unsigned n) {
   unsigned t;

   t = (((n >> 12) + n) >> 10) + (n << 2);
   t = ((t >> 6) + t + 3) & 0xFF;
   return (t - (t >> 6)) >> 2;
}
```

FIGURE 10–38. Unsigned remainder modulo 63, Keane's method.

```
int remu63(unsigned n) {
   n = (0x04104104*n + (n >> 4) + (n >> 10)) >> 26;
   return n & ((n - 63) >> 6);    // Change 63 to 0.
}
```

FIGURE 10–39. Unsigned remainder modulo 63, multiplication method.

Timing tests were run on a 667 MHz Pentium III, using the GCC compiler with optimization level 2, and inlining the functions. The results varied substantially with the "environment" (the surrounding instructions) of the code being timed. For this machine and compiler, the multiplication method beat Keane's method by about five to 20 percent. With the multiplication expanded, the code sometimes did better and sometimes worse than both of the other two methods.

Signed Remainder

As in the case of the digit summing method, the "multiply and shift right" method can be adapted to compute the remainder resulting from signed division. Again, there seems to be no better way than to add a few steps to correct the result of the method as applied to unsigned division. For example, the code shown in Figure 10–40 is derived from Figure 10–32 on page 20 (12 instructions, including a *multiply*).

```
int rems3(int n) {
   unsigned r;

   r = n;
   r = (0x55555555*r + (r >> 1) - (r >> 3)) >> 30;
   return r - (((unsigned)n >> 31) << (r & 2));
}
```

FIGURE 10–40. Signed remainder modulo 3, multiplication method.

Some plausible ways to compute the remainder of signed division by 5, 7, 9, and 10 are shown in Figures 10–41 to 10–44. The code for a divisor of 7 uses quite a few extra instructions (19 in all, including a *multiply*); it might be preferable to use a table as is shown for the cases in which the divisor is 5, 9, or 10. In the latter cases the table used for unsigned division is doubled in size, with the sign bit of the divisor factored in to index the table. Entries shown as u are unused.

```
int rems5(int n) {
   unsigned r;
   static signed char table[16] = {0, 1, 2, 2, 3, u, 4, 0,
                                    u, 0,-4, u,-3,-2,-2,-1};

   r = n;
   r = ((0x33333333*r) + (r >> 3)) >> 29;
   return table[r + (((unsigned)n >> 31) << 3)];
}
```

FIGURE 10–41. Signed remainder modulo 5, multiplication method.

```
int rems7(int n) {
   unsigned r;

   r = n - (((unsigned)n >> 31) << 2);  // Fix for sign.
   r = ((0x24924924*r) + (r >> 1) + (r >> 4)) >> 29;
   r = r & ((int)(r - 7) >> 31);         // Change 7 to 0.
   return r - (((int)(n&-r) >> 31) & 7);// Fix n<0 case.
}
```

FIGURE 10–42. Signed remainder modulo 7, multiplication method.

```
int rems9(int n) {
   unsigned r;
   static signed char table[32] = {0, 1, 1, 2, u, 3, u, 4,
                                    5, 5, 6, 6, 7, u, 8, u,
                                   -4, u,-3, u,-2,-1,-1, 0,
                                    u,-8, u,-7,-6,-6,-5,-5};

   r = n;
   r = (0x1C71C71C*r + (r >> 1)) >> 28;
   return table[r + (((unsigned)n >> 31) << 4)];
}
```

FIGURE 10–43. Signed remainder modulo 9, multiplication method.

```
int rems10(int n) {
   unsigned r;
   static signed char table[32] = {0, 1, u, 2, 3, u, 4, 5,
                                    5, 6, u, 7, 8, u, 9, u,
                                   -6,-5, u,-4,-3,-3,-2, u,
                                   -1, 0, u,-9, u,-8,-7, u};
   r = n;
   r = (0x19999999*r + (r >> 1) + (r >> 3)) >> 28;
   return table[r + (((unsigned)n >> 31) << 4)];
}
```

FIGURE 10–44. Signed remainder modulo 10, multiplication method.

10–20 Converting to Exact Division

Since the remainder can be computed without computing the quotient, the possibility arises of computing the quotient $q = \lfloor n/d \rfloor$ by first computing the remainder, subtracting this from the dividend *n*, and then dividing the difference by the divisor *d*. This last division is an exact division, and it can be done by multiplying by the multiplicative inverse of *d* (see Section 10–15 on page 190). This method would be particularly attractive if both the quotient and remainder are wanted.

Let us try this for the case of unsigned division by 3. Computing the remainder by the multiplication method (Figure 10–32 on page 20) leads to the function shown in Figure 10–45.

```
unsigned divu3(unsigned n) {
   unsigned r;

   r = (0x55555555*n + (n >> 1) - (n >> 3)) >> 30;
   return (n - r)*0xAAAAAAAB;
}
```

FIGURE 10–45. Unsigned remainder and quotient with divisor = 3, using exact division.

This is 11 instructions, including two multiplications by large numbers. (The constant `0x55555555` can be generated by shifting the constant `0xAAAAAAAB` right one position.) In contrast, the more straightforward method of computing the quotient `q` using (for example) the code of Figure 10–6 on page 4, requires 14 instructions, including two multiplications by small numbers, or 17 elementary operations if the multiplications are expanded into *shift*'s and *add*'s. If the remainder is also wanted, and it is computed from `r = n - q*3`, the more straightforward method requires 16 instructions including three multiplications by small numbers, or 20 elementary instructions if the multiplications are expanded into *shift*'s and *add*'s.

The code of Figure 10–45 is not attractive if the multiplications are expanded into *shift*'s and *add*'s; the result is 24 elementary instructions. Thus the exact division method might be a good one on a machine that does not have *multiply high*, but it has a fast modulo 2^{32} *multiply* and slow *divide*, particularly if it can easily deal with the large constants.

For signed division by 3, the exact division method might be coded as shown in Figure 10–46. It is 15 instructions, including two multiplications by large constants.

```
int divs3(int n) {
   unsigned r;

   r = n;
   r = (0x55555555*r + (r >> 1) - (r >> 3)) >> 30;
   r = r - (((unsigned)n >> 31) << (r & 2));
   return (n - r)*0xAAAAAAAB;
}
```

FIGURE 10–46. Signed remainder and quotient with divisor = 3, using exact division.

As a final example, Figure 10–47 shows code for computing the quotient and remainder for unsigned division by 10. It is 12 instructions, including two multiplications by large constants, plus an indexed *load* instruction.

```
unsigned divu10(unsigned n) {
   unsigned r;
   static char table[16] = {0, 1, 2, 2, 3, 3, 4, 5,
                            5, 6, 7, 7, 8, 8, 9, 0};

   r = (0x19999999*n + (n >> 1) + (n >> 3)) >> 28;
   r = table[r];
   return ((n - r) >> 1)*0xCCCCCCCD;
}
```

FIGURE 10–47. Signed remainder and quotient with divisor = 10, using exact division.

10–21 A Timing Test

The Intel Pentium has a 32×32 ⇒ 64 *multiply* instruction, so one would expect that to divide by a constant such as 3, the code shown at the bottom of page 178 would be fastest. If that *multiply* instruction were not present, but the machine had a fast 32×32 ⇒ 32 *multiply* instruction, then the exact division method might be a good one, because the machine has a slow *divide* and a very fast *multiply.* To test this conjecture, an assembly language program was constructed to compare four methods of dividing by 3. The results are shown in Table 10–4.

The machine used was a 667 mHz Pentium III (ca. 2000).

The first row gives the time in cycles for just two instructions: an `xorl` to clear the left half of the 64-bit source register, and the `divl` instruction, which evidently takes 40 cycles. The second row also gives the time for just two instructions: *multiply* and *shift right 1* (`mull` and `shrl`). The third row gives the time for a sequence of 21 elementary instructions. It is the code of Figure 10–6 on page 4 using alternative 2, and with the multiplication by 3 done with a single

TABLE 10–4. UNSIGNED DIVIDE BY 3 ON A PENTIUM III

Division Method	**Cycles**
Using machine's divide instruction (`divl`)	41.08
Using 32×32 ⇒ 64 *multiply* (code on page 178)	4.28
All elementary instructions (Figure 10–6 on page 4)	14.10
Convert to exact division (Figure 10–45 on page 25)	6.68

instruction (`leal`). Several *move* instructions are necessary because the machine is (basically) two-address. The last row gives the time for a sequence of 10 instructions: two multiplications (`imull`) and the rest elementary. The two `imull` instructions use four-byte immediate fields for the large constants. (The signed *multiply* instruction `imull` is used, rather than its unsigned counterpart `mull`, because they give the same result in the low-order 32 bits and `imull` has more addressing modes available.)

The exact division method would be even more favorable compared to the second and third methods if both the quotient and remainder were wanted, because they would require additional code for the computation $\boldsymbol{r} \leftarrow \boldsymbol{n} - \boldsymbol{q} * 3$. (The `divl` instruction produces the remainder as well as the quotient.)

10–22 A Circuit for Dividing by 3

There is a simple circuit for dividing by 3 that is about as complex as a 32-bit adder. It can be constructed very similarly to the elementary way one constructs a 32-bit adder from 32 1-bit "full adder" circuits. However, in the divider signals flow from most significant to least significant bit.

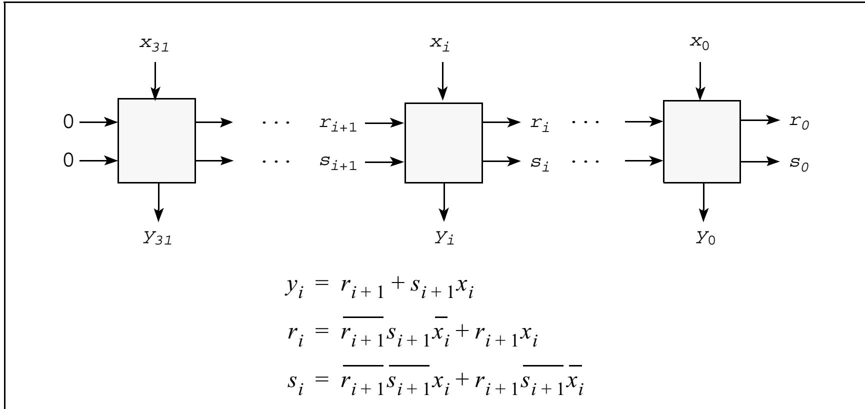
Consider dividing by 3 the way it is taught in grade school, but in binary. To produce each bit of the quotient, you divide 3 into the next bit, but the bit is preceded by a remainder of 0, 1, or 2 from the previous stage. The logic is shown in Table 10–5. Here the remainder is represented by two bits r_i and s_i, with r_i being

TABLE 10–5. LOGIC FOR DIVIDING BY 3

r_{i+1}	s_{i+1}	x_i	y_i	r_i	s_i
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	–	–	–
1	1	1	–	–	–

the most significant bit. The remainder is never 3, so the last two rows of the table represent "don't care" cases.

A circuit for dividing by 3 is shown in Figure 10–48. The quotient is the word consisting of bits y_{31} through y_0, and the remainder is $2r_0 + s_0$.



$$y_i = r_{i+1} + s_{i+1}x_i$$
$$r_i = \overline{r_{i+1}}s_{i+1}\overline{x_i} + r_{i+1}x_i$$
$$s_i = \overline{r_{i+1}}\,\overline{s_{i+1}}x_i + r_{i+1}\overline{s_{i+1}}\,\overline{x_i}$$

FIGURE 10–48. Logic circuit for dividing by 3.

Another way to implement the divide by 3 operation in hardware is to use the multiplier to multiply the dividend by the reciprocal of 3 (binary 0.010101…), with appropriate rounding and scaling. This is the technique shown on pages 158 and 178.

References

[Keane] Keane, Joe. Newsgroup sci.math.num-analysis, July 9, 1995.

[PB] Bonzini, Paolo. Private communication (email of December 27, 2003).